Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

PATting Linux

Venkatesh Pallipadi
Suresh Siddha
Intel Open Source Technology Center
{venkatesh.pallipadi|suresh.b.siddha}@intel.com

Abstract

The Page Attribute Table *(PAT)*, introduced in Pentium III, provides the x86 architecture with an option to assign memory types to physical memory, based on page table mappings. The Linux kernel, however, does not fully support this feature, though there were many earlier efforts to add support for this feature over the years.

The need for PAT in Linux is becoming critical with the latest platforms supporting huge memory; these cannot be supported by limited number of MTRRs, often leading to poor graphics and IO performance.

In this paper, we will provide insight into earlier attempts to enable PAT in the Linux kernel and details of our latest attempt, highlighting various issues that were encountered. We will describe the new APIs for userspace/drivers for specifying various memory attributes.

1 Introduction

Page Attribute Table *(PAT)* has been a hardware feature in x86 processors starting from the Pentium III generation of processors. The PAT extends the IA-32 architecture's page-table format to allow memory types to be assigned to regions of physical memory based on linear address mappings [1].

PAT is a complementary feature to Memory Type Range Registers *(MTRR)*. The MTRRs are used to map regions in physical address space with specific memory types.

PAT and MTRR together allow the processor to optimize operations for different types of memory such as RAM, ROM, frame-buffer memory, and memory-mapped I/O devices. The MTRRs are useful for statically describing memory types for physical ranges, and are typically set up by system firmware. The PAT allows dynamic assignment of memory types to pages in linear address space.

The Linux kernel (as in linux-2.6.25 [2]) does not fully support PAT. It only uses PAT for write-back and uncached mappings, and uses MTRR to dynamically assign write-combining memory type. This results in over-dependence on MTRR, causing issues like performance problems with the display driver (X), for example.

The Linux kernel also does not enforce the *no-aliasing* requirement while mapping memory-types, potentially resulting in various linear addresses from the same or different processes, mapping to the same physical address with different effective memory types. This aliasing can potentially cause inconsistent or undefined operations that can result in system failure [1].

In this paper, we present details about our recent effort towards adding PAT support for the x86 architecture in the Linux kernel [13]. We start with an architecture primer on PAT and MTRR in Section 2, followed by a description of the Linux kernel status (as in linux-2.6.25) in Section 3. Section 4 provides details about our proposed PAT implementation. APIs for drivers and applications to set memory types is discussed in Section 5. We conclude the paper with a look at the future in Section 6.

2 Architectural Background

In this section, we provide an architectural overview of PAT, MTRR, and interactions between PAT and MTRR. Refer to the chapter on `Memory Cache Control` in [1] for a complete reference.

2.1 PAT

The PAT extends the IA-32 architecture's page-table format to allow memory type to be assigned to pages based

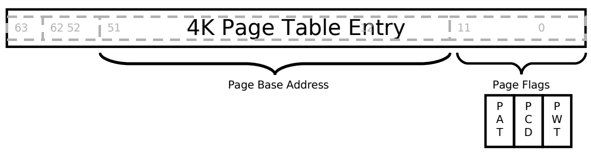

Figure 1: PAT flags in a 4K Page Table Entry

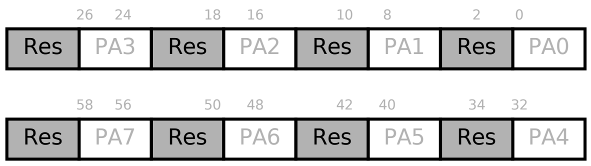
26 24 18 16 10 8 2 0
Res PA3 Res PA2 Res PA1 Res PA0
58 56 50 48 42 40 34 32
Res PA7 Res PA6 Res PA5 Res PA4

Figure 2: IA32_CR_PAT MSR

on linear address mappings. Figure 1 shows a regular 4K page table entry, with bits 12–51 forming the physical frame number. Bits 0–11 have various page flags associated with this mapping. Page flags PAT, PCD, and PWT together represent the PAT attribute of the page. These three bits index into 8-page attribute types in the `IA32_CR_PAT` MSR. This MSR's content is depicted in Figure 2, with each Page Attribute (PA0 through PA7) mapping to a particular memory type encoding.

The memory types that can be encoded by PAT are listed in Figure 3. PA0 through PA7 in the `IA32_CR_PAT` MSR can contain any encoding from Figure 3.

The `IA32_CR_PAT` MSR is set with a predefined default setting for each PAT entry (PA0 through PA7) upon power up or reset. System software can write different encodings to these entries with the `WRMSR` instruction. On a multi-processor system, the `IA32_CR_PAT` MSR of all processors must contain the same value.

The PAT allows any memory type to be specified in the page tables, and therefore it is possible to have a single physical region mapped to two or more different linear addresses, each with different memory types. These mappings may be the part of the same address space, or may be in the address spaces of different processes. Such a mapping of a single physical region with multiple linear address ranges with different memory types is referred to as *aliasing*. Architecturally, any such aliasing can lead to undefined operations that can result in a system failure. It is the operating system's responsibility to prevent such aliasing when PAT is being used.

Encoding	Memory Type
0	Uncacheable (UC)
1	Write Combining (WC)
2	Reserved
3	Reserved
4	Write Through (WT)
5	Write Protected (WP)
6	Write Back (WB)
7	Uncached (UC-)

Figure 3: Memory types that can be encoded with PAT

2.2 MTRR

The MTRR provides a mechanism for associating the memory types with physical address ranges in system memory. MTRR capability can be determined by the `IA32_MTRRCAP` MSR. The encodings supported by MTRR are listed in Figure 4. The MTRRs are defined as a combination of:

- **Fixed Range MTRRs:** A Fixed Range MTRR consists of predetermined regions of size 64K, 16K, and 4k in the 0–1MB physical memory range. This includes eight 64K ranges, sixteen 16K ranges, and sixty-four 4K ranges. Each such range can be defined as a particular memory type encoding.
- **Variable Range MTRRs:** Most x86 processors support up to eight Variable Range MTRRs. They are specified using a pair of MSRs: `IA32_MTRR_PHYSBASEn` defines the base address; and `IA32_MTRR_PHYSMASKn` contains a mask used to determine the range.
- **Default MTRR Type:** Any physical memory range not covered by a fixed or variable MTRR range takes the memory type attributes from the `IA32_MTRR_DEF_TYPE` MSR.

When MTRRs are enabled, MTRR range overlaps are not defined, except for:

- Any range overlap, with one of the ranges being of uncached type, will result in the effective memory type of uncached.
- Any range overlap, with one range being write-back and another range being write-through, will result in the effective memory type of write-through.

Encoding	Memory Type
0	Uncacheable (UC)
1	Write Combining (WC)
2	Reserved
3	Reserved
4	Write Through (WT)
5	Write Protected (WP)
6	Write Back (WB)
7–0xFF	Reserved

Figure 4: Memory types that can be encoded with MTRR

While MTRRs are enabled and being used, the operating system has to make sure that there are no overlapping MTRR regions with overlapping types not defined above.

2.3 MTRR and PAT overlap

PAT and MTRR may define different memory types for the same physical address. The effective memory type resulting from this overlap is architecturally defined as in the chapter on `Memory Cache Control` in reference [1]. Specifically,

- The PAT memory type of write-combine takes precedence over any memory type assigned to that range by MTRR.
- The PAT memory type of uncached-minus gives precedence to any MTRR write-combine setting for the same physical address. If there are no MTRR memory types or if the memory type in MTRR is write-back, the effective memory type for that region will be uncached.

3 Linux Kernel Background

All references to the Linux kernel in this section refers to version 2.6.25. The Linux kernel supports PAT in a very restrictive sense, with PAT memory types uncached and uncached-minus being used by kernel APIs like `ioremap_noncache()`, `set_memory_uc()`, `pgprot_noncached()`, etc. User-level APIs that use the PAT uncached memory type are the `/proc`, and `/sys` PCI resource interfaces and `mmap` of `/dev/mem`.

Linux also supports adding new MTRR ranges using the kernel API `mtrr_add()`. There is also a user-level API to set MTRR ranges by using `/proc/mtrr` writes.

The kernel does not do any aliasing checks while setting the PAT mappings. The kernel only makes sure that kernel identity mapping of physical memory is consistent while changing the PAT memory type with some of the APIs above. The kernel does check for any overlap with existing MTRR ranges, while adding a new MTRR.

Following is an example of MTRR usage on a typical server. Figure 5 shows the contents of `/proc/mtrr`. The effective memory type on this system will be as in Figure 6.

Below we will look at the memory-attribute-related problems that we have in the Linux kernel.

3.1 Limited number of MTRRs

As explained in Section 2, typically there are only eight variable-range MTRRs supported on x86 CPUs. With an increasing amount of physical memory in the platform, the platform firmware ends up using most of those MTRRs to statically define memory types for the system memory range. When a driver later wants to use MTRR to map some range as write-combining, for example, there may not be any free MTRRs available for use. This results in the driver not being able to set a memory type. This can appear to the end user as (for instance) the video driver running less optimially, as it ends up using an uncached memory type for frame-buffer instead of the desired write-combine memory type.

3.2 MTRR conflict with BIOS MTRRs

As described in Section 2, certain overlapping MTRR memory types like write-combine and uncached result in causing the effective type to be uncached. On some platforms, the BIOS sets up PCI ranges explicitly mapped as uncached and a potential overlapping write-back for RAM. Later, when some driver which wants to map the same range as a write-combine memory type using MTRRs, there will be a conflict which results in the effective memory type being uncached. This results in the driver not being able to get optimal performance.

This has been reported a few times by end users on `lkml` [10] [3] [11] on various platforms. Unfortunately,

```
# cat /proc/mtrr
  reg00: base=0xd0000000 (3328MB), size= 256MB: uncacheable, count=1
  reg01: base=0xe0000000 (3584MB), size= 512MB: uncacheable, count=1
  reg02: base=0x00000000 (   0MB), size=8192MB: write-back, count=1
  reg03: base=0x200000000 (8192MB), size= 512MB: write-back, count=1
  reg04: base=0x220000000 (8704MB), size= 256MB: write-back, count=1
  reg05: base=0xcff80000 (3327MB), size= 512KB: uncacheable, count=1
```

Figure 5: Variable MTRR example

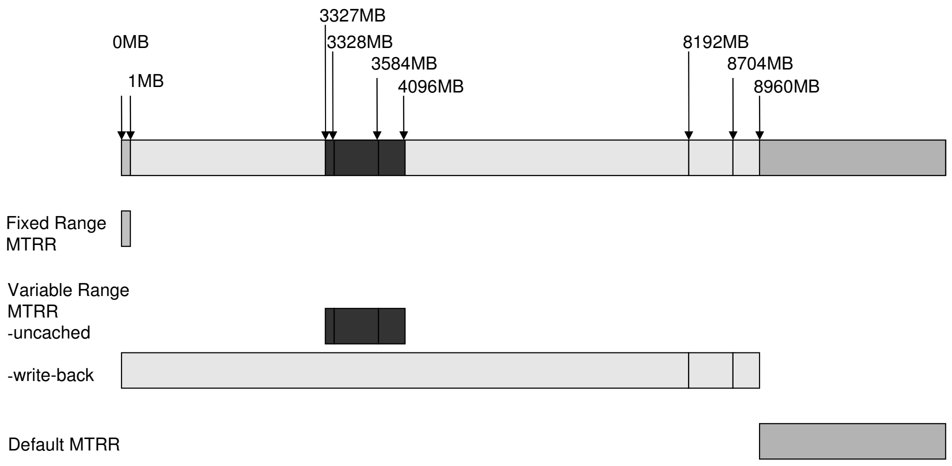

Figure 6: Effective memory types due to MTRR set by BIOS

there is no way to resolve this with existing PAT and MTRR support in the 2.6.25 kernel.

3.3 No well-defined APIs

Poorly defined APIs across PAT and MTRRs have led to various issues like:

- Drivers making assumptions about the underlying kernel implementation. For instance, the frame buffer driver is assuming that `ioremap()` will use PAT uncached-minus mapping and the driver follows the `ioremap()` call by an `mtrr_add()` call to set a write-combine memory type to the same range. Changing `ioremap()` from uncached-minus to uncached resulted in poor frame buffer performance in this specific case.

- Drivers setting the page table entries along with the PAT memory types natively, either by using `pgprot_noncached()` or by directly using `PAT`, `PCD`, and `PWT` bits. This results in flaky code, where the driver depends on kernel PAT usage.

4 Current PAT effort

4.1 Earlier PAT attempts

As was emphasised earlier, PAT as a hardware feature has been around for few years. During those years, there were quite a few attempts to enable PAT support in the Linux kernel.

One of the initial attempts was from Jeff Hartmann [8] in January, 2001. The patch proposed a `vmalloc()` kind of interface to support per 4K-page level write-combining memory type control. There were no responses on the mailing list archive, and so we conclude that the patch did not make its way into the Linux kernel for some unknown reason.

Terence Ripperda proposed PAT support with [9] in May, 2003, which supported adding write-combining mapping for AGP and framebuffers. There were concerns expressed about this patch in the mailing list, mainly related to keeping the memory type consistent for a physical address across different virtual mappings that may exist in the system.

```
mtrr: type mismatch for e0000000,8000000 old: write-back new: write-combining
mtrr: type mismatch for e0000000,4000000 old: write-back new: write-combining
```

Figure 7: common MTRR error message in kernel log

Terence followed it up with [4] in April, 2004, adding memory type tracking. That patch never made into the Linux kernel either, due to some concerns about various CPU errata on the mailing list.

Eric W. Biederman proposed PAT support with [6] in August, 2005, which started a fresh discussion on the mailing list about aliasing and PAT-related processor errata. Andi Kleen took up this patch and included it in his test tree [12]. However, the PAT support never got wide enough testing and did not get into the upstream Linux kernel. Also, none of the patches fully addressed attribute aliasing concerns.

4.2 Current PAT Proposal

Our initial proposals [7] [5] were based on Eric and Andi's patchset, with changes around identity mapping of reserved regions or holes and a few other cleanups and bug fixes. Those patches broke a lot of systems and provided us with a lot of feedback about what was not being done correctly.

Based on the feedback and breakage reports, we changed our patches to eliminate the issues around not having identity mapping for reserved regions and eliminated the changes for `early_ioremap`, simplifying our approach along the way. We also got benefits from other changes like `x86 change_page_attr()`.

The patches here [13] are version 3 of the patchset which was included in `linux-2.6.25-rc8-mm2`. The patchset also took a slightly different top-down approach, defining the PAT-related APIs and the eventual PAT bit setting for those APIs in different use cases, trying to ensure backward compatibility with the older versions of drivers and applications. All of the APIs related to PAT and memory type changes are described in detail in Section 5.

4.3 Preventing PAT aliasing

One of the big roadblocks for earlier PAT patches was aliasing related to PAT attributes. As per the processor specification [1], single physical address mapped to two or more different linear addresses should not have different memory types. Such aliasing can lead to undefined operations that can result in system failure.

The current PAT proposal handles this by using two internal functions, `reserve_memtype()` and `free_memtype()`. Any API that wants to change the memory type for a region first has to go through `reserve_memtype` to be sure that there are no aliases to the physical address, reserving the memory type for the region in the process. At the time of unmapping the memory type, API will free the range with a `free_memtype()` call. APIs will fail if `reserve` fails due to existing aliases.

Internally, the `reserve` function goes through a linked list which keeps track of physical address ranges with a specific memory type. The linked list is maintained in sorted order, based on the start address. The linked list may contain ranges with different sizes, and will detect partial or full overlaps with a single existing mapping or overlaps with multiple regions, with conflicting memory types.

If there is more than one user for a specific range with same memory type, the reference counting for such users are tracked by having multiple entries in the linked list.

To keep the implementation simple, this list is implemented as a simple doubly linked list. In the future, if there are any bottlenecks around this list, it can be optimized to have some cache pointers to previously reserved or freed regions, and changing the list into a more efficient data structure.

5 PAT APIs

One of the major challenges with PAT was to add the support in a clean manner, causing as few issues with existing applications and drivers as possible. As described in Section 3, the current memory type usage in Linux has some API-level confusion. That highlighted

API	RAM	ACPI, ...	Rsvd/ Holes
ioremap()	–	UC-	UC-
ioremap_cache()	–	WB	WB
ioremap_nocache()	–	UC	UC
ioremap_wc()	–	WC	WC
set_memory_uc() set_memory_wb()	UC	–	–
set_memory_wc() set_memory_wb()	WC	–	–
pci /sys resource file	–	–	UC-
pci /sys resource_wc file	–	–	WC
pci /proc/ device file	–	–	UC-
pci /proc/ device file ioctl PCIIOC_ WRITE_COMBINE	–	–	WC
/dev/mem read-write	WB	UC	UC
/dev/mem mmap with O_SYNC	–	UC	UC
/dev/mem mmap no O_SYNC	–	alias	alias
/dev/mem mmap no O_SYNC with no alias MTRR type WB	–	WB	WB
/dev/mem mmap no O_SYNC with no alias MTRR type not WB	–	–	UC-

Table 1: PAT related API cheat-sheet

the need to establish a clear API for everything related to memory type changes.

The API defined with the proposed PAT patches is described in detail below.

5.1 ioremap

`ioremap()`, `ioremap_cache()`, `ioremap_nocache()`, and `ioremap_wc()` are the interfaces that a driver can use to mark a physical address range with some memory type. The expected usage of these interfaces is over a physical address range that is either reserved/hole or a region used by ACPI, etc. `ioremap` and friends should never be used on a RAM region that is being used by the kernel.

`ioremap` interfaces, when they change the memory type, keep the memory type consistent across the virtual address where the address is being remapped into, and the kernel identity mappings.

`ioremap` interfaces may fail if there is an existing stricter memory type mapping. *Example:* If there is an existing write-back mapping to a physical range, any request for uncached and write-combine mappings will fail.

`ioremap` interfaces will succeed if there is an existing, more lenient mapping. *Example:* If there is an existing uncached mapping to a physical range, any request for write-back or write-combine mapping will succeed, but will eventually map the memory as uncached.

5.2 set_memory

`set_memory_uc`, `set_memory_wc`, and `set_memory_wb` are used to change the memory type of a RAM region. A driver can allocate a memory buffer and then use `set_memory` APIs to change the memory type for that region. It is the driver's responsibility to change the memory type back to write-back before freeing the page. A failure to do that can have nasty performance side effects as the page gets allocated for different usages later.

As with the ioremap interfaces, the kernel makes sure that the identity map aliases, if any, are kept consistent.

`set_memory` APIs can fail if there is a different memory type that is already in use for the same physical memory region.

5.3 `/proc` access to PCI resources

User-level drivers/applications can access a PCI resource region through the `/proc` interface. The resource file at `/proc/bus/pci/<dev>/` is `mmap`-able and an application can use that `mmap`ped address to access the resource.

By default, such an `mmap` will provide uncached access to the region. Applications can use `PCIIOC_WRITE_COMBINE` ioctl and get write-combine access to the resource, in cases where the region is *prefetchable.*

A request to uncached access can fail if there is already an existing write-combine mapping for that region. A request for write-combine access can succeed with uncached mapping instead, in the case of already existing uncached mapping for this region.

5.4 `/sys` access to PCI resource

Apart from the `/proc` interface described above, there is also a `/sys`-based interface that can be used to access PCI resources. It resides under `devices/pci<bus> /<dev>/`. This is again an `mmap`-able interface. The file `resource` is useful to get a UC access, and file `resource_wc`, to get write-combing access (in case the region is *prefetchable*).

The success and failure conditions of a `mmap` of the `/sys` PCI resource file are the same as in the `/proc` resource file description above.

5.5 read and write of `/dev/mem`

The existing API that allows read and write of memory through `/dev/mem` will internally use `ioremap()` and hence read and write using the uncached memory type. To read/write RAM, we use the identity mapped address, with existing WB mapping.

5.6 `mmap` of `/dev/mem`

`/dev/mem` `mmap` is an interface for applications to access any non-RAM regions. Applications have been using a `mmap` of `/dev/mem` for multiple uses. Changing `mmap` of `/dev/mem` behavior to go along well with PAT changes was one of the major challenges we had.

For example, X drivers use `mmap` to map a PCI resource first, followed by adding a new MTRR to make that physical address either uncached or write-combine. This will work, as the current `mmap` will just use write-back mapping in PAT, and MTRR uncached or write-combine takes higher precedence and changes the effective memory type for this region.

We will look at all the different usage scenarios of `/dev/mmap` below:

- *mmap with O_SYNC:* Applications can open `/dev/mem` with the `O_SYNC` flag and then do `mmap` on it. With that, applications will be accessing that address with an uncached memory type. `mmap` will succeed only if there is no other conflicting mappings to the same region.

- *mmap without O_SYNC and existing mapping for the same region:* Applications that do not use O_SYNC, when there is an existing mapping for the same region, will inherit the memory type from the existing mapping. This will be the case with applications mapping memory with a driver already having used `ioremap` to set the memory as write-back, write-combining, or uncached.

- *mmap without O_SYNC, no existing mapping, and write-back region:* The *ACPI data* region and a few other such regions are non-RAM, but can be accessed as write-back. There are applications like `acpidump` that `mmap` such regions. There is also a legacy BIOS region that is write-back that applications like `dmidecode` want to mmap. To be friendly to such usages, on an `mmap` of `/dev/mem` without `O_SYNC` and with no existing mappings, we look at the MTRR to figure out the actual type. If the MTRR says that region is write-back, then we use write-back mapping for the `/dev/mem` `mmap` as well.

- *mmap without O_SYNC, no existing mapping, and not a write-back region:* For an `mmap` that comes under this category, we use uncached-minus type mapping. In the absence of any MTRR for this region, the effective type will be uncached. But in cases where there is an MTRR, making this region write-combine, then the effective type will be write-combine. This behavior was added for a very special case, to handle existing X drivers without breakage. The X model of `mmap`ing graphics memory and then adding write-combining MTRR to it will work with this mapping.

Table 1 is a API cheat-sheet for application/driver interfaces to make any memory type changes.

6 Future Work

Given the success rate of earlier PAT patches, our first goal here is to ensure that the basic PAT patches make it to the upstream kernel with minimal disruptions, and also to provide APIs to applications and drivers that will remove long-standing MTRR limitations. As a result, our initial patchset does not address all the problems associated with PAT. Specifically, we have the following items in our immediate to-do list.

- Provide an API for drivers using `pgprot_noncached()` or handling the page table flags by hand to manipulate `PAT`, `PCD`, and `PWT` bits, so that they can do so ensuring no aliases. Currently, such drivers (mostly framebuffer and video drivers) do various things like mapping the pages as write-back in the page table and then calling MTRR to mark it write-combine, directly map the pages as uncached, etc.

- Ensure that PAT is not breaking any architectural guidelines while changing memory type attributes. Specifically, [1] mentions a sequence of steps that needs to be followed while changing the memory type attribute of a page from cacheable to write-combining. We need to make sure that the Linux kernel is not breaking any such rules.

- The future of the `/proc/mtrr` and `mtrr_add()` interfaces needs to be determined. The options are: deprecate those APIs and encourage the drivers and applications to switch to new PAT APIs, or emulate those APIs using PAT internally. This can be better addressed as current PAT patches makes into upstream and we get more feedback from the users about the PAT APIs and their usability.

7 Acknowledgements

Thanks to the developers and testers in the community who took time to comment on, report issues with, and contribute to PAT discussion on the mailing list. Special thanks to Asit Mallick and Arjan van de Ven for their continued support during our work on PAT pacthes; Ingo Molnar, Thomas Glexiner, H Peter Anvin for reviewing and fixing problems with PAT patches and also pushing the PAT patches through into x86 testing tree; Andi Kleen, Eric Biederman for their PAT patches which formed the basis for our first implementation; and everyone else who worked on PAT feature before or affected by PAT/MTRR issues on their platform and took time to report it on lkml mailing list.

References

[1] `Intel`$^{®}$ `64` and `IA-32` architectures software developer's manuals: `Volume 3A`. `http://developer.intel.com/products/processor/manuals`.

[2] Linux 2.6.25. `http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.25.tar.bz2`.

[3] Mailing list archive. `MTRR` initialization. `http://lkml.org/lkml/2007/9/14/185`.

[4] Mailing list archive. `PAT` support. `http://www.ussg.iu.edu/hypermail/linux/kernel/0404.1/0686.html`.

[5] Mailing list archive. `[patch 00/11] PAT` x86: `PAT` support for x86. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0801.1/1428.html`.

[6] Mailing list archive. `[PATCH]` i386, x86_64 initial `PAT` implementation. `http://www.ussg.iu.edu/hypermail/linux/kernel/0508.3/1321.html`.

[7] Mailing list archive. `[RFC PATCH 00/12] PAT` 64b: `PAT` support for x86_64. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0712.1/2268.html`.

[8] Mailing list archive. `[RFC] [PATCH] PAT` implementation. `http://www.ussg.iu.edu/hypermail/linux/kernel/0101.3/0630.html`.

[9] Mailing list archive. pat support in the kernel. `http://www.ussg.iu.edu/hypermail/linux/kernel/0305.2/0896.html`.

[10] Mailing list archive. type mismatch for e0000000,8000000 old: write-back new: write-combining on kernel 2.6.12. `http://lkml.org/lkml/2005/6/18/52`.

[11] Mailing list archive. type mismatch for f0000000, 1000000 old: write-back new: write-combining. `http://lists.us.dell.com/pipermail/linux-poweredge/2006-March/025043.html`.

[12] Mailing list archive. What will be in the x86-64/x86 2.6.21 merge. `http://www.ussg.iu.edu/hypermail/linux/kernel/0702.1/0832.html`.

[13] Mailing list archive. x86: `PAT` support updated - v3. `http://lwn.net/Articles/274175/`.