



peterc@gelato.unsw.edu.au

© Gelato@UNSW

1

Novel Page Tables

Oct 2006

Novel Pagetables and Superpages for Itanium

Paul Davies

Ian Wienand

Adam Wiggins

Peter Chubb^a

October 2006

National ICT Australia

The University of New South Wales

^a[paul.d,ian.w,wiggins,peterc}@gelato.unsw.edu.au](mailto:{paul.d,ian.w,wiggins,peterc}@gelato.unsw.edu.au)

peterc@gelato.unsw.edu.au

© Gelato@UNSW

2

Overview

- Gelato@UNSW Virtual Memory WIP
- Focus: Removing limitations in two key areas
 1. Limited choice of Page Size
 - Transparent Superpages (Ian)
 2. Single Page Table Format
 - Abstract the Page Table Interface (Paul)
 - Guarded Page Table (Adam)

peterc@gelato.unsw.edu.au

© Gelato@UNSW

3

Novel Page Tables

Oct 2006

Large Pages

- Problem: Want to use larger pages (TLB pressure, I/O performance)
 - Itanium (etc) provides more than one page size (12 for I2)
 - How can we make them available? Desiderata:
 - * Minimise (zero?) API/ABI changes
 - * Maximise number of sizes

peterc@gelato.unsw.edu.au

© Gelato@UNSW

4

Current State

- Extensive literature review undertaken

<http://www.gelato.unsw.edu.au/~ianw/litreview>

- Issues identified, research directions being planned

peterc@gelato.unsw.edu.au

© Gelato@UNSW

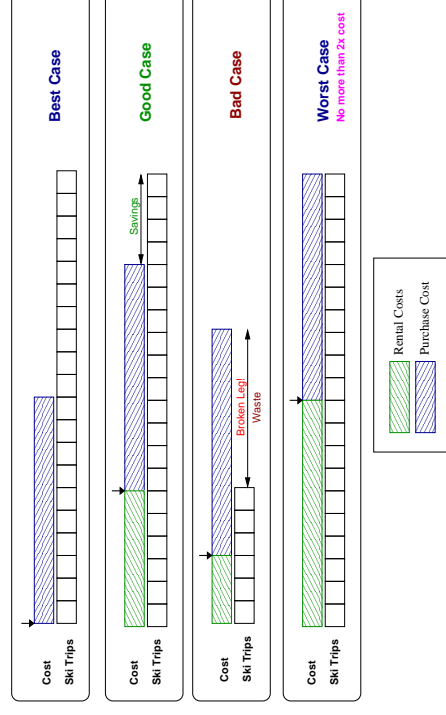
Key issues

1. Which way to grow?
 - **Promotion**: start small and get **big**
 - **Demotion**: start **big** and get small
2. **big** pages need **big** contiguous memory, small pages fragment memory.
3. Many apps want **huge** pages, many apps don't. Can we give everyone what they want?

peterc@gelato.unsw.edu.au

© Gelato@UNSW

When should we promote?



Romer (95), Fang (01) — need cheap promotion for success

peterc@gelato.unsw.edu.au

© Gelato@UNSW

peterc@gelato.unsw.edu.au

© Gelato@UNSW

peterc@gelato.unsw.edu.au

© Gelato@UNSW

The ski rental problem is an easy-to-explain analogue of some of the issues with promotion.

If you go to a ski resort to learn to ski, then you have a choice to buy or hire your skis. Say buying the skis costs as much as ten days hire. Then if you buy the skis at the start, then ski for twenty days you're ahead. If you hedge your bets (maybe you won't like skiing?) and hire for a few days, then buy, you're still ahead compared with hiring for the full twenty days. If on the other hand you buy, then break a leg, then you've wasted your money.

It can be shown that you have to buy before you've spent the cost of the skis in hire charges to get the least worse case.

This is about the same as comparing the cost of promotion with the costs of continuing with small pages. Unless the cost of promotion is really small, it's not worth doing, because you don't know how far ahead the superpage will be used.

Key point with Romer's work was that he had a very high overhead for promotion; this may not be the case on all architectures. But even so, Fang et al. went and re-visited Romer's work, and still found that it was extremely difficult to regain the overheads from complex promotion decisions. In fact, one particularly pertinent point they raise is that a very low IPC (instructions per cycle) TLB miss handler can have a very large effect on overall performance of a high IPC application; taking the trap essentially flushes all the pipelines, and then you start feeding in a very sequential instruction stream from the TLB miss handler. The key point is that we can't be doing too much at TLB miss time or even with superpages we won't reclaim the overheads.

Briefly, the problem here is

- fragmentation removes contiguous memory, so we use a buddy allocator to reduce fragmentation.
- buddy allocators pack everything together, leaving no room to grow.
- if we want to promote, we obviously need room to grow. a logical idea is to "reserve" some space for us to grow into.
- but how much do we reserve?
- and eventually, we're going to have filled up memory with large reservations, how do we get them back for the common case of

Where do we promote to?

- Large pages need contiguous memory.
- Buddy system reduces fragmentation

smaller allocations? the second figure shows this, and is based on the Rice work – thick boxes are reservations, but the system has a request for 16KiB of memory. Where do we get it? We need to keep track of all the reservations, and how much free space each one has. We can then make a decision about what to break up.

- reservations start to interfere with the page cache, which expects to be able to grab pages when they are not in use. but if that page is reserved ...

Managing wide range of page sizes

- The more page sizes, the more complexity in managing reservations.
- Algorithms must scale with number of page sizes
- Limit page size choices – **removing choice can hit some apps badly**
- Non-largest sweet spots

peterc@gelato.unsw.edu.au

© Gelato@UNSW

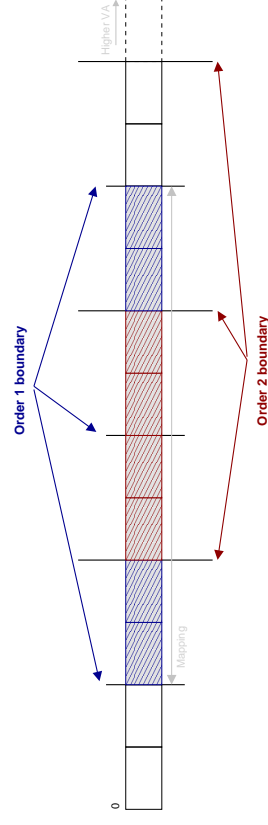
9

The point of the first figure is not to be explained in depth, but to make the point that more page sizes means a deeper tree and more complexity. To be fair to Navarro, the point should be made that he came up with an alternative scheme, but it relies on a VM splay-tree of ordered pages which Linux by default doesn't provide.

The solutions are not straight forward. You can simply artificially limit the number of page sizes an application can choose from, but because some apps take big hits without the largest page size this might be a bad idea. Some apps have a sweet spot that is not the largest page size so we don't want to take that away from them. Navarro did come up with some conclusions about page sizes, but we can agree more is better.

Conclusions

- Concentrate on demotion rather than promotion.
- Allocate the largest possible superpages for a mapping (Shimizu)



- Demote when required (mem pressure, protection)

peterc@gelato.unsw.edu.au

© Gelato@UNSW

10

Research targets

- Investigate trade-offs with hardware walkers
 - Short-format can not insert arbitrary large pages
 - Long-format has cache implications
 - Investigate possibility of SF with only software loaded large pages.
- PTE replication – can we shortcut through page table?

peterc@gelato.unsw.edu.au

© Gelato@UNSW

11

This is my current thinking for research targets. I think these questions are currently un-answered by the literature.

- of course we need something to experiment with
- both the long and the short format have different implications for large page support. The short format can't load an arbitrary superpage, but if we were to keep all superpage entries as invalid in the page table we end up with a situation of essentially a software loaded TLB for superpages. I think it would be interesting to see just how bad this actually is; especially since hardware walkers for multiple page sizes aren't really around (the long format VHPT has space for page size in the PTE, but the hash function is based on a fixed page size).

Page Table Interface

- Open-coded accesses to page tables
 - **Advantages**
 - * Simple
 - * Fast for chosen implementation
 - * Flexible within the implementation
 - **Disadvantages**
 - * Replicated code
 - * Ties OS to an implementation
 - * Ties OS to a primary architecture
 - * Conceptually ugly

peterc@gelato.unsw.edu.au

© Gelato@UNSW

12

- keeping parts of the page table invalid to the hardware walker but valid to the operating system also opens up some other opportunities, such as “short-cuts” where we check the value of the PMD which may be marked as a superpage PMD, etc. Previous experimentation has tried this (Winwood) but we are of the opinion it creates a lot of overheads. This is overlap area with the GPT, as it wants to complicate the page table walker too. Ken Chen has also maybe looked at this, but afaik there is nothing published on it.

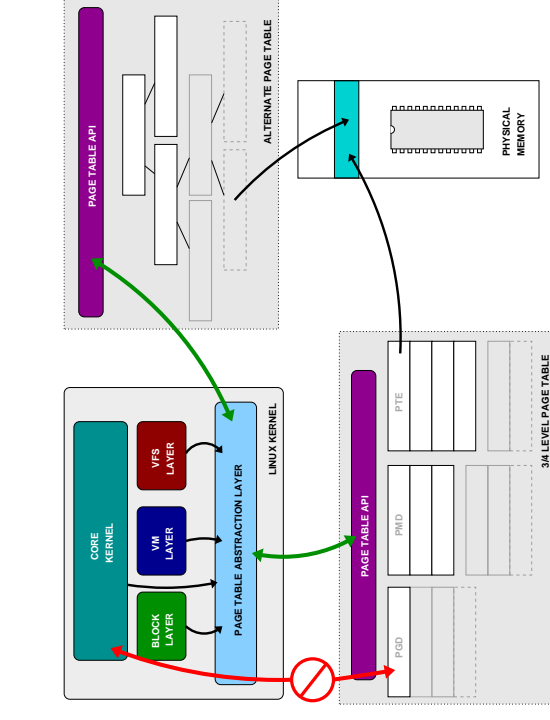
- The Linux kernel accesses the page table implementation directly. This is a very practical engineering approach that has served Linux very well for smaller address spaces (up to 32 bits). Clever use of macros (see `pgtable.h`) has seen the implementation adapt to different architectures seamlessly. Keeping it simple has reduced the likelihood of bugs and races. Direct access has eliminated the cost of accessing the page tables through an interface. It has also proved to be enormously flexible within the implementation, with each iteration being tailored for each individual need.
- With the move into a sparsely occupied 64 bit address space, Linux may have outgrown the current implementation. Unfortunately the approach described above has served to tie the OS

to the MLPT and has served to entrench the x86 since the page table is hardware walked.

- Changing the page table implementation can be achieved by accessing the page table through a well defined and simple interface. A clean interface would create, destroy, build, lookup and iterate.

The Linux Page Table Interface

- Linux kernel code **assumes** a 2,3,4 level hierarchical page table (MLPT)
- It could be **abstracted** from the PT implementation
- This work: Modify kernel to be accessed by **defined interface ONLY**
- Page table implementations implement the page table API
- Page tables could be swappable on boot, or maybe in the future, per process depending on workload



peterc@gelato.unsw.edu.au

© Gelato@UNSW

14

Oct 2006

Novel Page Tables

The Linux Page Table Interface

- Linux kernel code **assumes** a 2,3,4 level hierarchical page table (MLPT)
- It could be **abstracted** from the PT implementation
- This work: Modify kernel to be accessed by **defined interface ONLY**
- Page table implementations implement the page table API
- Page tables could be swappable on boot, or maybe in the future, per process depending on workload

• Linux assumes a multi level hierarchical page table (MLPT). To change the page table implementation we firstly abstracted the page table implementation from the core kernel code. Secondly, we defined a conceptually clean interface and modified the kernel to access the page tables only through this interface.

• The kernel should access the page table through this interface (think coding by contract). Page table implementors then implement the page table API e.g.,: GPT or MLPT. The immediate goal is to chose a page table implementation at boot time, but a longer term goal would be for processes to have different page tables depending on workload.

Our Page Table Interface

- PTI has arch independent and dependent components
- Most arches will not change page table. For these, the PTI is cost only.
- For arches that change implementation (IA64), PTI cost offset by:
 - Seamless Superpages **maybe**
 - Page Table Sharing **perhaps**
 - Virtualisation opportunities **in the future**

Progress Achieved

- Abstracted the page table for 2.6.17.1
- Measured the cost of the PTI
 - Benchmarked with and without PTI.
 - PTI costs around 4% on fork, gives 14% speedup on page fault; /bin/sh speeds up slightly.
- Experimenting with alternate pagetable implementations
 - Guarded Page Table (GPT), running rough in 2.6.17.1

- We have abstracted the page table for 2.6.17.2 and the measured the cost of the PTI. The cost of the PTI is summarised below:

1. Fork
 2. Exec
 3. Minor page fault
- We have a page table running rough (untuned) under the PTI for 2.6.17.2.

Problems and Conclusion

- PTI **hard** to maintain internally
 - Touches many functions across many files
 - Ideally requires upstream acceptance
 - More complex: new bugs and evil races??
- **Excessive** tailoring of iterations hampering simplicity
- **Lack** of community **interest**
 - PTI may be used to **enhance** current implementation for upstream acceptance

- The PTI has proved a heavy burden to maintain internally for several reasons

1. The page table interface touches many functions across many files which are constantly evolving.
2. The kernel is increasingly being tied to the MLPT with more and more iterations being tailored to individual needs as the kernel evolves. At the very least, this is killing the prospect of a simple elegant interface. (They are exercising the flexibility within the implementation alluded to earlier).

- Most developers prefer the simple direct approach as workloads that will benefit from a sparsely occupied 64 bit address space

are still in the future. When we are able to demonstrate a new implementation benefiting important workloads we hope to gain community interest. We can also try to use the PTI's flexibility to study page tables and find ways to enhance the current page table implementation.

A Guarded Page Table (GPT) for Linux

- First attempt at using PTI for an alternative PT - Successful.
- GPT designed to support **large** & **sparse** address spaces.
- Combining path-/level-compression:
 - Aims to reduce **number of PT nodes** & **lookup cost**.
 - Extra guard & level size fields → **Larger nodes?**
- Limited applicability, only if Linux PT **not** hardware walked.
 - Requires long format VHPT for IA64 - **Patch in limbo**.

peterc@gelato.unsw.edu.au

© Gelato@UNSW

18

What is a GPT? It's a generalised multi-level page table that supports both path-compression (think Patricia tree) through the use of guards fields in nodes, and level compression through the use of a size field for internal levels.

Path-compression collapses unary paths in the tree into a single pointer, while level-compression collapses full sub-trees into a single level.

The potential advantage of the GPT is reducing the number of nodes in the tree. The tradeoff, though, is that the extra guard and level size fields potentially increase the size of the node which can negate the gains made through the reduction of the number of nodes.

Besides having tried out a different page table, the GPT has the potential to save memory and reduce lookup cost. However, it is only applicable to architectures where the Linux page table is NOT walked by hardware (IA64, PowerPC, MIPS, etc). For IA64 the LVHPT patchset

is required which is currently not finding uptake by the community.

The Linux GPT was based on a GPT prototype being independently developed with support for superpages, full sized guards at both internal levels and leaves, and support for mixed level sizes and a framework for supporting level-compression in the future. Using the PTI proved fairly simple (once the interface became more settled), main issues have been with adapting the existing GPT code to the PTI and completing the GPT code. The only issue with PTI is that when to lock is generally unclear particularly in the iterators. This needs to be documented in the PTI.

Although the GPT supports superpages, they are unused as Linux does not currently use superpages. Similarly, for now, fixed page-size levels are used so as to provide a baseline comparison with the standard Linux page table by keeping the memory allocation the same as the standard Linux page table and removing the overheads of level-compression from

Experience

- Code based on an independently developed GPT prototype.
 - Supports superpages & variable level sizes - **Unused!**
- Lessons learnt:
 1. Node size must be reduced to a **single word!**
 - **Remove** level size field - **Unused** (currently).
 - **Remove** guards field from leaves - **Not required.**
 - **Restrict** guards fields size - **Hard to do!**
 2. **Simplify** code - **Superpages** support complex & unused.

the comparison. Level-compression will aim to increase the level size anyway.

The prototype uses two 64-bit words to encode nodes (cf a single word used by Linux's standard page table nodes). This introduces a number of problems. Firstly, it increases the memory requirements and cache footprint of the GPT, which potentially removes any memory gains achieved via guarded levels being skipped. Secondly, it introduces concurrency issues as multiword loads and stores are typically not atomic. As a result node reads and writes must incur the overhead of making them appear atomic, particularly as page table lookup does not use any page table locking mechanisms.

An issue not mentioned in the slides is that we currently don't have the VHPT enabled. This mainly effects page-fault cost and has no effect on the iterators, hence minimal impact on `fork()/exec()` costs.

When using 16KB pages, each level contains 2^{10} entries mapping 10-bits. Thus 5 levels cover 50 bits, the page offset supplying the remaining 14-bits. Compare this with Linux's standard page table on IA64 which requires 64KB pages and the full 4 levels to map the same. Unlike the standard page table, the GPT adapts during runtime to address space usage — it doesn't require compile time configuration of table depth. This should attract distribution providers who like to go with the single kernel image method of distribution.

Experience playing with the GPT has been that during startup, applications typically lookup paths only 2–3 levels deep — shorter than the 3 or 4 levels of the standard page table on IA64.

Leaves are never guarded. This is expected as the level sizes are large (1024 entries) and we would expect dense clumps of PTEs.

The main observation has been the cost of the actions which use iter-

ators heavily such as `fork()`, `exec()` and `mmap()`. The performance of these as seen in `LMbench` has been up to twice as slow as standard Linux. These costs almost undoubtedly are dominated by the increase in node size for the GPT blowing out the cache footprint. Regardless of anything else, the PTE nodes of which 'n' will be visited during iteration are twice the size.

Additional problems will result from the currently complexity of the GPT code due to support for superpages (through replication) and the variable level sizes. The iterators themselves may also have design overheads, for example, the copy and move iterators still re-traverse the tree for the target PTE range though this iterator effects are limited to `fork()`. So what have we learnt from these observations? Firstly, it is critical to reduce the node size back down to a single word. We don't use variable sized levels currently so ditch the field. Guards on leaves (PTEs) are

used to ditch them. The last step that remains is to reduce the guard field's size, but this is hard. We don't have many bits to spare and the guard requires a size of up to 50 bits for the value and then another field to say how long it is! This remains the main challenge to reducing the node size. The second lesson is to embrace KISS (keep it simple stupid) and remove support for unused features like superpages, level-compression, etc., as these add a lot of complexity increasing the chance of bugs and potentially impacting performance. As a side note reducing the node size to a single word simplifies the code for accessing nodes as node loads/stores are then atomic.

Future direction

- Develop tools to better understand applications PT structure.
- Cherry pick GPT ideas & build **hybrids** with standard PT.
 1. Add guards to **PUD/PMD** in standard PT, **no PTI**.
 2. GPT with **PTE arrays** as leaves - **Requires PTI**.
- Incremental road map from standard PT to full GPT.
- More digestible for community uptake.



The first thing we are looking into is developing better tools to understand the structure of both Linux's standard page table and the GPT for various applications of interest. The aim of this is to ascertain where exactly gains can be made through the use of GPT mechanisms.

The next step is to cherry pick ideas from our experience with the GPT under Linux and develop a road map from the standard Linux page table to potentially a general GPT implementation in the future. The motivations behind this are that it provides a more digestible progression to facilitate uptake into the mainstream kernel, and that it allows a number of features to be preserved during development.

The focus of the hybrid approach is to keep the page table node size down to a single word and benefits can be achieved.

Simply adding guards to the PUD/PMD levels of the standard page table is the least intrusive approach, as the PTI isn't required. However, it also has the least potential gains, allowing only the middle two levels of the page table to be skipped.

A GPT with PTE arrays as leaves provides more room for potential gains and for experimentation. Using PTE arrays as leaves simplifies the GPT code, while at the same time providing the benefits mentioned below. It is however more intrusive, requiring the PTI.

Benefits of both hybrid stages are they preserve the following features:

- Maintaining the fine-grained locking of PTE arrays (GPT uses

Future direction - Cont.

- Hybrid focus on keeping PT nodes size a **single word!**
- Advantages of both hybrid approaches over full GPT:
 - **Fine grained locking on PTE arrays** can be used.
 - Use the **short format VHPT** on IA64.
 - Can be combined with **shared page table patch set**.
- If hybrids **beneficial**, level-compression can be **revisited**.

global lock per PT) → Better scalability.

- Both using IA64's short format VHPT, rather than being dependent on yet another set of patches → Simpler update.
- Both can be combined with the shared page table patch set from Dave McCracken → Further reduction of PT memory.

If both of these stages prove to be beneficial the next step would be to revisit level-compression once more.

Summary

Gelato@UNSW's Virtual Memory research focuses on

- Flexibility,
- Scalability
- and Performance

