Verification of OS-level Cache Management

Renato Mancuso

Sagar Chaki

OSPERT 2018
Goal + Approach

Colored Lockdown for deterministic cache management via CBMC

source code

C

Linux Kernel Module
Last Level Cache

Management Model

✔ Addresses all the sources of interference
✔ Converts the LLC cache in a deterministic object at the granularity of a single memory page
✔ Allows the use of legacy code
✔ Provides flexibility in cache assignment

Profile ➔ Remap ➔ Allocate

Background
Colored Lockdown
**Last Level Cache**

Management Model

- Addresses all the sources of interference
- Converts the LLC cache into a deterministic object at the granularity of a single memory page
- Allows the use of legacy code
- Provides flexibility in cache assignment

Profile ➔ Remap ➔ Allocate

100% hits on allocated pages
100% misses on non-allocated pages
Caches are critical, constrained resources. Optimal allocation?

Location of hot region(s) is unknown

Absolute virtual memory addresses may change
Profile-Driven

Cache Allocation

**PROBLEM**

Caches are critical, constrained resources. Optimal allocation?

**PROFILE MEMORY**

Extract memory traces and produce memory usage profile.

**Process Address Space**

- data
- text
- heap

**Application Profile**

<table>
<thead>
<tr>
<th>Page</th>
<th>Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100 K</td>
</tr>
<tr>
<td>B</td>
<td>10 K</td>
</tr>
<tr>
<td>C</td>
<td>1 K</td>
</tr>
<tr>
<td>D</td>
<td>700</td>
</tr>
<tr>
<td>E</td>
<td>500</td>
</tr>
<tr>
<td>F</td>
<td>90</td>
</tr>
<tr>
<td>C</td>
<td>50</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
</tr>
</tbody>
</table>

**threshold** ${\mu}$

residual cache misses

Location of hot region(s) is unknown

Absolute virtual memory addresses may change
COLORING

Ways

Sets

- Leverages on the virtual $\rightarrow$ physical translation layer
- Used to move page mapping across sets (up/down)
- Transparent to the application
COLORING

- Leverages on the virtual → physical translation layer
- Used to move page mapping across sets (up/down)
- Transparent to the application

LOCKDOWN

- Uses architecture-specific lockdown features
- Used to allocate pages on selected ways (left/right)
- Once allocated, pages trigger cache hits until deallocation
Task tracing + analysis
Profile generation

offline

online
Task tracing + analysis
Profile generation
Profile load
Color detection + re-assignment
Cache locking
✔
✔
Task cleanup
Task tracing + analysis

Profile generation

Profile load

Color detection + re-assignment

Cache locking

Task cleanup

If cache is large enough, allocation is performed

VERIFIED PROPERTIES

✔

✔
If cache is large enough, allocation is performed

If allocation is performed, corresponding physical addresses locked in cache

VERIFIED PROPERTIES

✔ ✔ ✔ ✔
If cache is **large enough**, allocation is performed

If allocation is performed, corresponding physical addresses **locked in cache**

No **extra** memory locked in cache
If cache is large enough, allocation is performed. If allocation is performed, corresponding physical addresses locked in cache. No extra memory locked in cache.

All temporary kernel resources released at the end.
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If cache is <strong>large enough</strong>, allocation is performed.</td>
</tr>
<tr>
<td>2</td>
<td>If allocation is performed, corresponding physical addresses locked in cache.</td>
</tr>
<tr>
<td>3</td>
<td>No <strong>extra</strong> memory locked in cache.</td>
</tr>
<tr>
<td>4</td>
<td>All temporary <strong>kernel resources</strong> released at the end.</td>
</tr>
</tbody>
</table>
what was
Verified

1. After locking issued, line is in cache
2. Profile is correct
3. No extra memory locked in cache
4. All temporary kernel resources released at the end

what was
Assumed

1. If cache is large enough, allocation is performed
2. If allocation is performed, corresponding physical addresses locked in cache
**what was**

**Verified**

1. After locking issued, line is in cache
2. Profile is correct
3. No extra memory locked in cache
4. All temporary *kernel resources* released at the end

**what was**

**Assumed**

1. If cache is *large enough*, allocation is performed
2. If allocation is performed, corresponding physical addresses locked in cache
3. Kernel descriptors correct
what was

Verified

1. After locking issued, line is in cache
2. Profile is correct
3. Kernel descriptors correct
4. Kernel routines correct

what was

Assumed

4. All temporary kernel resources released at the end
3. No extra memory locked in cache
2. If allocation is performed, corresponding physical addresses locked in cache
1. If cache is large enough, allocation is performed
what was Verified

1. If cache is large enough, allocation is performed
2. If allocation is performed, corresponding physical addresses locked in cache
3. No extra memory locked in cache
4. All temporary kernel resources released at the end
5. Initial status of lockdown bit is known

what was Assumed

1. After locking issued, line is in cache
2. Profile is correct
3. Kernel descriptors correct
4. Kernel routines correct
5. Initial status of lockdown bit is known
Verification
Boundaries

Verified
Logic
Verification
Boundaries

User-Space

Hardware

Template Generic Profile

Verified

Logic

Instantiate memory model

Abstract cache model

Verified
Verification Boundaries

User-Space

Hardware

Verified Logic

Template Generic Profile

Initialize descriptors

Abstract & replace routines

Instantiate memory model

Abstract cache model
Set Associative Cache
typedef struct {
    void * addr;
    char locked;
} cache_line_t;
Set Associative Cache

typedef cache_line_t cache_set_t [CACHE_ASSOC];

typedef struct {
    void * addr;
    char locked;
} cache_line_t;
typedef cache_set_t cache_t [CACHE_NSETS];

typedef cache_line_t cache_set_t [CACHE_ASSOC];

typedef struct {
    void *addr;
    char locked;
} cache_line_t;
Set Associative Cache

Cache Model & Locking
Set Associative Cache

typedef cache_set_t cache_t [CACHE_NSETS];

typedef cache_line_t cache_set_t [CACHE_ASSOC];

typedef struct {
    void * addr;
    char locked;
} cache_line_t;
Set Associative Cache

typedef cache_set_t cache_t [CACHE_NSETS];

typedef cache_line_t cache_set_t [CACHE_ASSOC];

typedef struct {
    void * addr;
    char locked;
} cache_line_t;

DCBTLS in HW
Data Cache Block Touch & Lock Set

Cache Model & Locking
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

**Main parameters**
(as of kernel 3.0-rc7)

- `struct task_struck * tsk`
- `struct mm_struct * mm`
- `unsigned long start`
- `unsigned long nr_pages`
- `int write, int force`
- `struct page ** pages`
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

**Main Parameters**
(as of kernel 3.0-rc7)

```c
struct task_struck * tsk
struct mm_struct * mm
unsigned long start
unsigned long nr_pages
int write, int force
struct page ** pages
```
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

**main parameters**
( as of kernel 3.0-rc7 )

```c
struct task_struct * tsk
struct mm_struct * mm
unsigned long start
unsigned long nr_pages
int write, int force
struct page ** pages
```

```c
struct page * page_ptr;
```
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

main parameters (as of kernel 3.0-rc7)

- struct task_struck * tsk
- struct mm_struct * mm
- unsigned long start
- unsigned long nr_pages
- int write, int force
- struct page ** pages

 assert(nr_pages == 1);
Abstraction of Kernel Routine

get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

main parameters (as of kernel 3.0-rc7)

- struct task_struck * tsk
- struct mm_struct * mm
- unsigned long start
- unsigned long nr_pages
- int write, int force
- struct page ** pages

{ 
    struct page * page_ptr;
    assert(nr_pages == 1);
    assert(write == 0);

}
**get_user_pages(...)**

used to pin physical pages, returns page descriptors from virtual addresses

---

**main parameters**

(as of kernel 3.0-rc7)

- `struct task_struck * tsk`
- `struct mm_struct * mm`
- `unsigned long start`
- `unsigned long nr_pages`
- `int write, int force`
- `struct page ** pages`

---

```c
assert(force == 0);
assert(nr_pages == 1);
assert(write == 0);
```
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

**main parameters**
(as of kernel 3.0-rc7)

- `struct task_struck * tsk`
- `struct mm_struct * mm`
- `unsigned long start`
- `unsigned long nr_pages`
- `int write, int force`
- `struct page ** pages`

```c
assert(tsk == current);
```
Abstraction of
Kernel Routine

get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

main parameters (as of kernel 3.0-rc7)

- struct task_struck * tsk
- struct mm_struct * mm
- unsigned long start
- unsigned long nr_pages
- int write, int force
- struct page ** pages

{  
  struct page * page_ptr;
  assert(nr_pages == 1);
  assert(write == 0);
  assert(force == 0);
  assert(tsk == current);
  assert(mm == tsk->mm);
}
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

**main parameters**
(as of kernel 3.0-rc7)

- struct task_struck * tsk
- struct mm_struct * mm
- unsigned long start
- unsigned long nr_pages
- int write, int force
- struct page ** pages

```c
struct page * page_ptr;
assert(nr_pages == 1);
assert(write == 0);
assert(force == 0);
assert(tsk == current);
assert(mm == tsk->mm);
page_ptr = __CPROVER_ui_void_ptr(...);
```
get_user_pages(...)

used to pin physical pages, returns page descriptors from virtual addresses

**main parameters**
(as of kernel 3.0-rc7)

- `struct task_struck * tsk`
- `struct mm_struct * mm`
- `unsigned long start`
- `unsigned long nr_pages`
- `int write, int force`
- `struct page ** pages`

```c
{  
  struct page * page_ptr;
  assert(nr_pages == 1);
  assert(write == 0);
  assert(force == 0);
  assert(tsk == current);
  assert(mm == tsk->mm);
  page_ptr = __CPROVER_ui_void_ptr(...);
  __CPROVER_assume(page_ptr >= mem_map &&
                    page_ptr < (mem_map + MAX_PAGES));
}
```
**get_user_pages(...)**

used to pin physical pages, returns page descriptors from virtual addresses

---

**main parameters**

(AS OF KERNEL 3.0-RC7)

- `struct task_struck * tsk`
- `struct mm_struct * mm`
- `unsigned long start`
- `unsigned long nr_pages`
- `int write, int force`
- `struct page ** pages`

---

```c
{  
  struct page * page_ptr;  
  assert(nr_pages == 1);  
  assert(write == 0);  
  assert(force == 0);  
  assert(tsk == current);  
  assert(mm == tsk->mm);  
  page_ptr = ___CPROVER-ui-void_ptr(...);  
  ___CPROVER_assume(page_ptr >= mem_map &&  
                     page_ptr < (mem_map + MAX_PAGES));  
  ___CPROVER_assume(ALIGNED_TO(page_ptr,  
                        sizeof(struct page)));  
}
```
get_user_pages(...) used to pin physical pages, returns page descriptors from virtual addresses

**main parameters**
(as of kernel 3.0-rc7)

```c
struct task_struck * tsk
struct mm_struct * mm
unsigned long start
unsigned long nr_pages
int write, int force
struct page ** pages
```

```c
struct page * page_ptr;
assert(nr_pages == 1);
assert(write == 0);
assert(force == 0);
assert(tsk == current);
assert(mm == tsk->mm);
page_ptr = __CPROVER_ui_void_ptr(...);
__CPROVER_assume(page_ptr >= mem_map &&
    page_ptr < (mem_map + MAX_PAGES));
__CPROVER_assume(ALIGNED_TO(page_ptr, sizeof(struct page)));
*pages = page_ptr;
```
**get_user_pages(...)**
used to pin physical pages, returns page descriptors from virtual addresses

**main parameters**
(as of kernel 3.0-rc7)

- `struct task_struck * tsk`
- `struct mm_struct * mm`
- `unsigned long start`
- `unsigned long nr_pages`
- `int write, int force`
- `struct page ** pages`

```c
{
    struct page * page_ptr;
    assert(nr_pages == 1);
    assert(write == 0);
    assert(force == 0);
    assert(tsk == current);
    assert(mm == tsk->mm);
    page_ptr = __CPROVER_ui_void_ptr(...);
    __CPROVER_assume(page_ptr >= mem_map &&
                      page_ptr < (mem_map + MAX_PAGES));
    __CPROVER_assume(ALIGNED_TO(page_ptr,
                              sizeof(struct page)));
    *pages = page_ptr;
    return 1;
}
```
Evaluation

Memory Instance #1

From cache controller's perspective
- Tag (23)
- Index (3)
- Offset (6)

From OS's perspective
- Page Frame Number (24)
- Offset (8)
- Color (1)
Evaluation

Memory Instance #1

From cache controller’s perspective
- Tag (23)
- Index (3)
- Offset (6)

From OS’s perspective
- Page Frame Number (24)
- Offset (8)
- Color (1)

Memory Instance #2

From cache controller’s perspective
- Tag (16)
- Index (10)
- Offset (6)

From OS’s perspective
- Page Frame Number (20)
- Offset (12)
- Color (4)

Graph:
- x-axis: Number of profile pages
- y-axis: Time (s)
- Line graph with Assoc. 1
Colored Lockdown via CBMC

Summary

Hardware

User-Space

Kernel

Hardware
Colored Lockdown
via CBMC

Summary

successful with realistic memory model
Colored Lockdown via CBMC

Summary

User-Space

Kernel

Hardware

scalability needed

+ + + + + + + + +
Thanks.
rmancuso@bu.edu