Guarded Page Tables on Mips R4600
Or
An Exercise in Architecture-Dependent Micro Optimization

Jochen Liedtke

GMD — German National Research Center for Information Technology *

jochen.liedtke@gmd.de

Kevin Elphinstone

School of Computer Science University of New South Wales †

kevine@vast.unsw.edu.au

Abstract

Guarded Page Tables implement huge sparsely occupied address spaces efficiently and have the advantages of multi-level tables (tree structure, hierarchy, sharing). We present an implementation of guarded page tables on the R4600 processor. The paper describes both the architecture-dependent design process of the algorithms and the resulting tool box.

1 Rationale

This work was originated as part of the Mungi [2] project at UNSW, which aims to build an object-oriented single address space operating system. 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.

*GMD SET-RS, 53754 Sankt Augustin, Germany

†Sydney 2052, NSW, Australia

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

2 Guarded Page Tables

Guarded Page Tables have been described in [6, 7]. 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 more detailed in [5, 8]. 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. Look at the following

example where 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. 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.

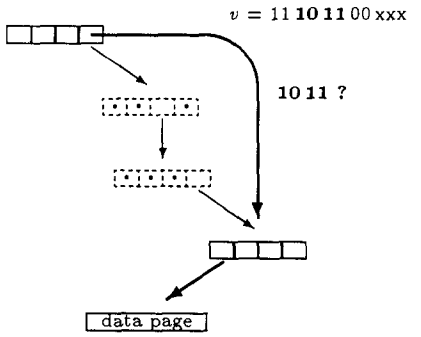


Figure 1: *Guarded Page Tables.*

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 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.

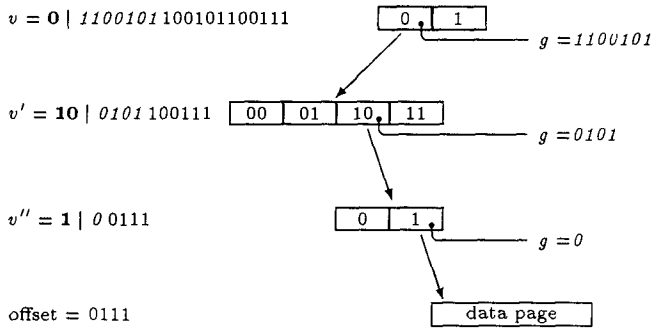


Figure 2: *Guarded Page Table Tree*

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.

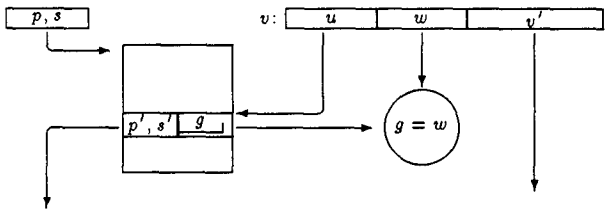


Figure 3: *Guarded Translation Step*

In the following parts, 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 (we hope to later reduce this 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 flexible 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 .

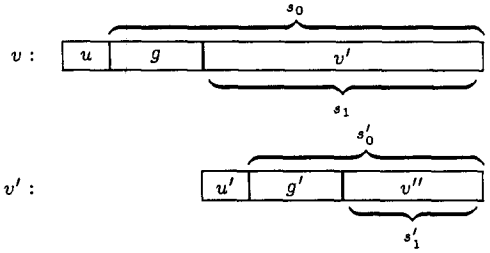
After eliminating common subexpressions, this algorithm requires 17 arithmetic and load operations.

3.1 From 17 To 10 Operations

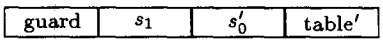
Note that although v is an input variable of the translation process, the length $|v|$ is a constant which is determined by the depth of the table in the GPT tree. Furthermore, the table size s and the guard length $|g|$ are fixed per page table entry. So the values

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

meaning



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

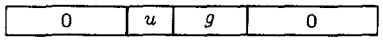


Fortunately, s'_0 can be as easily determined as s_0, as $s'_0 = v'_{len} - s' = v_{len} - s - g_{len} - s' = s_1 - s'$. The improved algorithm

$u := v \gg s_0$;
$g := [p + 32u].\text{guard}$;
$g_{mask} := [p + 32u].g_{mask}$;
$s_1 := [p + 32u].s_1$;
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 optimization 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 by 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

$u := v \gg s_0$;
$G := [p + 32u].\text{extended\_guard}$;
$s_1 := [p + 32u].s_1$;
if $(v \text{ XOR } G) \gg s_1 = 0$
  then $v' := v \text{ XOR } G$;
    $s_0' := [p + 32u].s_0'$;
    $p' := [p + 32u].\text{table}$;
  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 yet required data:

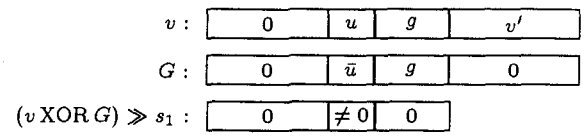
$u := v \gg s_0$;
$G := [p + 32u].\text{extended\_guard}$;
$s_0' := [p + 32u].s_0'$;
$s_1 := [p + 32u].s_1$;
$p' := [p + 32u].\text{table}$;
$v' := v \text{ XOR } G$;
if $v' \gg s_1 \neq 0$
  then page_fault
fi .

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

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

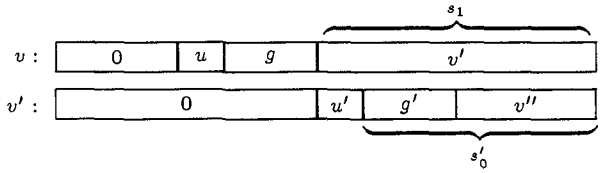
A better solution is to introduce a *pseudo mismatch* at leaf page table entries. We need an extended guard G, which includes the matching guard g, which in all cases leads to a mismatch, i.e. $(v \text{ XOR } G) \gg s_1 \neq 0$. Now recall that the extended guard of the u^{th} entry of a page table 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 a concrete 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 uncached 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 don't 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

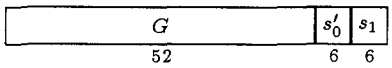
For the R4600 implementation, four 64-bit registers are needed. We name them r1, r2, v and P. A first compilation of the algorithm leads to 11 instructions per translation step:

```
do:
        srl     r2,v,r2          u := v ≫ s0
        sll     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
        srl     r1,v,r1          v ≫ s1
        bz      r1,do
sll     r1,r2            (v ≫ s1) ≪ s'0
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.

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

Further optimizating, 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 lowermost 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

```
do:   srl    r2,v,r2          u := v >> s0
      sll    r2,4             16u
      add    P,r2             p + 16u
      ld     r1,[P]           r1 := (G, s'0, s1)
      ld     p,[P].table
      xor[2] v,r1             v := v XOR G
      srl    r2,r1,6          r2 := s'0
      srl    r1,v,r1          v >> s1
      bz     r1,do
```

requires only 9 instructions per translation step.

The instruction '`sll r2,4`' is somehow anoying, 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 `srl` instruction already includes the multiplication with 16. Indeed, it is not necessary that the 4 lowest bits must be 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

```
xor   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:   srl    r2,v,r2          r2 := v >> (s0 - 4)
      or     P,r2             p + 16u + 15
      ld     r1,[P-15]        r1 := (G, s'0 - 4, s1)
      ld     P,[P-15].table
      xor    v,r1             v := v XOR G
      srl    r2,r1,6          r2 := s'0
      srl    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 senseful instructions, for an n-step guarded page table walk, the single-issue R4600 processor needs

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

where p is the penalty of accessing a page table entry which actually 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 [5] 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 i steps, depending on OS strategy and type of application. Assuming 4 cycles penalty for a cache miss, this corresponds to costs of $[24 \ldots 36]$ (3 steps) up to $[56 \ldots 84]$ (7 steps) cycles per GPT walk.

4 R4600 Memory Management

An introduction to R4600 memory management is needed before further presenting 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

```
      srl    r2,v,r2
do:   sll    r2,5
      ...
      bzl    r1,do
      srl    r2,v,r2
```

[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.

a joint translation lookaside 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, allowing it to be optimized 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[3] in some applications. While such high contributions are not normal, it is none the less important to mininize TLB refill costs as much as possible.

Before presenting or analyzing 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[4]: 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 setable 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.

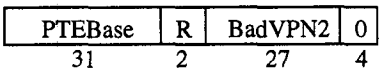


Figure 4: XContext Register Format

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.

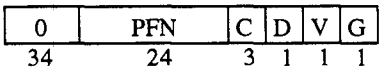


Figure 5: EntryLo0 and EntryLo1 Register Format

The best case TLB refill routine:

```
dmfc0  k0, XContext
nop
ld     k1,[k0]
ld     k0,[k0+8]
dmtc0  k1,EntryLo0
dmtc0  k0,EntryLo1
nop
tlbwr  ; 1 cycle slip[4]
```

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:

$$C_{best} = C_{excpt} + 9 = 15$$

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
        ⋮
srl    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 the two refill routines as the amount of data accessed varies markedly.

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

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 show the cost for the refill routines presented so far, assuming all data cache hits and then assuming all data cache misses.

Refill Comparison Direct comparison between C_{best} and C_{gpt} is fairly irrelevent as it does not take

[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.

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

Table 1: TLB refill routine cost (cycles).

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. Assumming cycles due to TLB refill (C_{tlb}), and grouping other cycles (C_{other}) not related to TLB refill,

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

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

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

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

For avoiding misunderstandings, we explicitly mention:

- The miss rate we used is neither the TLB miss rate per memory access nor per instruction. Instead, we use the miss cost per cycle that is not related to TLB miss. Cum grano salis, these are instruction execution and cache miss cycles. For illustration: assume an application with a TLB miss rate per LD/ST instruction of 1% (which is high), on average one LD/ST per 3 instructions and 5% cache miss rate (8 cycles penalty). Then one TLB miss occurs per 340 cycles, i.e. our TLB miss rate r_{miss} is 0.003.

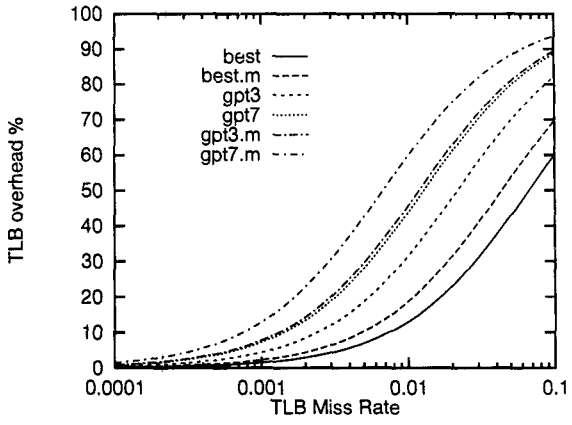


Figure 6: TLB overhead for TLB refill routines

- We use the "best" mechanism for comparison only. Its TLB refill cost is a theoretical minimum. In practice, higher-level page table misses impose additional costs. Nagle et al. [10] report up to twofold increase even for traditional (non-sparse) applications and operation systems.

It can be seen that with miss rates less than 0.0001, it is largely irrelevent 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%. 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, hiearchical operations, and graceful performance degradation when faced with increasing TLB miss rates. A second level TLB (like described in [1]) 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 is a single tagged cache for TLB2.

5.1.2 Size

Required performance dictates the size of TLB2, however 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 effectively ignore this limitation.

TLB2 flushing grows 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 swapout 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 immediates. 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 likelyhood 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 minimized 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 also be set by the operating system in the case when adding a TLB entry not associated with a TLB exception is required.

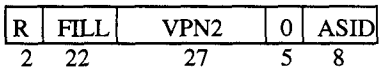


Figure 7: EntryHi Register Format

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.

The optimisition of this is to recognise 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 optimistized 32-bit entries are loaded using two "load word" instructions which sign extend the values to 64-bit once loaded for free. 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 direct mapped TLB2 is:

```
        ⋮
srl     at,k0,9
and     0xfff0
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
                ⋮
        srl     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
```

Timing for miss routine is $14 + 8n$. Complete timing for reload that misses TLB2, $36 + 8n$. 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 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, 3 level GPT refill, TLB2 best case refill, TLB2 worst case refill, and 7 level GPT refill with cache misses. It can be seen that, at worst, TLB2 refill is slightly faster than a 3 level GPT refill; at best it has significantly lower overheads.

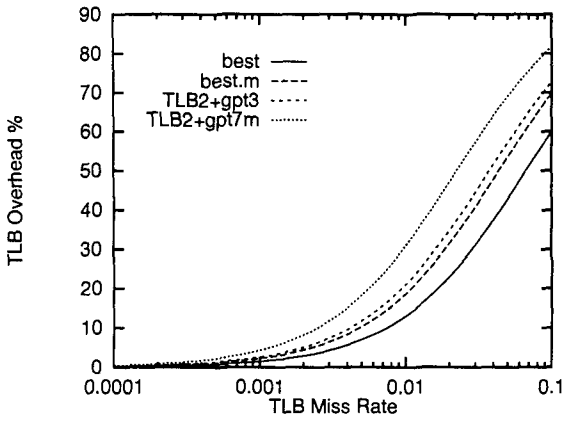


Figure 8: Direct mapped TLB2 overhead

6 Concluding Remarks

The presented software is available through the WorldWideWeb under

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

A more detailed version of this paper (including a discussion of page access right and n-way TLB2s) is available as UNSW Technical Report [9].

References

[1] K. Bala, F. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 243–254, Monterey, CA, November 1994.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] Jochen Liedtke and Kevin Elphinstone. Gpt on mips r4600. Technical Report UNSW-CSE-TR-9503, School of Computer Science and Engineering, University of New South Wales, 1995.

[10] 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.