Guarded Page Tables on the MIPS R4600

Jochen Liedtke

GMD — German National Research
Center for Information Technology
GMD SET-RS, Schloß Birlinghoven,
53757 Sankt Augustin, Germany

email: jochen.liedtke@gmd.de

Kevin Elphinstone

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

email: kevine@vast.unsw.edu.au

UNSW-CSE-TR-9503 — 23 NOVEMBER 1995

Communicated by Jayasooriah

Abstract

The introduction of 64-bit microprocessors has increased demands placed on virtual memory systems. The availability of large address spaces has led to a flurry of new applications and operating systems that further stress virtual memory systems. Consequently, much interest has recently focussed on translation lookaside buffer (TLB) performance and page table efficiency. Guarded Page Tables are a mechanism for overcoming some of the problems associated with conventional page tables.

Guarded Page Tables are tree structured like conventional page tables. Also like conventional pages tables, they have the advantages of supporting hierarchical operations and sharing of sub-trees. Unlike conventional page tables, guarded page tables implement huge sparsely occupied address spaces efficiently.

We describe guarded page tables and the associated parsing algorithm. R4600 processor dependent micro-optimisation is undertaken and presented. R4600 TLB refill is discussed in detail, including a comparison of guarded page tables with more convention page tables. A software second level TLB is introduced and analysed as a way of increasing guarded page table performance.

1 Rationale

The advent of generally available 64-bit machines like the R4600[7] and DEC Alpha[14] has led to researchers proposing new ways to use virtual memory that previously was restricted by virtual address space size. Single address space systems such as Angel[12], Mungi[5], Nemisis[4], and Opal[2] and persistent operating systems such as Grasshopper [3] resulted from wide address space availability.

These systems, other proposals[1], and large UNIX style address spaces have increased VM demands in comparison to the traditional UNIX virtual memory model modern processors are designed to support. This has generated much interest in how to best support 64-bit address spaces[17, 16, 6, 15, 13]. This technical report explores the implementation of one proposed mechanism—Guarded Page Tables.

This work was originated as part of the Mungi [5] project at UNSW. Since it makes heavy use of a sparsely-occupied address space, the VM system must be targeted to support sparsity efficiently. We selected the Guarded Page Table mechanism (see section 2) which combines the advantages of multi-level and inverted page tables.

The critical point was whether the GPT mechanism could be implemented efficiently on the R4600 processor. Therefore, we developed R4600-specific GPT parsing algorithms (section 3) and complemented them with a second-level software TLB (section 5). How to best combine the elements, depends on both the concrete memory system (cache and memory timing) and the TLB-miss characteristics of the OS and applications. Therefore, we include a detailed performance discussion and make the software available as a tool box.

The purpose of this technical report is threefold:

1. It offers a tool box for experimenting with Guarded Page Tables on the R4600 processor.

2. It can be used as a guide for implementing Guarded Page Tables on other processors that support software-controlled TLBs.

3. Independent of the concrete problem, section 3 can serve as an example of architecture-dependent micro optimisation. An interesting result is that about 2/3 of the optimisation process – though architecture-dependent – can be made in terms of a high-level language and are based on algorithmic and data structure optimisations. The example shows that substantial performance gains (factors of 2.5 or more) are achievable by combining this method with specific assembler-level optimisations where general automatic code optimisation techniques do not help.

2 Guarded Page Tables

Guarded Page Tables have been described in [9, 10]. They combine the advantages of tree-structured multi-level page tables and hashed page tables: unlimited sparsity (2 page table entries per mapped page are always sufficient), tree structure (subtree sharing, hierarchical operations) and multiple page sizes. These properties are described in more detail in [8, 11]. Here we give only a short sketch of the basic mechanism.

The main problem with multilevel page tables is sparsity: we need huge amounts of page table entries for non-mapped pages. In the following example, the mapping of page 11 10 11 00 in a sparsely occupied address space is shown. (For demonstration purposes we use very small addresses and small page tables. Nil pointers are marked by "•".) The second- and third-level page table are extremely sparse page tables: each contains one single non-nil entry. Consequently, there is only one valid path through these two tables: when the leftmost two bits are "11", the subsequent address bits must be "10 11"; all other addresses lead to page faults. As shown in Figure 1, we can omit the two page tables and skip the associated translation steps.

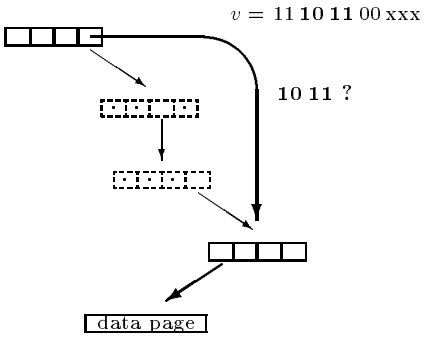


Figure 1: *Guarded Page Tables.*

Whenever entry 3 of the top-level page table is reached, we have to check whether "10 11" is a prefix of the remaining address. If so, this prefix can be stripped off, and the translation process can directly continue at the level-4 page table.

Therefore, each entry is augmented with a bit string g of variable length, which is referred to as a *guard*. This is the key idea of *guarded page tables*.

The translation process works as follows: first, a page table entry is selected by the highest part of the virtual address upon each transformation step in the same way as in the conventional multi-level page table method. The selected entry however contains not only a pointer (and perhaps an access attribute) but also the guard g. If g is a prefix of the remaining virtual address, the translation process either continues with the remaining postfix or terminates with the postfix as page offset. As an example, Figure 2

presents the transformation of 20 address bits by 3 page tables. Note that

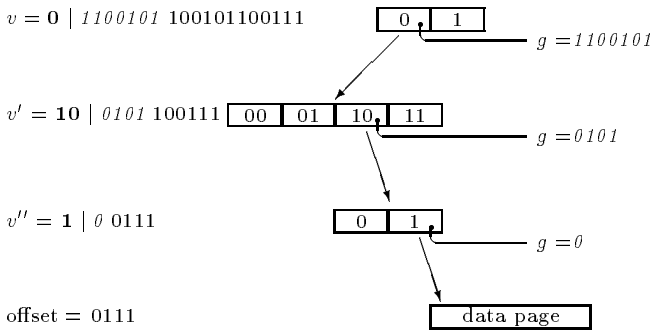


Figure 2: *Guarded Page Table Tree*

the length of the guards may vary from entry to entry. Furthermore, page table sizes can be mixed; all powers of 2 are admissible. The same holds for data pages, i.e., a mixture of 2-, 4-, ...1024-, ...entry page tables and pages can be used.

Guarded page tables contain conventional tables as a special case: if a guard has length zero, a translation step works exactly like in the conventional mechanism. However in all cases conventionally requiring a table with only one valid entry, a guard can be used instead. It can even replace a sequence of such "single-entry" page tables. This saves both memory capacity and transformation steps, i.e., guards act as a *shortcut*.

3 GPT Parser

At first, we describe a GPT translation step in general, independent of concrete hardware (see Figure 3). Here, v is the part of the original virtual address that is still subject to translation, and the pair (p, s) determines the page table (p: physical address, s: $\log_2$ of table size) that has to be used for the current translation step. The result of this step is either a new page table (p', s') and a postfix v' of v, or the data page (p', s') and offset v'.

The translations step starts by extracting u, the uppermost s bits of v. u is used for indexing the page table. The addressed entry specifies a guard g of variable size, i.e. possibly empty, which is checked against the remaining bits of the virtual address ($w = g$). When equal, the remaining v' is either used for the next level translation, or as the offset part. This operates as a shortcut, since not only u, but both u and w are stripped off the virtual address in one step; no table is necessary to decode w.

Note that the width of u, (determined by the page table size), may vary from step to step and that the size of w may differ from entry to entry.

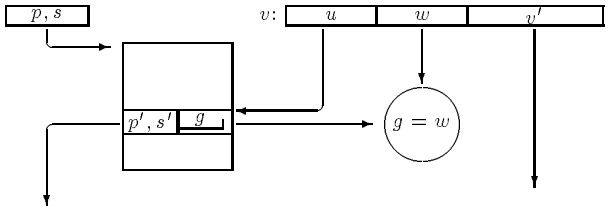


Figure 3: *Guarded Translation Step*

In the following, we use $|x|$ to denote the bit length of a flexible bit string x. For improved clarity, we always use x' for an item that belongs to next translation step (i.e., refers to the next lower level page table) and x for an item belonging to the current level.

Assuming at first 32-byte page table entries (later this is reduced to 16 bytes), one GPT translation step is:

$u := v \gg (|v| - s)$;
$g := [p + 32u].\text{guard}$;
if $g = ((v \gg (|v| - s - |g|))$ AND $(2^{|g|} - 1)$
  then $v' := v$ AND $2^{|v|-s-|g|} - 1$;
    $s' := [p + 32u].\text{size}'$;
    $p' := [p + 32u].\text{table}'$;
  else page_fault
fi .

This algorithm cannot be implemented 'as is', because the R4600 processor does not support variable length bit strings as a basic data type. Therefore, we have to hold $|v|$ and $|g|$ in additional variables v_{len} and g_{len}:

$u := v \gg (v_{len} - s)$;
$g := [p + 32u].\text{guard}$;
$g_{len} := [p + 32u].\text{guard\_len}$;
if $g = (v \gg (v_{len} - s - g_{len}))$ AND $(2^{g_{len}} - 1)$
  then $v'_{len} := v_{len} - s - g_{len}$;
    $v' := v$ AND $2^{v'_{len}} - 1$;
    $s' := [p + 32u].\text{size}'$;
    $p' := [p + 32u].\text{table}'$;
  else page_fault
fi .

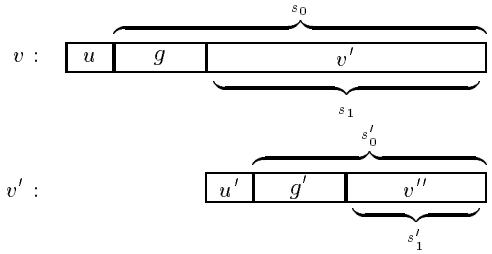
After eliminating common subexpressions, this algorithm requires 17 arithmetic and load operations.

3.1 From 17 To 10 Operations

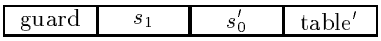
Note that although v is an input variable of the translation process, the length $|v|$ is a constant which is determined by the depth of the table in the GPT tree. Furthermore, the table size s and the guard length $|g|$ are fixed per page table entry. So the values

$$\begin{aligned} s_0 &= v_{len} - s \\ s_1 &= v_{len} - s - g_{len} \\ g_{mask} &= 2^{g_{len}} - 1 \end{aligned}$$

meaning



can be computed when constructing a GPT entry and can be stored per entry. Note that we have to store the *present level's* s_1 and the *next level's* s'_0 in a page table entry:

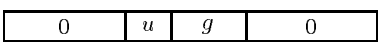


Fortunately, s'_0 can be as easily determined as s_0, as $s'_0 = v'_{len} - s' = v_{len} - s - g_{len} - s' = s_1 - s'$. The improved algorithm

$u := v \gg s_0$;
$g := [p + 32u].\text{guard}$;
$g_{mask} := [p + 32u].g_{mask}$;
$s_1 := [p + 32u].s_1$;
if $g = (v \gg s_1)$ AND g_{mask}
  then $v' := v$ AND $2^{s_1} - 1$;
    $s'_0 := [p + 32u].s'_0$;
    $p' := [p + 32u].\text{table}'$;
  else page_fault
fi .

requires only 14 arithmetic/load operations and no longer needs the variable v_{len}.

The next optimisation is based on the idea of adjusting the guard bits in the GPT entry variable and extending it by the number u of this entry



so that XORing v with this field removes u and g in one step and avoids one shift and one add operation. More precisely, we store the *extended guard*

$$G \;=\; ((u \ll |g|) + g) \ll (|v| - s - |g|)$$

in each page table entry instead of the guard g. The resulting algorithm

$u := v \gg s_0$;
$G := [p + 32u]$.extended_guard ;
$s_1 := [p + 32u].s_1$;
if $(v$ XOR $G) \gg s_1 \;=\; 0$
    then $v' := v$ XOR G ;
        $s_0' := [p + 32u].s_0'$;
        $p' := [p + 32u]$.table ;
    else page_fault
fi .

requires only 10 arithmetic/load operations and avoids the per entry field g_{mask}.

Up to this point, we have looked at only one translation step. For a complete translation, a loop is required. To approximate an until-loop, we first move the then-part statements before the if statement. This is possible because these three statements do not destroy data required later:

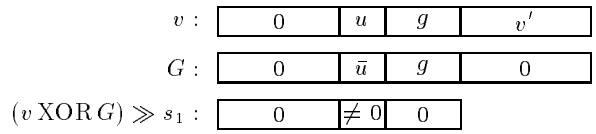
$u := v \gg s_0$;
$G := [p + 32u]$.extended_guard ;
$s_0' := [p + 32u].s_0'$;
$s_1 := [p + 32u].s_1$;
$p' := [p + 32u]$.table ;
$v' := v$ XOR G ;
if $v' \gg s_1 \;\neq\; 0$
    then page_fault
fi .

Unifying p', v' and s_0' with p, v and s_0, we get a very simple loop:

do
$u := v \gg s_0$;
$G := [p + 32u].\text{extended\_guard}$;
$s_0 := [p + 32u].s_0'$;
$s_1 := [p + 32u].s_1$;
$p := [p + 32u].\text{table}$;
$v := v$ XOR G ;
until $v \gg s_1 \neq 0$ **od** ;

The loop terminates when a page fault, i.e. a guard mismatch, is detected. Of course, the translation process must also terminate in the positive case, i.e. if the translation finishes without page fault. Adding a further termination condition to the loop would increase our costs per translation step.

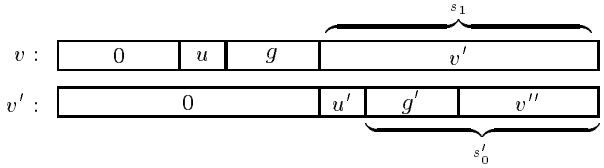
A better solution is to introduce a *pseudo mismatch* at leaf page table entries. We need an extended guard G, which includes the matching guard g, which in all cases leads to a mismatch, i.e. $(v \text{ XOR } G) \gg s_1 \neq 0$. Now recall that the extended guard of the uth entry of a page table always contains the index u. Therefore, we can achieve a pseudo mismatch by using an "incorrect" u for building the extended guard. $G = ((\bar{u} \ll |g|) + g) \ll s_1$ with $\bar{u} \neq u$ always leads to a mismatch:



The loop terminates either due to detecting a page fault or a leaf entry. In the case of

$$(v \gg s_1) \ll (64 - |g|) = 0 ,$$

we have a pseudo mismatch, i.e. a successful translation. For the mentioned check, we need a field holding the value $64 - |g|$. In leaf entries, the s_0'-field is free and can be used for this purpose. Then, $(v \gg s_1) \ll s_0'$ differentiates between true mismatch and pseudo mismatch, *if the current entry is a leaf entry.* We have to check, whether a mismatch at an higher level entry (which does not hold $64 - |g|$ in its s_0'-field) is also classified as a true mismatch. Fortunately, $(v \gg s_1) \ll s_0'$ evaluates always to non zero in this case, since s_0' is always less than s_1:



Concluding, the loop can be complemented by

if $(v \gg s_1) \ll s_0 = 0$
  then page_frame_addr $:= p$;
    page_frame_size $:= s_1$
  else *page_fault*
fi .

so that in the case of successful termination, $s1$ determines the size and p the physical address of the page.

3.2 R4600 Implementation

Before presenting an actual implementation of GPT parsing, a brief R4600 introduction is necessary. The R4600 is a member of the MIPS R4000 family of processors which feature 64-bit integer and floating point operations. They have thirty-two general purpose 64-bit registers of which two are special. Register *r0* ignores writes and always returns zero when read. Register *r31* is used to store the return address of Jump And Link (JAL) instructions.

The R4600 has a primary 16KB instruction cache and a 16KB data cache on chip. Both caches are two-way set associative, use a 32 byte line size, and FIFO replacement within a set. Secondary cache is external and optional.

A four (64-bit) word write buffer is used to buffer writes to external memory arising from cache write-back, cache write-through, and un-cached stores. This enables the processor to proceed in parallel while external memory is updated.

The R4600 has a five stage pipeline which has a one cycle latency for computational instructions. Computational instructions perform arithmetic, logical, and shifting operations using register operands or a register operand and a 16-bit signed immediate.

Load instructions do not allow the instruction immediately following, termed the *load delay slot*, to use the result of the load, thus giving a load latency of two cycles. Scheduling of instructions in the delay slot is desirable for increased throughput, though not strictly required, as the pipeline will slip one cycle in the case of a dependent instruction in the delay slot.

All jump and branch instructions have a latency of 2 cycles. The instruction in the *delay slot* following the jump is executed while the target of the jump is being fetched. The exception being if a *conditional branch likely* instruction is not taken, in which case the delay slot instruction is nullified.

3.3 From 11 To 8 Instructions

For the R4600 implementation, four 64-bit registers are needed. We name them r1, r2, v and P. A first compilation of the algorithm leads to 11 in-

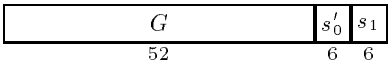
structions per translation step:

```
do:
        shr     r2,v,r2         u := v ≫ s0
        shl     r2,5            32u
        add     P,r2            p + 32u
        ld      r1,[P].ext_guard
        ld      r2,[P].s0
        xor     v,r1            v := v XOR G
        ld      r1,[P].s1
        ld      P,[P].table
        shr     r1,v,r1         v ≫ s1
        bz      r1,do

shl     r1,r2               (v ≫ s1) ≪ s0'
bnz     r1,page_fault
```

Note that all load delay slots in this (and the following) versions are filled with useful operations, i.e. do not cost additional cycles. By using appropriate coding[1], the same holds for the branch delay slot.

Further optimising, we use the fact that the R4600's minimal page size is 4K and the range of s_0' and s_1 is always $0 \ldots 63$. Therefore $2 \times 6 = 12$ bits are sufficient for s_0' and s_1 and since the 12 least significant bits of G are never used, we combine these three fields in one 64-bit word:



The second 64-bit word is used for pointing to the next level table (or data page). By this, we avoid load instructions and reduce the page table entry size to 16 bytes. The resulting code

[1]Use the `bzl` instruction which nullifies the immediately following instruction if the branch is *not* taken:

```
        shr     r2,v,r2
do:     shl     r2,5
        ...
        bzl     r1,do
        shr     r2,v,r2
```

```
do:     shr     r2,v,r2         u := v ≫ s0
        shl     r2,4            16u
        add     P,r2            p + 16u
        ld      r1,[P]          r1 := (G, s0', s1)
        ld      p,[P].table
        xor[2]  v,r1            v := v XOR G
        shr     r2,r1,6         r2 := s0'
        shr     r1,v,r1         v ≫ s1
        bz      r1,do
```

requires only 9 instructions per translation step.

The instruction '`shl r2,4`' is somehow annoying, because it is only used for setting the 4 lowest bits to zero. Without this requirement, we could have stored $s_0' - 4$ instead of s_0' in the s_0'-fields so that the previous `shr` instruction already includes the multiplication with 16. Indeed, it is not necessary that the 4 lowest bits are zero. It is sufficient that the 4 lowest bits of p after the addition have a *fixed value* which does not depend on the value of the actual v. This can be achieved by

```
or    p,r2
```

instead of adding, provided that the 4 lowest bits of p are always 1111. Therefore, we store $p + 15$ instead of p in the table-fields and always use `P-15` instead of `P` for addressing a table or table entry.

```
do:     shr     r2,v,r2         r2 := v ≫ (s0 − 4)
        or      P,r2            p + 16u + 15
        ld      r1,[P-15]       r1 := (G, s0' − 4, s1)
        ld      P,[P-15].table
        xor     v,r1            v := v XOR G
        shr     r2,r1,6         r2 := s0'
        shr     r1,v,r1         v ≫ s1
        bz      r1,do
```

The final code requires only 8 instructions per translation step.

3.4 Timing

Since no instruction interlocks are effective in the algorithm, i.e. since all delay slots are filled with useful instructions, for an n-step guarded page table walk, the single-issue R4600 processor needs

[2]Note that although '`xor v,r1`' destroys the 12 lowest bits of v (the 12 lowest bits of `r1` contain s_0' and s_1), it does not affect the algorithm, since these bits certainly belong to the offset part of the virtual address and are not required for translation.

$$8n \ \ldots \ (8+p)n \quad \text{cycles},$$

where p is the penalty of accessing a page table entry which is not in the primary data cache. If the table walking code is not in the instruction cache, another $2p$ penalty cycles may occur.

Since, within one address space, the R4600 supports 40-bit addresses and the smallest page is 4K, no more than $(40-12)/4 = 7$ translation steps should be necessary [8] per translation. Recall that the required steps can vary from page to page. Less than 7 steps are required in very sparse or in contiguous regions. It seems reasonable to expect 3 to 7 steps, depending on OS strategy and type of application. Assuming 4 cycles penalty for a cache miss, this corresponds to costs of [24...36] (3 steps) up to [56...84] (7 steps) cycles per GPT walk.

3.5 Adding Access Rights

So far, the algorithms presented are not sensitive to page-level access rights. Typical access rights are 'write permit' and 'user permit'. Without 'write permit', a page is mapped read only; without 'user permit', it can only be accessed by the kernel. For including access rights into the guarded translation mechanism, we developed 4 different scenarios.

Stable page access rights. We assume that access rights may differ from page to page but are changed infrequently, typically upon initial page mapping and copy on write, i.e. once or twice in the life of a mapped page.

Holding access rights only in leaf entries and checking them at the end of each table walk is sufficient in this case. This solution is very cheap (2 cycles, no additional per-step cycles). However, short term (transient) changing access rights of large regions (read/write $\rightarrow$ read only) is not possible, although short term locking (present $\rightarrow$ not present $\rightarrow$ present) is possible.

Cheap space If space is cheap enough and there are only few different access types, we can use one GPT tree per access type. Assume that read and write are the only access types and that read only and read/write are the possible access rights. Then we use two GPT trees: any read access is translated by the *read* tree, any write access by the *write* tree. The read tree holds all present pages, i.e. read only and read/write mapped ones, and the write tree only the read/write mapped pages. If 50% of the accessible pages are mapped read/write, we need about 1.5 times as much space as in the case of a single GPT, i.e. 3 GPT entries per mapped page instead of 2. The translation process is not explicitly slowed down by access right checks,

but the working set used for GPT parsing increases. Dependent on OS and application characteristics, this may or may not lead to a higher cache miss rate.

In many systems, we have to differentiate between more than two access types and to deal with more access right values. Common access types are:

user	read
user	write
kernel	read
kernel	write

Correspondingly, the access right values are

user/kernel	read only
user/kernel	read/write
kernel only	read only
kernel only	read/write

In this case, we need 4 GPT trees. Provided that 50% of the pages are mapped read/write and 5% kernel only accessible, we need about 2.9 times as much space as for a single GPT.

Cheap cycles If execution cycles are cheap enough to permit one additional cycle per GPT translation step, we can do access right checking per translation step. For this purpose, we slightly restrict the address space size and use the uppermost bits of the guard word G for the access right. In the above example, we need 2 bits:

bit 63	write deny	$= 1$	read only
		$= 0$	read/write permitted
bit 62	user deny	$= 1$	kernel only accessible
		$= 0$	user/kernel accessible

Correspondingly, the actual access type is held in an access register a:

bit 63	write	$= 1$	write access
		$= 0$	read access
bit 62	user	$= 1$	user access
		$= 0$	kernel access
bit 61		$= 1$	
$\vdots$		$= 1$	reserved, all 1's
bit 0		$= 1$	

Per translation step, the guard word G is anded by the access register a:

```
do:     shr     r2,v,r2          r2 := v >> (s0 - 4)
        or      P,r2             p + 16u + 15
        ld      r1,[P-15]        r1 := (G, s0' - 4, s1)
        ld      P,[P-15].table
        and     r1,a
        xor     v,r1             v := v XOR G
        shr     r2,r1,6          r2 := s0'
        shr     r1,v,r1          v >> s1
        bz      r1,do
```

Any access right violation now leads to pseudo guard mismatch, terminates the translation process and can further analysed in the page fault branch. The additional costs are 1 cycle per translation step.

4 R4600 Memory Management

An introduction to R4600 memory management is needed before presenting further details of GPT implementation. The R4000 architecture has a 64-bit virtual address space, however the R4600 only implements a 1TB (40-bit) user mode virtual address space together with a 64 GB physical address space. It uses a joint translation look-aside buffer (JTLB) to translate instruction and data virtual memory references to physical memory references.

The JTLB is a 48 entry fully associative memory. Each entry maps an even-odd pair of virtual pages to their corresponding physical addresses, giving a potential of 96 mapped virtual pages. Page size is per entry configurable from 4KB to 16MB in multiples of 4.

An 8 bit address space identifier (ASID) is associated with each entry in the JTLB. The ASID is used together with the virtual address when checking for a match, thus allowing multiple address spaces in the JTLB simultaneously, which reduces the need for JTLB flushing during context switching.

The R4600 also contains a 2 entry instruction TLB (ITLB) and a 4 entry data TLB (DTLB), with each entry mapping a 4KB page. ITLB and DTLB misses are automatically refilled from the JTLB making operation of the ITLB and DTLB transparent to users.

The handling of JTLB misses is via a *TLB Refill* exception and a software routine to load a new entry into the JTLB. Other TLB related exceptions are handled by the processor general exception mechanism, alleviating the TLB refill routine from determining the exception involved, and thus allowing it to be optimised solely for refill. Refill software can overwrite selected TLB entries or use a hardware provided mechanism to overwrite a randomly selected entry.

4.1 TLB Refill in Detail

TLB refill has been measured contributing up to 40% of total execution time[6] in some applications. While such high contributions are not normal, it is none the less important to minimise TLB refill costs as much as possible.

Before presenting or analysing any TLB refill routines, the basic cost of taking a null exception (C_{excpt}) needs to be determined. This is the cost of taking an exception that simply performs an exception return (`eret`) instruction. An exception generating instruction causes execution to begin, at the appropriate exception vector, when it reaches the fifth stage of the pipeline[7]: cost 4 cycles. Assuming `eret` has a delay slot similar to a branch or jump, it costs 2 cycles. Thus $C_{excpt} = 6$ cycles.

Refill—Virtual Array To serve as a reference, the best case TLB refill is presented. However before presentation, four coprocessor 0 (CP0) registers need introducing.

MIPS designers provide limited hardware support to speed up the software refill process via the *Context* or *XContext* registers. The *Context* register is a 32 bit version of the 64 bit *XContext* register, which is described below.

The *XContext* register illustrated in Figure 4, contains an operating system set-able Page Table Entry Base (PTEBase) field which is used to store the base of a page table array. Upon a TLB miss, the BadVPN2 field is set to the virtual page-pair number that misses. For 4K pages, the register can simply be used as the address of a page table entry pair to be loaded into the TLB. The format of page table entries are the same as *EntryLo* registers.

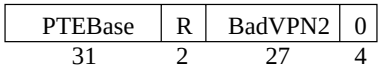


Figure 4: XContext Register Format

EntryLo0 and *EntryLo1* are identical registers used for reading and writing the physical page numbers into and out of the TLB, including TLB misses. *EntryLo* contains the physical frame number (PFN), cache coherency attributes (C), dirty bit (D), valid bit (V) and global bit (G) as illustrated in Figure 5.
The best case TLB refill routine is:

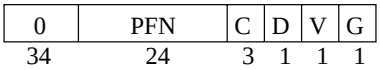


Figure 5: EntryLo0 and EntryLo1 Register Format

```
dmfc0  k0, XContext
nop
ld     k1,[k0]
ld     k0,[k0+8]
dmtc0  k1,EntryLo0
dmtc0  k0,EntryLo1
nop
tlbwr  ; 1 cycle slip[7]
```

Assuming the ideal situation, no cache misses and no second level TLB misses on the virtual array, the timing of the routine is 9 cycles. Hence the cost of the best case TLB refill (C_{best}) is:

$$\begin{aligned} C_{best} &= C_{excpt} + 9 \\ &= 15 \end{aligned}$$

Refill—Skeleton Before presenting more complicated refill routines, the following TLB refill skeleton is factored out as it is common in all routines presented later.

The skeleton loads the miss address from a CP0 register and frees an extra register. After page table entries are loaded it: loads the page entries into *EntryLo* registers, writes the TLB, and restores the freed register.

```
dmfc0  k0,CP0_reg
lui    k1,0x8000
sd     at,[k1].save_offset
  ⋮
dmtc0  k1,EntryLo0
dmtc0  k0,EntryLo1
lui    k1,0x8000
tlbwr       ; 1 cycle slip
ld     at,[k1].save_offset
```

The timing of the skeleton(C_{skel}) is 9 cycles. If extra registers are needed for page table lookup, it costs 2 cycles per register (C_{xreg}).

Refill—GPT Firstly, GPT translation is modified slightly. Instead of translation terminating with `P` pointing to the physical address, it finishes with `P` pointing to and even-odd pair of page table entries suitable for direct loading into *EntryLo*.

Using the skeleton above, with *BadVAddr* as the `CP0_reg` (which contains the address at which the TLB miss occured), the GPT refill routine is:

```
          ⋮
ld      P,[r1].gpt_base
          ⋮
8 cycle GPT loop
          ⋮
shr     r1,r2
bnz     r1,page_fault
ld      r1,[P]
ld      r2,[P+8]
```

The timing of GPT refill (C_{gpt}), where n is the number of levels traversed in the page table, is:

$$\begin{aligned} C_{gpt} &= C_{excpt} + C_{skel} + C_{xreg} + 5 + 8n \\ &= 6 + 9 + 2 + 5 + 8n \\ &= 22 + 8n \end{aligned}$$

For the 3 level lookup $C_{gpt3} = 46$ cycles, for a 7 level lookup $C_{gpt7} = 78$ cycles.

Cache Effects So far it has been assumed that all data and instructions are in cache. Instruction cache misses will have similar effects on all refill routines with the penalty being proportional to the length of the routine. However, data cache misses have the potential to show large differences between any two refill routines as the amount of data accessed can vary markedly.

Given a data cache penalty of 6 cycles for a single double-word, plus 2 cycles for each extra double word, up to 12 cycles for an entire cache line[3], data access can be expensive.

[3]These numbers represent the pipeline cycles wasted while running minimum external bus cycles to a secondary cache. The actual miss penalty due to a cache refill my be lower due to parallelism between refill and instruction execution, or much higher if no second level cache exists.

The best case routine assuming cache misses is $C_{best.m} = C_{best} + 8$. For the GPT routine $C_{gpt.m} = C_{gpt} + 8(n+1) + 6$. Table 1 shows the cost for the refill routines presented so far, assuming all data cache hits and then assuming all data cache misses.

Routine	Cache Hit	Cache Miss
C_{best}	15	23
C_{gpt3}	46	84
C_{gpt7}	78	148

Table 1: TLB refill routine cost (cycles).

Refill Comparison Direct comparison between C_{best} and C_{gpt} is fairly irrelevant as it does not take into account the frequency of TLB misses. In the extreme, it does not matter how long refill takes if the TLB never misses. To facilitate a more revealing comparison, we use the metric of *percentage of cycles due to TLB refill ($\%_{tlb}$) compared to total cycles*, which we aim to minimize. Distinguishing cycles due to TLB refill (C_{tlb}), and grouping other cycles (C_{other}) not related to TLB refill,

$$\%_{tlb} = \frac{C_{tlb}}{C_{tlb} + C_{other}} \times 100$$

Given a miss rate per C_{other} (r_{miss}) and TLB refill cost (C_{refill}):

$$\begin{aligned} C_{tlb} &= r_{miss} C_{other} C_{refill} \\ \%_{tlb} &= \frac{r_{miss} C_{other} C_{refill}}{r_{miss} C_{other} C_{refill} + C_{other}} \times 100 \\ &= \frac{r_{miss} C_{refill}}{r_{miss} C_{refill} + 1} \times 100 \end{aligned}$$

Figure 6 illustrates the TLB overhead associated with the six routines tabulated above, for various miss rates.

It can be seen that with miss rates less than 0.0001, it is largely irrelevant which routine is chosen for TLB refill, as refill's contribution to overall runtime is negligible.

In the case of high miss rates, for example 0.01, TLB overheads are significantly different. The best case routine overhead is expected to vary between 13% and 19%, however GPT overhead varies between 32% and 59%.

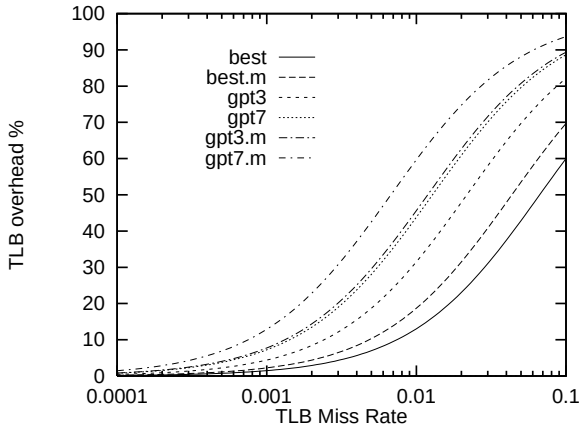


Figure 6: TLB overhead for TLB refill routines

Or to look at it differently, given a tolerable overhead of 10%, the best case routine can tolerate miss rates 2-10 times higher than GPT refill.

Thus it appears GPTs are unsuitable for TLB refill where it is expected that TLB miss rates may be high, especially if cache miss costs are also high.

5 The Second Level TLB

Ideally, a robust mechanism is needed that supports address space sparsity, fast lookup, hierarchical operations, and graceful performance degradation when faced with increasing TLB miss rates. A second level TLB in combination with GPTs should be the answer. The second level TLB (TLB2) is a software cache of page table entries used to refill the hardware TLB.

5.1 TLB2 Design Issues

5.1.1 Tagged or Per-Process

The first design decision to be made is whether TLB2 should be a per-process cache or a global, address space tagged, cache. A per-process cache slows the context switch time as the cache base address needs to be changed, though this may be insignificant when compared to other switching overheads.

A single tagged cache is more space efficient. A per-process cache takes n times the space for n processes for the same potential per-process cache capacity. A single tagged cache will adapt to the workload, caching only active TLB entries, whereas a per-process cache may itself be entirely inactive.

A single tagged cache is small enough to use unmapped physical memory. A per-process cache is more suited to implementation in virtual memory as the number of processes is unknown and potentially large. Virtual memory implementation requires handling of complex nested TLB misses which are avoided in the physical implementation.

Flushing all cache entries associated with a physical frame is simpler and faster with a single tagged cache, than with n per-process caches of similar size.

For these reasons, we choose to use a single tagged cache for TLB2.

5.1.2 Size

Performance dictates the size of TLB2. While a large TLB2 will reduce TLB2 miss rate, the following factors make it desirable to keep TLB2 small. TLB2 uses unmapped physical memory which is a limited resource, though it is expected that TLB2 will be small enough to effectively ignore this limitation.

TLB2 flushing becomes more expensive as size increases. Flushing can be on a per physical page frame basis, or on a per address space tag basis. These events occur, for example, on page frame swap-out and address space destruction respectively. These are expected to be infrequent operations when compared to TLB2 lookup, though they should be kept in mind when sizing TLB2.

The R4600 has 16-bit immediate operands. This gives a 16-bit mask operation or a load operation from a 64KB address space, in a single instruction. Larger masks or load offsets require multiple instructions. This needs to be kept in mind as TLB2 lookup is time critical. The performance gained by having a large cache may be offset by the extra time taken to access it.

5.1.3 Associativity

High associativity is desirable in a cache to decrease the likelihood of conflict misses. In a hardware cache implementation, n associativity requires n comparisons in parallel to determine a hit. In software, n associativity requires n comparisons in sequence. Sequential comparisons need to be minimised as TLB2 lookup is time critical. The tradeoff between increased lookup time due to sequential comparisons and decreased miss rate due to associativity needs to be carefully balanced.

5.2 A Direct-Mapped TLB2

Before describing a direct mapped TLB2, another CP0 register needs introducing. The *EntryHi* register is used to set the hardware lookup tag in a TLB entry when adding a new TLB entry or probing for an existing one.

It contains a virtual page number of a page-pair (VPN2) and an associated address space identifier (ASID) as illustrated in Figure 7. *EntryHi* is set on TLB miss to a value appropriate for adding a new entry into the TLB. It can also be set by the operating system when adding a TLB entry not associated with a TLB exception.

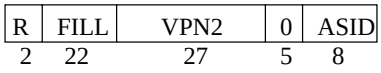


Figure 7: EntryHi Register Format

The structure of a TLB2 cache entry needs to contain the tag for matching with the *EntryHi* register, and an even-odd pair of page table entries for loading into *EntryLo0* and *EntryLo1*. A naive implementation would use three 64-bit words which makes indexing awkward.

This can be optimised by making use of the fact that the upper 34 bits of the page table entries are always zero. This allows two 32-bit page table entries to be stored in a single 64-bit word, giving a block size of two 64-bit words which is easily indexed in TLB2.

This optimisation costs nothing in terms of speed. The two 64-bit page table entries would be loaded using two "load double" instructions. The optimised 32-bit entries are loaded using two "load word" instructions which sign extend the values to 64-bit for free once loaded. By having two TLB2 blocks within a single 32 byte data cache line instead of one, the compact structure may indeed be faster as it reduces the chance of a data cache miss on load.

The refill routine to implement a 128K direct mapped TLB2 is:

```
          ⋮
shl     at,k0,44
shr     at,53
add     at,k1
ld      k1,[at+TLB2]
nop
bne     k1,k0,miss
lw      k1,[at+8+TLB2]
lw      k0,[at+12+TLB2]
          ⋮
```

The timing for a hit is $C_{excpt} + C_{skel} + 8 = 23$ cycles. A miss is a little more complicated as it includes a GPT lookup, and replacing the missed TLB2 entry (C_{repl}) . The cost is $C_{excpt} + C_{skel} + 7 + C_{gpt} + C_{repl}$. The TLB2

miss routine is:

```
miss:
        sd      k0,[at+TLB2]
        dmfc0   k0,BadVAddr
        lui     k1,0x8000
        sd      P,[k1].save_P
        ld      P,[k1].gpt_base
        sd      r2,[k1].save_r2
                  ⋮
        8 cycle GPT loop
                  ⋮
        shr     k1,r2
        bnz     k1,page_fault
        lw      k1,[P]
        lw      r2,[P+4]
        sw      k1,[at+8+TLB2]
        sw      r2,[at+12+TLB2]
                  ⋮
        ld      P,[k1].save_P
        ld      r2,[k1].save_r2
```

The timing for the miss routine is $14 + 8n$. The complete timing for a reload that misses TLB2 is $36+8n$. The same timing assuming a cache miss on every load is $56 + 16n$.

GPT level	Cache hits		Cache misses	
	hit	miss	hit	miss
3	23	60	31	104
7	23	92	31	168

Table 2: Direct mapped TLB2 costs

Now, assuming TLB2 is sized such that it has, on average, a 10% miss rate. The average timing for the case of a 3 level GPT translation assuming data cache hits is $0.9 * 23 + 0.1 * 60 = 26.7$. The worst case average timing assuming 7 level translation with cache misses is $0.9 * 31 + 0.1 * 168 = 44.7$.

With the assumption of 10% TLB2 miss rate, Figure 8 shows the TLB overhead for: best case refill (for comparison purposes), 3 level GPT refill using TLB2 and assuming cache hits, 7 level GPT refill using TLB2 with all cache misses, 3 level GPT refill assuming cache hits, and 7 level GPT refill assuming cache misses.

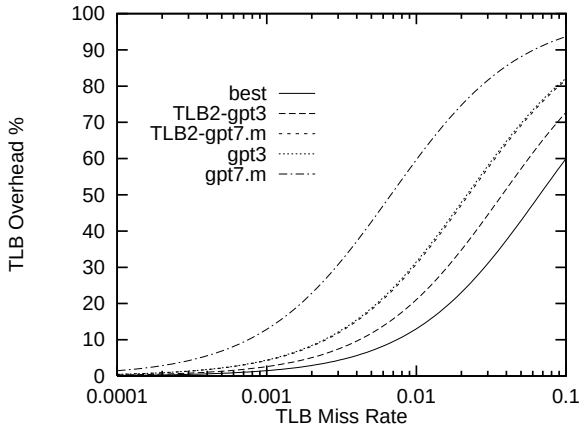


Figure 8: Direct mapped TLB2 overhead

It can be seen that the addition of TLB2 reduces the overhead of a 3-level refill from 32% to 21% at a miss rate of 0.01. This is a significant step towards the idealised best case refill which, in practice, is expected to be above the 13% illustrated. TLB2 also reduces the overhead associated with the worst case (all cache misses) 7 level refill from 60% to 31% at miss rate of 0.01.

Hence, TLB2 introduction has the desirable effects of increasing performance of expected normal case GPT refill, and limiting the effect of the worst case refill to a reasonable level.

5.2.1 The Sharing Problem

The ASID is used as part of the tag used in matching TLB2 entries allowing the same virtual page with different ASIDs to exist in TLB2 at any one time. However the ASID is not used as part of the indexing of TLB2, meaning that virtual pages with differing ASIDs will hash to the same entry. If TLB2 is direct mapped as in this case, the indexing precludes entries with the same virtual page number, even though they have differing ASIDs.

As a consequence, if the same address is shared between different address spaces, the potential for many conflict misses in TLB2 exists. The solution to this is obviously including the ASID in index formation at the expense of executing 3 extra instructions upon each refill.

Instead of directly indexing TLB2 using some masked region of the virtual address, we use a hash function which *xor*s the ASID and virtual address as illustrated below:

```
shl     at,k0,44
shl     k1,k0,57
xor     at,k1,at
shr     at,53
lui     k1,0x8000
```

5.3 4-way Associative TLB2

The speed advantage of the direct mapped TLB2 relies on a low miss rate. A low miss rate coupled with small TLB2 size may not be achievable with 1-way associativity. It may prove better to use a slightly slower 4-way associative structure to achieve a lower miss rate.

A 4-way TLB2 is similar to the direct mapped case in structure, except the basic block used above is grouped into sets of four. Indexing is done on a set basis, and each of the four blocks in the set is checked sequentially for a match.

Ignoring the extra comparisons needed, the basic routine is 3 cycles slower than the direct mapped case. 2 extra cycles are needed to dump and reload an extra register needed for the interleaved compare, the remaining extra cycle is due to a branch delay slot being unable to be filled with a useful operation.

The comparison operation is interleaved such that the load delay slot of one tag is used to load or compare another tag. This is illustrated in the assembly routine which follows.

```
                ⋮
        shr     at,k0,9
        and     at,0xffc0
        add     at,k1
        ld      k1,[at+TLB2]
        ld      t0,[at+16+TLB2]
        beq     k1,k0,hit1
        ld      k1,[at+32+TLB2]
        beq     t0,k0,hit2
        ld      t0,[at+48+TLB2]
        beq     k1,k0,hit3
        lw      k1,[at+56+TLB2]
        bne     t0,k0,miss
        lw      k0,[at+60+TLB2]
                ⋮
hit1:
        lw      k1,[at+8+TLB2]
        lw      k0,[at+12+TLB2]
                ⋮
```

Timing for a hit is $C_{excpt} + C_{skel} + C_{xreg} + 9, 11, 13, 13 = 26, 28, 30, 30$ cycles. Assuming equal likelihood of hit1, 2, 3 , and 4; the average timing is 28.5 cycles. For a miss, cost is 30+ miss routine.

The miss routine is similar to the direct mapped case, except selection of which entry in the set to replace is required. Pseudo-random replacement can be achieved by using the *Count* register, which simply increments at half the pipeline rate. The assembly to generate an appropriate offset follows.

```
        dmfc0   k1, Count
                ⋮
        andi    k1,0x0030
        add     at,k1,at
```

This extra work is offset slightly by the fact the 4-way comparison has freed one extra register requiring only one further register to be freed in the miss routine, instead of two, which was the case for a direct mapped miss. Timing for the miss routine is $15 + 8n$. The complete timing for a reload that misses TLB2, $45 + 8n$. The same timing assuming a cache miss on every load is $81 + 16n$.

Again assuming a 10% TLB2 miss rate the average refill cost is 32.55 cycles best case (3 level GPT translation, no cache misses) and 59.35 worst case (7 level GPT with all cache misses).

GPT level	Cache hits		Cache misses	
	hit	miss	hit	miss
3	28.5	69	44.5	129
7	28.5	101	44.5	193

Table 3: 4-way TLB2 costs

Figure 9 illustrates the TLB overheads for: best case refill, TLB2 best case refill, TLB2 worst case refill, 3 level GPT refill with no cache misses, and 7 level GPT refill with all cache misses.

The 4-way TLB2 in the best case does perform better than the straight 3 level GPT refill, and only slightly worse in the worse case. When compared to a direct mapped TLB2, the 4-way TLB2 is slightly poorer performing assuming the same miss rate, however the 4-way TLB2 is, in reality, likely to have a lower miss rate giving better performance in the case where conflict misses tend to dominate direct mapped TLB2 behaviour.

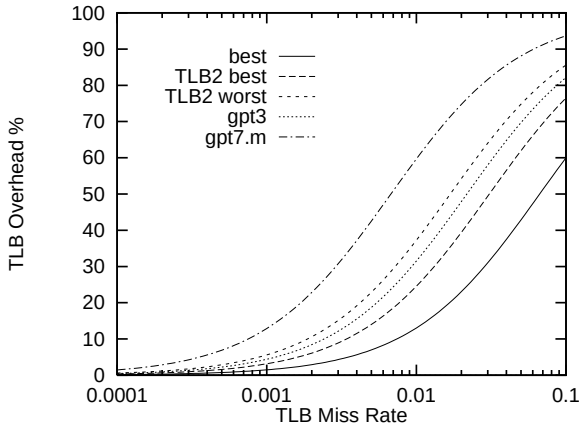


Figure 9: 4-way TLB2 overhead

Like in the direct-mapped case, there is potential for many conflict misses in the case where several address spaces share the same virtual address, as indexing does not include the ASID. The problem is less severe due to the higher associativity, however still exists. The solution is similar to the direct-mapped case, ie use a hashed index instead of a direct index, except that a spare register reduces the penalty per refill to 2 extra instructions.

```
shr     t0, k0, 7
shl     at, k0, 6
xor     at, at, t0
and     at,0xffc0
```

5.4 Summary

Table 4 summarizes the best case lookup times for TLB2s with various degrees of associativity and the two different indexing schemes. It assumes all cache hits.

Associativity	Direct Index	Hashed Index
1	23	26
2	26	28
4	28.5	30.5
8	32.75	34.75

Table 4: TLB2 lookup times

The directly indexed, 1-way TLB2 is the fastest and thus the TLB2 of choice assuming it satisfies sizing constraints and low sharing occurs.

In the case of sharing causing an unsatisfactory amount of conflict misses, then a hash indexed TLB2 of 1-way or 2-way associativity is the TLB2 of choice. Given that 2-way is only slightly slower than the 1-way TLB2, and has a significantly higher stochastic capacity, the 2-way is favoured.

It appears unlikely that a 4-way or 8-way TLB2 will outperform the 1-way or 2-way given that sizing contraints are unlikely to be restrictive, and hence a larger low associativity TLB2 is favourable to a smaller high associativity TLB2.

6 Concluding Remarks

This exploration of GPT implementation has shown them to be a viable alternative to conventional page tables on the R4600.

The presented software is available through the WorldWideWeb under

http://www.vast.unsw.edu.au/Mungi/Mungi.html.

References

[1] Alberto Bartoli, Sape J. Mullender, and Martijn van der Valk. Wide-address spaces — eploring the design space. Technical Report Pegasus paper 92–3, University of Cambridge Computer Laboratory, 1992.

[2] J. S. Chase, H. M. Levy, M. J. Freely, and E. D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, November 1994.

[3] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan. Grasshopper: An othogobnally persistent operating system. Technical Report GH-03, Basser Dept. Computer Science, University of Sydney, 1994.

[4] Robin Fairbairns. Pegasus summary report, kernel work package. Technical Report Pegasus paper 93–1, University of Cambridge Computer Laboratory, 1993.

[5] G. Heiser, K. Elphinstone, S. Russell, and G. R. Hellestrand. A distributed single address-space operating system supporting persistence. SCS&E Report 9302, Univ. of New South Wales, School of Computer Science, Kensington, Australia, March 1993.

[6] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[7] Integrated Device Technology, Inc. *IDT79R4600 ORION Hardware User's Manual*, October 1993.

[8] J. Liedtke. Some theorems about guarded page tables. Arbeitspapiere der GMD No. 792, German National Research Center for Computer Science (GMD), Sankt Augustin, 1993.

[9] J. Liedtke. Address space sparsity and fine granularity. In *6th SIGOPS European Workshop*, pages 78–81, Schloß Dagstuhl, Germany, September 1994. also in *Operating Systems Review 29*, 1 (Jan. 1995), 87–90.

[10] J. Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, pages xx–xx, xx 1994. also published as Arbeitspapier der GMD No. 872, German National Research Center for Computer Science (GMD), Sankt Augustin, 1993.

[11] J. Liedtke. Some theorems about restricted guarded page tables. Arbeitspapiere der GMD No. 834, German National Research Center for Computer Science (GMD), Sankt Augustin, 1994.

[12] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashley Saulsbury, Tom Stiemerling, and Paul Kelly. Design and Implementation of an Object-Oriented 64-bit Single Address Space Microkernel. Technical Report 9, SARC, Dept. Computer Science, City University, London, 1993.

[13] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software managed TLBs. In *20th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–38, San Diego, CA, May 1993.

[14] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, Maynard, M.A., 1992.

[15] M Talluri, S. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *19th Annual International Symposium on Computer Architecture (ISCA)*, pages 415–424, Gold Coast, Australia, May 1992.

[16] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[17] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proc. SOSP'95*, 1995.