Universitat Oberta de Catalunya

Desenvolupament mòdul per traduir adreces virtuals en sistemes GNU / Linux

Arquitectures x86 i x86_64

Enginyeria Informàtica (2on Cicle)

José Antonio Martín Pérez

Consultor: Francesc Guim Bernat

Juny 2011

Índex

1. Introducció

1.1. Motivació del projecte

Aquest projecte final de carrera està inclòs dintre de l'àrea d'Arquitectura de Computadors i Sistemes Operatius.

És interessant i curiós que, en un món dominat per sistemes operatius de tipus comercial, el que va ser en un inici un petit projecte d'un estudiant finlandès hagi acabat donant un dels sistemes operatius més potents de l'actualitat, gràcies a un desenvolupament conjunt de centenar de persones al voltant del món.

Aquesta natura lliure del projecte ha permès que un mateix sistema hagi sigut possible portar-lo a un ampli ventall d'arquitectures diferents, des de grans ordinadors destinats a la recerca com a petits dispositius i telèfons intel·ligents.

El sistema gaudeix d'una gran fama i prestigi que el fan ser el sistema operatiu de referència tant a les universitats per fer recerca com la base per donar serveis a la xarxa. No és estrany que sigui la plataforma base e grans empreses com Google[1] o Facebook[2].

El nucli del sistema operatiu Linux és de tipus monolític, concentrant totes les funcionalitats , com gestió de processos, memòria o sistema de fitxers, dintre d'un programa complexe que interactua directament amb el hardware del sistema i proporciona un espai d'usuari per l'execució tant de les interfícies gràfiques com de les aplicacions del sistema.

El sistema GNU/Linux[3] dóna la possibilitat de desenvolupar petites aplicacions que el nucli del sistema, també anomenat *kernel*[4], carrega de forma dinàmica per tal de proporcionar noves funcionalitats i la possibilitat d'adaptar el sistema a unes necessitats específiques.

Dins d'aquest marc és on es situa aquest projecte, el desenvolupament d'un mòdul per al nucli del sistema operatiu que permeti traduir adreces virtuals a físiques, per a un procés determinat i poder comparar aquest mètode de traducció amb un altre ja implementat, que fa servir un mètode totalment diferent.

Diferents proves i tests ens determinaran quin dels dos mètodes proporciona una major velocitat en la resolució d'adreces per a processos complexos.

---

[1] Plana web: www.google.com
[2] Plana web: www.facebook.com
[3] Més informació sobre el sistema operatiu GNU/Linux: http://en.wikipedia.org/wiki/GNU/Linux
[4] Informació addicional sobre el nucli del sistema, anomenat *kernel*: http://en.wikipedia.org/wiki/Kernel_(computing)

1.2. Objectius del projecte

Els objectius definits per aquest projecte han sigut els següents:

- Desenvolupament d'un mòdul en llenguatge C per al nucli del sistema operatiu GNU/Linux que proporcioni informació sobre quines adreces físiques accedeixen els diferents fils del sistema.

- Aquest mòdul farà una traducció de les adreces virtuals per a un procés donat, consultant les seves taules de pàgina.

- Es desenvoluparan dues versions del mòdul, per a les següents arquitectures:
  - Intel(R) x86
  - Intel(R) x86_64

- Comparar la traducció d'adreces obtinguda amb el mòdul i l'actual, fent servir el mètode Pagemap.

- Ús de l'eina d'instrumentació Pin durant el procés d'instrumentació i proves amb el mòdul descrit.

- Modificació d'un pintool donat per tal d'afegir l'ús del mòdul desenvolupat i poder instrumentar amb aquest.

- Ús del conjunt de proves NAS Parallel Bechmanks[1] per tal de sotmetre el mòdul a un conjunt de proves d'estrès per avaluar la millora en el rendiment de traducció d'adreces virtuals per a un procés concret.

- Extreure conclusions amb els resultats obtinguts.

[1] Plana web oficial del projecte: http://www.nas.nasa.gov/Resources/Software/npb.html

1.3. Resultat del projecte

Aquest projecte donarà els següent resultats:

- Aquesta memòria on es detalla el desenvolupament del projecte i els resultats finals obtinguts.

- Codi font dels dos mòduls desenvolupats per al nucli del sistema operatiu GNU/Linux, segons l'arquitectura.

- Codi font del pintool, anomenat 'mtaptrace', amb les modificacions oportunes per tal d'afegir una nova casuística amb el mòdul desenvolupat.

- Presentació virtual del projecte.

2. Pla de treball

2.1. Llistat d'activitats

El projecte s'ha dividit en quatre fases. A continuació es detalla cadascuna d'elles.

Fase 1

- Definir les motivacions que han portat a la realització d'aquest projecte i marca els objectius que es volen assolir.
- Donar el pla de treball que es seguirà durant el desenvolupament d'aquest.

Fase 2

- Estudi del nucli de Linux.
- Estudi dels mòduls per al nucli. Teoria i proves de desenvolupament.
- Avaluació de les diferents possibilitats de desenvolupament.

Fase 3

- Presentació de la solució escollida.
- Desenvolupament del mòdul per al nucli seguint la solució anterior.
- Conjunt de proves per avaluar el correcte funcionament del mòdul.

Fase 4

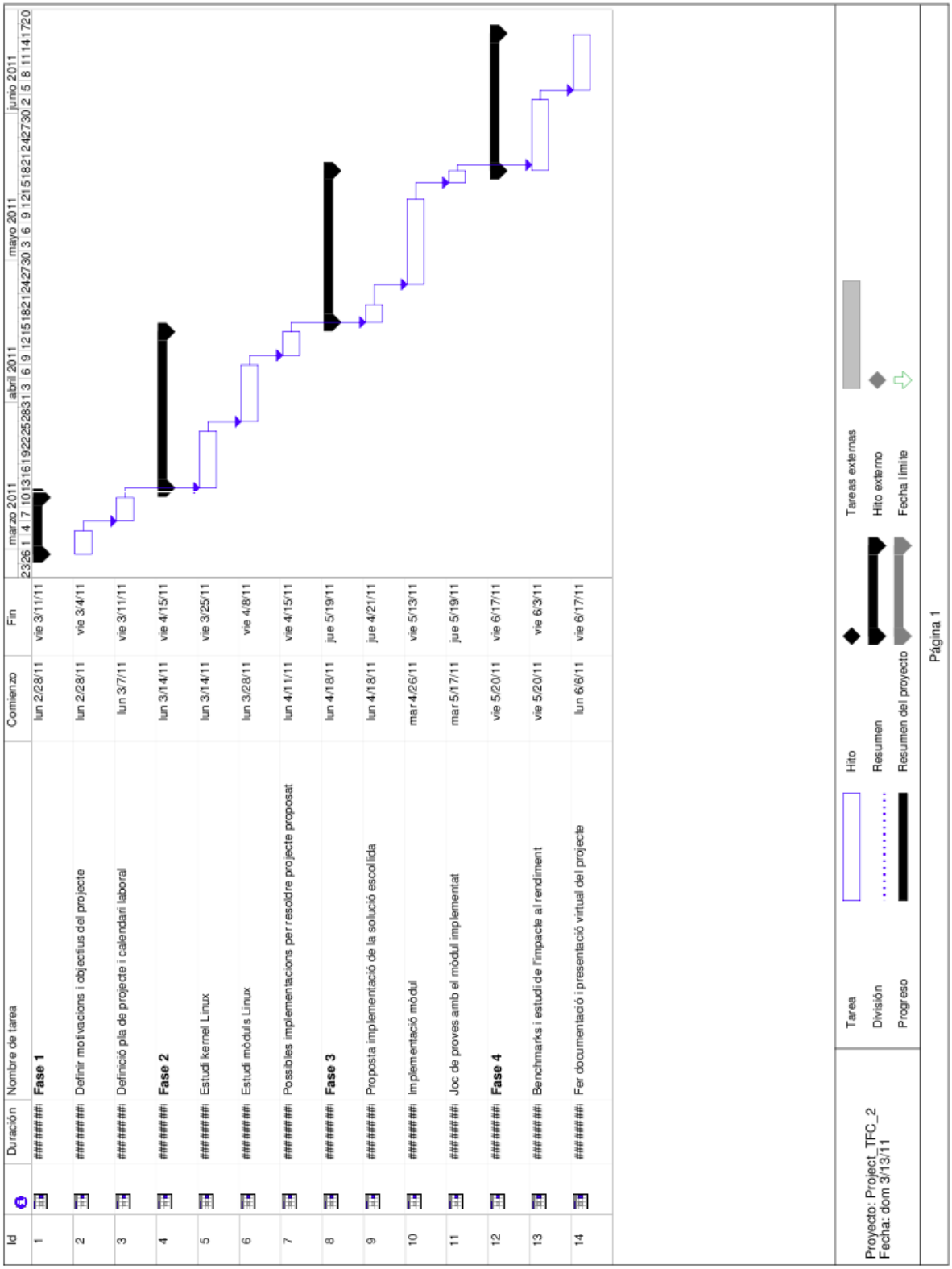
- Tests d'estrès per avaluar el rendiment durant el procés d'instrumentació amb l'eina Pin i NAS Parallel Benchmarks.
- Elaboració de la documentació i la presentació virtual.

2.2. Calendari de treball

El calendari de treball està definit seguint l'horari de treball habitual, de dilluns a divendres i deixant lliures els cap de setmana i els possibles dies festius que pugui haver.

L'inici del projecte s'ha definit el dia 28 de Febrer de 2011 i la finalització el dia 17 de Juny de 2011. Tenim un total de 16 setmanes, de les quals hem d'eliminar dos dies festius: 22 i 25 d'Abril. Per tant, tenim 78 dies laborals per al desenvolupament del projecte.

A continuació s'adjunta el diagrama de Gantt os es mostra el calendari establert per al projecte i es detallen les quatre fases principals del projecte, així com les diferents etapes que conformen cada fase.


Id
Duración
Nombre de tarea
Comienzo
Fin
marzo 2011
abril 2011
mayo 2011
junio 2011
1 ######## Fase 1 lun 2/28/11 vie 3/11/11
2 ######## Definir motivacions i objectius del projecte lun 2/28/11 vie 3/4/11
3 ######## Definició pla de projecte i calendari laboral lun 3/7/11 vie 3/11/11
4 ######## Fase 2 lun 3/14/11 vie 4/15/11
5 ######## Estudi kernel Linux lun 3/14/11 vie 3/25/11
6 ######## Estudi mòduls Linux lun 3/28/11 vie 4/8/11
7 ######## Possibles implementacions per resoldre projecte proposat lun 4/11/11 vie 4/15/11
8 ######## Fase 3 lun 4/18/11 jue 5/19/11
9 ######## Proposta implementació de la solució escollida lun 4/18/11 jue 4/21/11
10 ######## Implementació mòdul mar 4/26/11 vie 5/13/11
11 ######## Joc de proves amb el mòdul implementat mar 5/17/11 jue 5/19/11
12 ######## Fase 4 vie 5/20/11 vie 6/17/11
13 ######## Benchmarks i estudi de l'impacte al rendiment vie 5/20/11 vie 6/3/11
14 ######## Fer documentació i presentació virtual del projecte lun 6/6/11 vie 6/17/11
Proyecto: Project_TFC_2
Fecha: dom 3/13/11
Tarea
División
Progreso
Hito
Resumen
Resumen del proyecto
Tareas externas
Hito externo
Fecha límite
Página 1

3. Gestor de memòria en sistemes Linux

3.1. Pàgines

El nucli de Linux divideix la memòria que té disponible en la unitat més petita amb la qual pot treballar. Aquesta unitat se la coneix amb el nom de pàgina [1].

L'arquitectura del sistema determina la mida de pàgina que farà servir el nucli. En arquitectures de 32 bits, la mida de pàgina habitual és de 4KB. En canvi, en arquitectures de 64 bits, aquesta mida varia, sent la mida més habitual de 8KB [2].

La idea de fer servir pàgines per manegar la memòria ve del fet és que, per tal de treballar amb la memòria física del sistema, el nucli divideix la mida de la memòria en pàgines. D'aquesta manera, el número total de pàgines amb les quals treballarà el *kernel* ve donat pel resultat de dividir la mida total de la memòria per la mida de pàgina.

Cada pàgina es representada al sistema fent servir una estructura anomenada *struct page* que relaciona aquestes amb les pàgines físiques. És important notar que la idea és que el nucli disposi d'una descripció de la memòria, no les dades contingudes en aquesta. Per tant, l'objectiu de l'estructura *page* és descriure la seva pàgina homònima a memòria física.

Trobem aquesta estructura definida a *<linux/mm_types.h>.*

Definició de l'estructura *page*:

```
struct page  {
     unsigned long         flags;
     atomic_t              _count;
     atomic_t              _mapcount;
     unsigned long         private;
     struct address_space  *mapping;
     pgoff_t               index;
     struct list_head      lru;
     void                  *virtual;
};
```

[1] En anglès, *page*.
[2] La mida de la pàgina depèn força de l'arquitectura.

Aquesta és una versió simplificada de l'estructura *page* només per mostrar una idea de com s'implementa aquesta amb els principals camps.

Seguidament veiem quina és la finalitat de cadascun d'ells.

- *flags* informació sobre l'estat de la pàgina.
- *_count* número de referències a la pàgina. Quan pren el valor -1 significa que no està referenciada per ningú.
- **virtual* adreça de la pàgina a memòria física.
- **mapping* quan la pàgina és usada per la cache de pàgines (*cache page*).
- *private* quan la pàgina conté dades de tipus privat.

Per manegar el conjunt de pàgines disponibles, el nucli defineix un array, anomenat *mem_map*, on cada posició identifica cadascuna de les pàgines que el sistema té definides.

3.2. Nodes

El sistema divideix la memòria total en els anomenats *bancs*, com per exemple, la memòria cau del processador o aquella més propera als dispositius per fer les tasques de DMA [1].

El nucli anomena a cadascun d'aquests bancs com a nodes i els defineix fent servir una estructura anomenada *struct pglist_data*. Aquesta estructura es referenciada fent servir el nom *pg_data_t*.

El conjunt de nodes que el sistema té definits s'agrupen en una llista, anomenada *pgdat_list*, terminada en NULL.

Agrupació dels nodes

L'estructura *pglist_data* es troba definida a <*linux/mmzone.h*>.

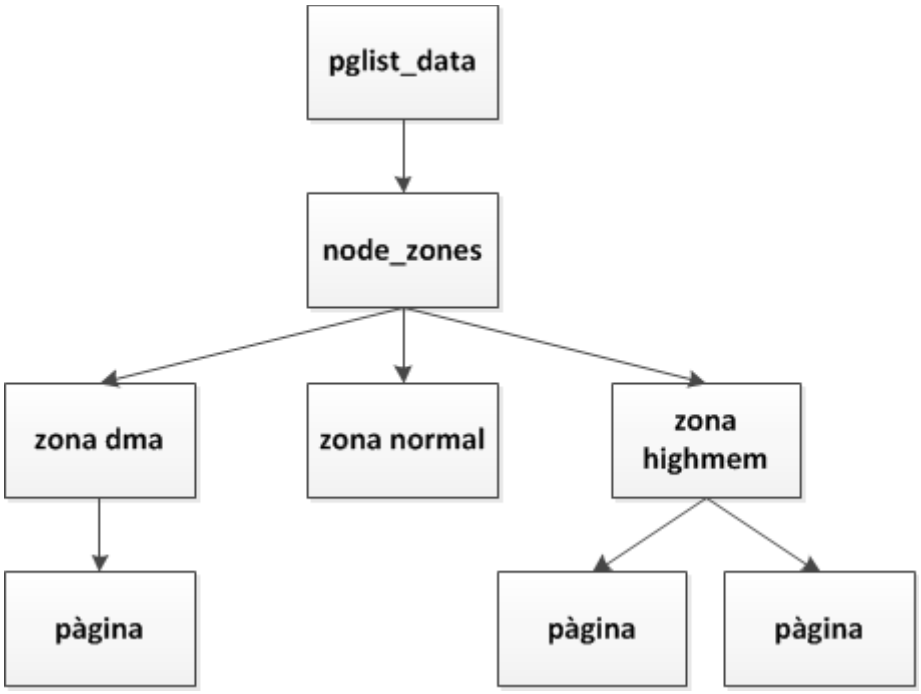
Definició de l'estructura *pglist_data*:

```
typedef struct pglist_data  {
      zone_t                node_zones[MAX_NR_ZONES];
      zonelist_t            node_zonelists[GFP_ZONEMASK+1];
      int                   nr_zones;
      struct page           *node_mem_map;
      unsigned long         *valid_addr_bitmap;
      struct bootmem_data   *bdata;
      unsigned long         node_start_paddr;
      unsigned long         node_start_mapnr;
      unsigned long         node_size;
      int                   node_id;
      struct pglist_data    *node_next;
} pg_data_t;
```

[1] El DMA (de l'anglès *Direct Memory Access*) és un mecanisme que possibilita que diferents subsistemes accedeixin a la memòria del sistema sense tindre que fer servir el processador. D'aquesta manera, el processador queda exclòs d'aquesta tasca i no es consumeixen cicles de CPU.

3.3. Zones

Cada node que el nucli defineix es divideix en tres parts, anomenades zones.



Relació entre nodes, zones i pàgines

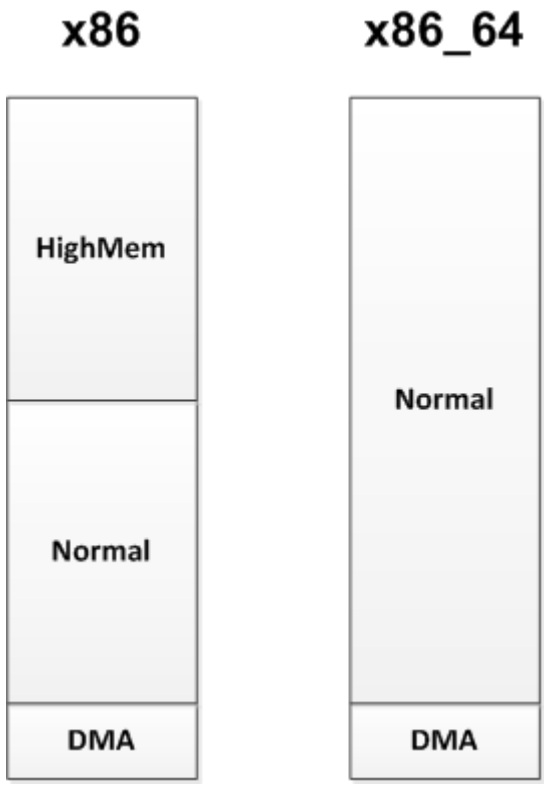
Diferents limitacions *hardware* van propiciar tindre que dividir els nodes en aquestes tres zones [1].

- *zone_dma* Pàgines relacionades amb DMA.
  Són els primers 16MB de memòria física.
- *zone_normal* Pàgines estàndard directament accessibles pel nucli.
  Rang entre els 16MB i 896MB de memòria física.
- *zone_highmem* Rang de pàgines no accessibles directament pel nucli.
  En sistemes de 64 bits, aquesta zona queda sempre buida.
  És la part de la memòria física per damunt dels 896MB.

[1] Aquestes limitacions *hardware* només en sistemes Intel(R) x86 sense PAE (*Physical Address Extension*).

En l'arquitectura Intel(R) x86, les adreces superiors a 896MB es troben definides a la zona '*HighMem'*, en canvi, en l'arquitectura Intel(R) x86_64, aquesta zona sempre quedarà buida i totes les adreces superiors a 16MB passen a estar en la zona '*Normal'*.

En totes dues arquitectures, els primers 16MB es dediquen a l'anomenada '*Zona DMA*'.



Definició de les diferents zones segons l'arquitectura

Cada zona té la seva pròpia estructura que la defineix i que es troba definida a *<linux/mmzone.h>*.

Definició de l'estructura *zone*:

```
struct zone  {
     unsigned long              watermark[NR_WMARK];
     unsigned long              lowmem_reserve[MAX_NR_ZONES];
     struct per_cpu_pageset     pageset[NR_CPUS];
     spinlock_t                 lock;
     struct free_area           free_area[MAX_ORDER];
     spinlock_t                 lru_lock;
     struct zone_lru {
          struct list_headlist  list;
          unsigned long         nr_saved_scan;
     } lru[NR_LRU_LISTS];
     struct zone_reclaim_stat   reclaim_stat;
     unsigned long              pages_scanned;
     unsigned long              flags;
     atomic_long_t              vm_stat[NR_VM_ZONE_STAT_ITEMS];
     int                        prev_priority;
     unsigned int               inactive_ratio;
     wait_queue_head_t          *wait_table;
     unsigned long              wait_table_hash_nr_entries;
     unsigned long              wait_table_bits;
     struct pglist_data         *zone_pgdat;
     unsigned long              zone_start_pfn;
     unsigned long              spanned_pages;
     unsigned long              present_pages;
     const char                 *name;
};
```

L'estructura emmagatzema informació i estadístiques sobre l'ús de les pàgines, espai lliure a la zona i bloquejos.

Seguidament es detallen els principals camps que formen l'estructura:

- *lock*: Control accessos concurrents a la zona.
- *free_pages*: Número total de pàgines lliures a la zona.
- *need_balance*: 'Flag' que indica al dimoni *kswapd* que la zona necessita ser balancejada. Una zona necessita ser balancejada quan el nombre de pàgines lliures passa el límit establert.
- *free_area*: Espais lliures fets servir pel 'Buddy Allocator'.
- *wait_table*: Taula amb la cua de processos en espera que una pàgina sigui alliberada.
- *wait_table_size*: Nombre de processos en cua a la taula anterior.
- *zone_pgdat*: Apunta a l'estructura *pg_data_t* pare.
- *zone_mem_map*: Primera pàgina en *mem_map* a la que aquesta estructura referencia.
- name: Nom de la zona: 'DMA', 'Nomal' o 'HighMem'.
- size: Mida, en nombre de pàgines, de la zona.

3.4. Slab

Una de les tasques més habituals que fa el nucli és crear i destruir estructures. Per tal de facilitar aquesta tasca i evitar fragmentar la memòria amb contínues accessos a memòria, es va implementar un mètode anomenat *slab layer* o *slab allocator* [1].

Aquest mètode es basa en tenir prèviament carregats a memòria les estructures de dades més habituals que fa servir el nucli. D'aquesta manera, quan arriba una petició de creació d'una nova estructura a memòria, el nucli pot assignar-la de forma molt ràpida, ja que l'estructura ja està creada i carregada a memòria.

Cada estructura de dades té definit el seu propi grup, que el nucli anomena *kmem_cache.*

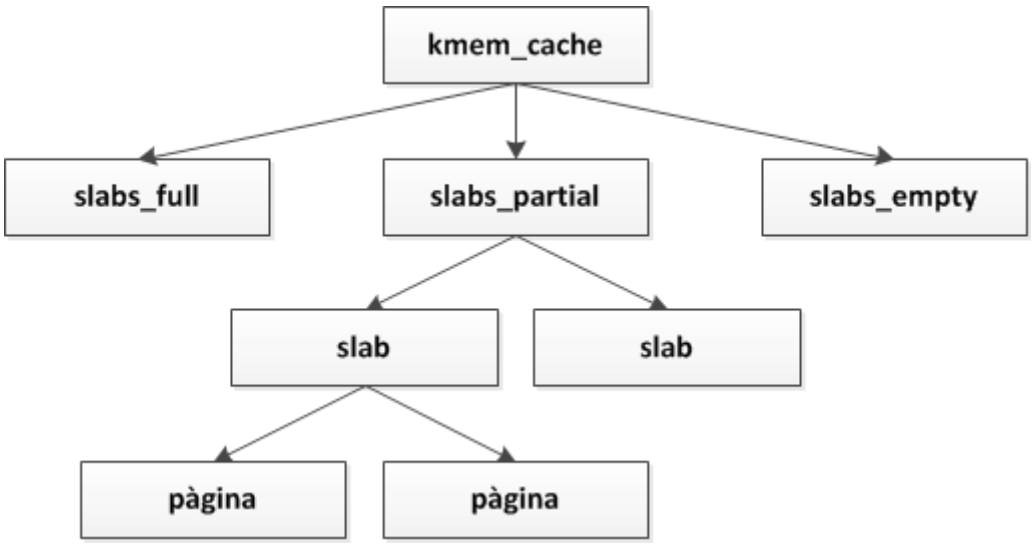


Diagrama de l'estructura kmem_cache

Cada caché es divideix en blocs de memòria contigus (habitualment pàgines contínues), anomenats *slabs.* Hi ha tres *slabs*:

- *slabs_full*: Slabs que estan completament assignats.
- *slabs_partial*: Slabs parcialment assignats.
- *slabs_empty*: Slabs buits, sense objectes assignats.

[1] Més informació sobre *Slab Allocator*: http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/

Dintre d'aquests tres grups, es defineix una llista de *slabs*. De la mateixa manera, cada *slab* es divideix en objectes. Aquests objectes són els elements fonamentals que s'assignen i desassignen. El *slab* és la unitat més petita que por assignar l'algoritme '*Slab Allocator*'. Per tant, si necessita créixer, aquesta serà la unitat mínima en que ho podrà fer. Cada *slab*, habitualment, es composa de diferents tipus d'objectes.

Definició de l'estructura *slab*:

```
struct slab  {
     struct list_head     list;
     unsigned long        colouroff;
     void                 *s_mem;
     unsigned int         inuse;
     kmem_bufctl_t        free;
};
```

Descripció dels camps que formen l'estructura anterior:

- *list*: Llista a la qual pertany el slab. Hi ha tres possibilitats: '*slabs_full*' , '*slabs_partial*' o '*slabs_empty*'.
- *colouroff*: Desplaçament respecte a l'adreça base del primer objecte del *slab*. L'adreça d'aquest primer objecte és *s_mem + colouroff*.
- *s_mem*: Adreça del primer objecte.
- *inuse*: Nombre d'objectes actius al *slab*.
- *free*: Array usat per emmagatzemar la localització dels objectes lliures.

3.5. Espai d'adreces de processos

El nucli, a més a més de manegar la seva pròpia memòria, també s'encarrega de manegar la memòria dels processos d'usuari. Aquest espai de memòria és conegut com l'espai d'adreces de processos [1].

Aquest espai, encara que pugui ser vist de forma lineal des del punt de vista de l'usuari, realment es divideix en dues parts: espai d'usuari, que canvia en cada canvi de context, i l'espai d'adreces del nucli, que romandrà constant.

L'estructura *mm_struct* s'encarrega de manegar l'espai d'adreces que els processos poden fer servir. Es troba definida a <*linux/sched.h*>.

Definició de l'estructura *mm_struct*:

```
struct mm_struct  {
      struct vm_area_struct      * mmap;
      rb_root_t                  mm_rb;
      struct vm_area_struct      * mmap_cache;
      pgd_t                      * pgd;
      atomic_t                   mm_users;
      atomic_t                   mm_count;
      int                        map_count;
      struct rw_semaphore        mmap_sem;
      spinlock_t                 page_table_lock;

      struct list_head           mmlist;

      unsigned long              start_code, end_code, start_data, end_data;
      unsigned long              start_brk, brk, start_stack;
      unsigned long              arg_start, arg_end, env_start, env_end;
      unsigned long              rss, total_vm, locked_vm;
      unsigned long              def_flags;
      unsigned long              cpu_vm_mask;
      unsigned long              swap_address;

      unsigned                   dumpable:1;

      /* Architecture-specific MM context */
      mm_context_t               context;
};
```

[1] En anglès, *Process Address Space*.

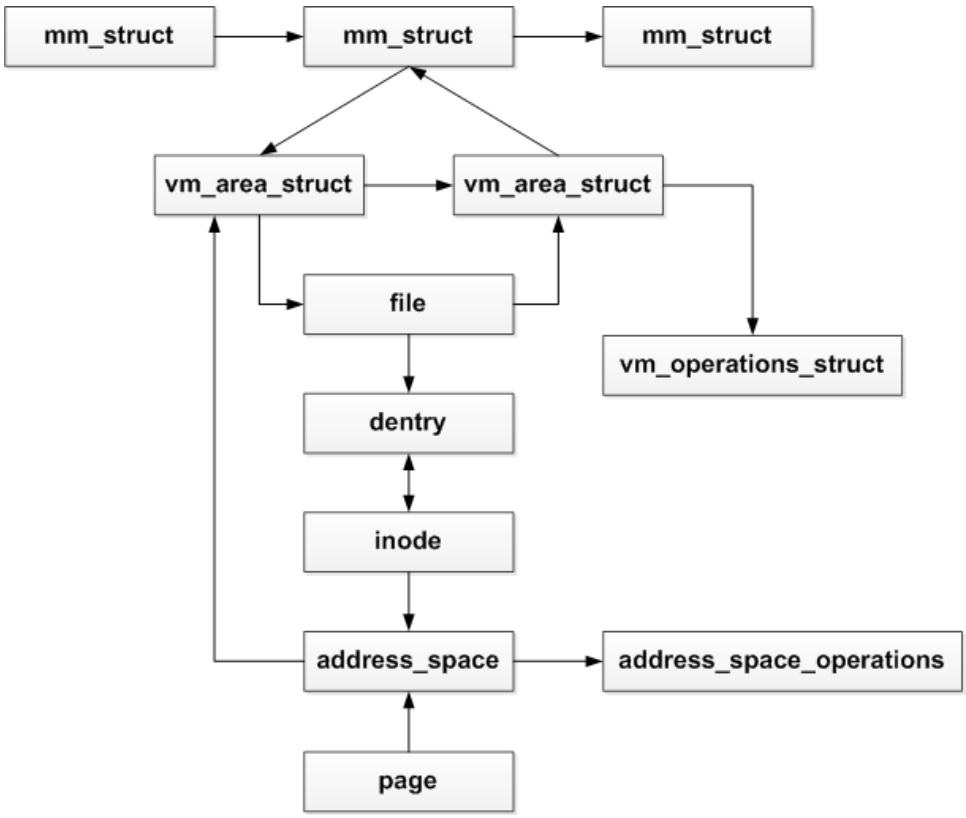
Descripció dels camps que formen l'estructura anterior:

- *mmap*: Inici de la llista amb totes les àrees VMA en l'espai d'adreces.
- *mm_rb*: Arrel de l'arbre (*red black tree*[1]) en que s'organitzen les VMA per cercar-les ràpidament.
- *mmap_cache*: VMA trobada amb la darrera crida a la funció *find_vma()*.
- *pgd*: '*Page Global Directory*' para aquest procés.
- *mm_users*: Nombre d'usuaris que accedeixen a l'espai d'adreces.
- *mm_count*: Nombre d'usuaris anònims per l'estructura.
- *map_count*: Nombre de VMA's en ús.
- *mmap_sem*: Bloqueja la llista de VMA's tant per lectures com per escriptures.
- *page_table_lock*: Protegeix molts camps d'aquesta estructura d'accessos concurrents.
- *mmlist*: Totes les estructures mm_struct estan unides en una llista fent servir aquest camp.
- *star_code,end_code*: Adreça inicial i final de la secció de codi.
- *start_data,end_data*: Adreça inicial i final de la secció de dades.
- *star_brk,brk*: Adreça inicial i final del heap[2].
- *start_stack*: Inici de la pila.
- *arg_start,arg_end*: Adreça inicial i final dels argument per la línia de comandes.
- *env_start,env_end*: Adreça inicial i final de les variables d'entorn.
- *rss*: Nombre de pàgines residents per al procés.
- *total_vm*: Espai total ocupat per totes les VMA's del procés.
- *locked_vm*: Nombre de pàgines bloquejades a memòria.
- *def_flags*: Només pot tindre un valor, VM_LOCKED. Determina si els mapejos futurs són bloquejats per defecte.
- *cpu_vm_mask*: Representació de les possibles CPU's en sistemes multi-processador (SMP). És important durant el procés '*TLB flush*[3]' per cada CPU.
- *swap_address*: Registra la darrera paginada quan el procés ha sigut enterament passat a l'àrea de swap.
- *dumpable*: Flag important quan es rastreja un procés.
- *context*: Context MMU[4] específic a l'arquitectura.

[1] Més informació de l'estructura *Red Black Tree*: http://en.wikipedia.org/wiki/Red-black_tree
[2] Més informació de l'estructura *Heap*: http://en.wikipedia.org/wiki/Heap_(data_structure)
[3] Acrònim de *Translation Lookaside Buffer*: http://en.wikipedia.org/wiki/Translation_lookaside_buffer
[4] Acrònim de *Memory Management Unit*: http://en.wikipedia.org/wiki/Memory_management_unit

La memòria es divideix en àrees, que el nucli representa fent servir l'estructura *vm_area_struct*. Es troba definida a <*linux/mm_types.h*>.

Cada estructura defineix un àrea de memòria contigua dins d'un espai d'adreces donat i el nucli tracta cada àrea de memòria com un únic objecte.



Relació entre les estructures mm_struct i vm_area_struct

Si una regió s'acompanya d'un fitxer, el camp '*vm_file*' estarà present. Aquest camp es pot trobar a l'estructura '*vm_area_struct*'. Si es vol obtindre l'estructura '*adress_space*' associada a la regió, cal seguir els següents camps:

vm_file (estructura *vm_area_struct*) --> f_dentry (estructura *file*) --> d_inode (estructura *dentry*) --> i_mapping (estructura *inode*)

L'estructura 'address_space' conté tota la informació específica al sistema de fitxers necessària per realitzar operacions relacionades amb les pàgines a disc.

Definició de l'estructura *vm_area_struct*:

```
struct vm_area_struct {
  struct mm_struct * vm_mm;
  unsigned long vm_start;
  unsigned long vm_end;

  /* linked list of VM areas per task, sorted by address */
  struct vm_area_struct *vm_next;

  pgprot_t vm_page_prot;
  unsigned long vm_flags;

  rb_node_t vm_rb;

  struct vm_area_struct *vm_next_share;
  struct vm_area_struct **vm_pprev_share;

  /* Function pointers to deal with this struct. */
  struct vm_operations_struct * vm_ops;

  /* Information about our backing store: */
  unsigned long vm_pgoff;
  struct file * vm_file;
  unsigned long vm_raend;
  void * vm_private_data;
};
```

Descripció dels camps de l'estructura:

- *vm_mm*: A quin *mm_struct* pertany aquesta VMA.
- *vm_start*: Adreça d'inici de la regió.
- *vm_end*: Adreça final de la regió.
- *vm_next*: Donat que totes les VMA's formen una llista, aquest camp indica quina és la següent VMA de la llista.
- *vm_page_prot*: Flag de protecció per cada PTE ('*Page Table*') en aquesta VMA.
- *vm_flags*: Flags de protecció i propietats de la VMA.
- *vm_rb*: Per fer cerques ràpidament, les VMA's s'emmagatzemen també en un arbre (*Red Black Tree*), de la mateixa manera que les estructures *mm_struct*.
- *vm_next_share*: Enllaça VMA compartides pel mapeig d'arxius.
- *vm_pprev_share*: Complement de *vm_next_share*.
- *vm_ops*: Punters de funció per open(), close() i nopage().
- *vm_pgoff*: Desplaçament de pàgina per a un fitxer mapejat a memòria.
- *vm_file*: Punter a l'estructura *file* del fitxer mapejat.
- vm_raend: Adreça final d'una finestra de lectura.
- vm_private_data: Emmagatzema informació privada que es usada per alguns controladors de dispositius.

3.6. Taules de pàgina

Encara que els diferents processos que treballen es espai d'usuari fan servir adreces virtual, l'anomenada memòria virtual, el processador, en rebre una petició d'accés a memòria necessita una adreça física.

Aquesta necessitat de traduir les adreces virtuals a físiques va ser solucionada amb la implementació de les anomenades taules de pàgina[1].

En l'arquitectura x86, les taules de pàgina es divideixen en tres nivells[2]:

- PGD o *page global directory*
  Array que conté estructures de tipus *pgd_t*.
  Aquesta estructura és habitual que sigui del tipus *unsigned long*.
  Són punters [3] al segon nivell.

- PMD o *page middle directory*
  Array d'estructures *pmd_t*.
  Punters [3] al tercer nivell.

- PTE o *page table*
  Array d'estructures *pte_t*.
  Punters [3] a adreces físiques.

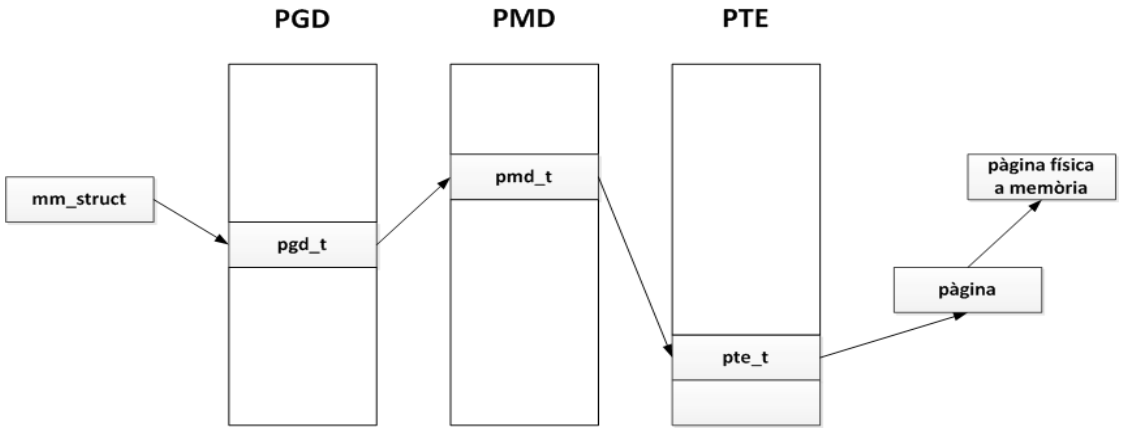


Diagrama de les taules de pàgina en l'arquitectura Intel(R) x86

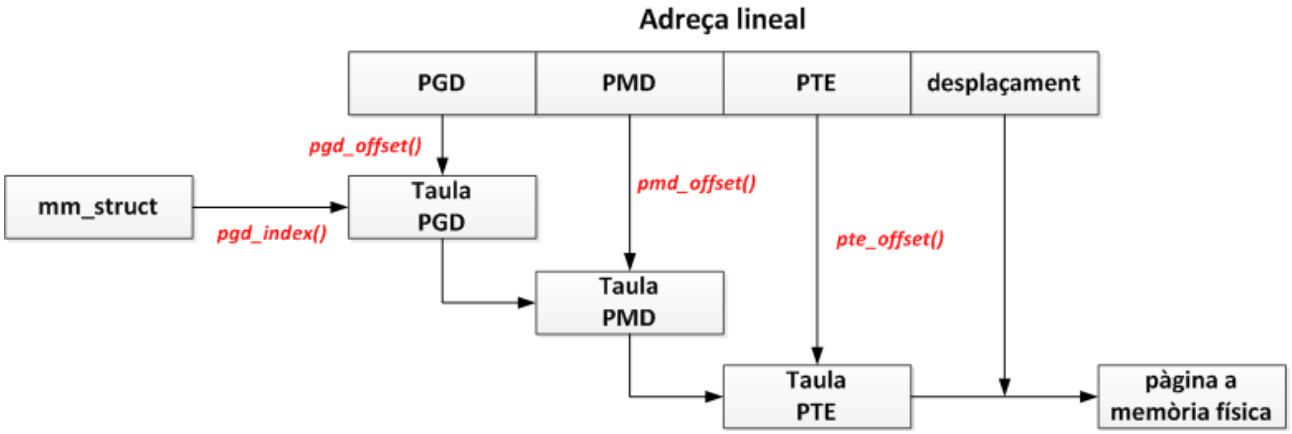
[1] De l'anglès. *Page Tables*.
[2] Aquesta és la configuració més habitual, encara que depèn de l'arquitectura.
[3] Aquests punters són coneguts com *Page Frame Numbers*.

A més a més de les tres estructures definides a les tres taules de pàgina (*pgd_t* , *pmd_t* i *pte_t*) també es defineix una quarta estructura, anomenada *pgprot_t*, que emmagatzema els bits de protecció i que es fan servir a mode de *flags*.

Bits	Funció
PAGE_PRESENT	Pàgina present a memòria física
PAGE_PROTNONE	Pàgina no accesible
PAGE_RW	Determina si la pàgina ha de ser escrita
PAGE_USER	Determina si la pàgina pot ser accedida des de l'espai d'usuari
PAGE_DIRTY	Determina si la pàgina ha sigut escrita
PAGE_ACCESSED	Determina si la pàgina és accesible

Una adreça lineal es composa de quatre camps:

- Desplaçament del PGD del procés
- Desplaçament de la pàgina PMD
- Desplaçament de la pàgina PTE
- Desplaçament



Esquema de traducció d'una adreça lineal en l'arquitectura Intel(R) x86

La funció *pgd_index()* proporciona, a partir d'una estructura *mm_struct* donada, el punter base sobre el qual s'aplicaran els desplaçaments per tal d'obtindre l'adreça física.

Les funcions donades per tal d'extreure els diferents desplaçaments d'una adreça linear són les següents:

- *pgd_offset()*
- *pmd_offset()*
- *pte_offset()*

Amb l'arquitectura x86_64 varia lleugerament les taules de pàgina, passant a tindre un total de quatre taules. Apareix una nova, anomenada 'Page Upper Directory'.

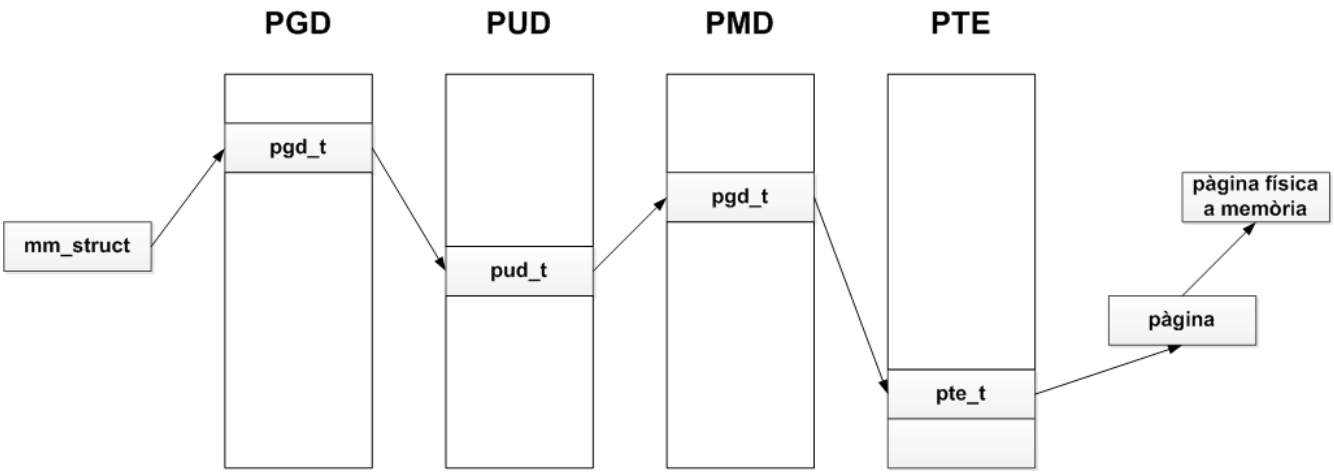
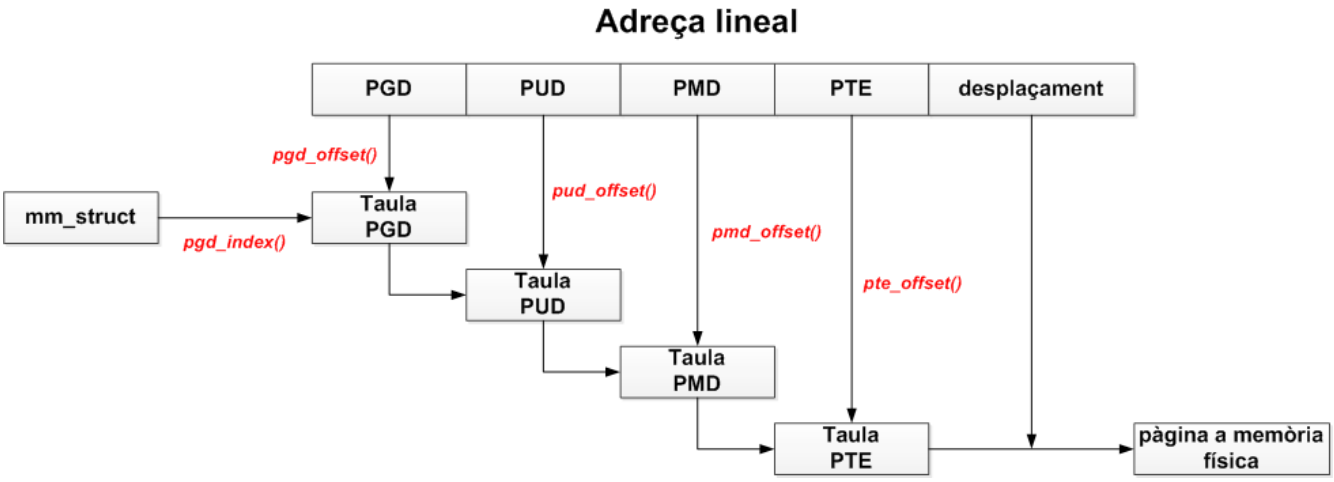


Diagrama de les taules de pàgina en l'arquitectura Intel(R) x86_64

L'esquema de traducció per l'adreça lineal també es veu modificat per aquesta nova taula.



Esquema de traducció d'una adreça lineal en l'arquitectura Intel(R) x86_64

L'adreça lineal passa a tindre cinc camps, afegint el camp per la nova taula. La funció que dóna el desplaçament a la taula afegida és la següent:

- *pud_offset()*

3.7. TLB

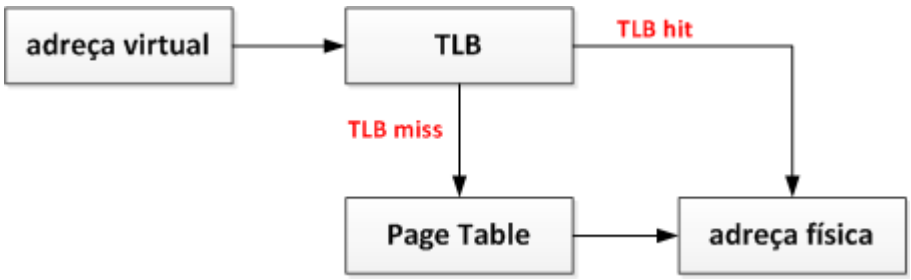
El TLB[1] és una memòria caché que tenen els processadors moderns i que fa servir el sistema de gestió de memòria per tal d'augmentar el rendiment i velocitat a l'hora de fer traduccions d'adreces virtuals a físiques.

El TLB s'encarrega de tindre una relació entre adreces físiques i virtuals. Quan el processador fa una petició d'accés a memòria, primer consulta al TLB. Si aquest té l'adreça demanada pel processador, la resposta és molt ràpida, ja que no ha de consultar les taules de pàgines (*TLB hit*). Pel contrari, si l'adreça demanada no està present al TLB, s'han de consultar les taules de pàgines[2] (*'TLB miss'*).

El fet que els processadors tinguin diferents nivells de memòries caché fa que el TLB no estigui implementat amb una única taula, sinó que és habitual fer servir varis TLB's, cadascun associat a una memòria caché corresponent.

La gestió del TLB pot ser gestionada per hardware (s'encarrega el propi processador) o per software (el sistema operatiu qui s'encarrega).

En cas que la gestió sigui per hardware, quan es produeix una fallada a l'hora de cercar l'adreça al TLB (TLB miss) és el propi processador qui s'encarrega d'anar a les taules de pàgina, fent servir el registre CR3 [3]. En canvi, si la gestió la fa el propi sistema operatiu, és aquest qui s'encarregarà d'anar a les taules de pàgina a buscar l'adreça.



Petició d'adreça al TLB

En sistemes multiprocessador, cada processador té el seu propi TLB, anomenat TLB local de la CPU[4]. No cal fer una sincronització entre els diferents TLB's de cada CPU.

---

[1] Acrònim de *Translation Lookaside Buffer*
[2] Aquest procés de consulta a les taules de pàgina se'l coneix com a *Page Walk*.
[3] Més informació sobre el registre CR3: http://en.wikipedia.org/wiki/Control_register
[4] Acrònim de *Control Process Unit*.

4. Anàlisi de requeriments

4.1. Descripció de l'eina Pin

L'eina Pin[1] és una eina desenvolupada i mantinguda per Intel(R) que fa servir la tècnica de la instrumentació dinàmica per extreure informació d'un programa durant la seva execució, afegint codi extra.

Els orígens d'aquesta eina van ser l'anàlisi i estudis de diferents arquitectures, però la seva popularitat i el fet de tindre una comunitat d'usuaris ben àmplia (coneguts com *PinHeads*[2]) han fet que apareguin altres utilitats, com la seguretat, l'emulació o l'anàlisi de programació paral·lela.

L'eina Pin es distribueix de forma gratuïta per a ús no comercial i està disponible per diferents arquitectures (Intel x86 , Intel x86_64 i IA64) i sistemes operatius (Windows , Linux i MacOS).

La instrumentació dinàmica aporta la gran avantatge de no tindre que modificar el codi del programa original ni tenir que construir-lo de nou. A més a més, permet fer una instrumentació del procés d'execució, veure que fa el codi dinàmicament o, fins i tot, manipular-lo.

La comanda que s'executa per instrumentar una aplicació és la següent:

pin -t pintool -- aplicació

Pintool és l'eina feta servir per la instrumentació i que es desenvolupa en C/C++. Es pot desenvolupar una de nova, modificar una existent o fer servir les diferents eines que hi ha disponibles amb el paquet. És habitual treballar amb una pintool ja definida i modificar-la per tal d'adaptar-la a les nostres necessitats.

Pin, en certa manera, és un compilador Just In Time[3], encara que difereix en el fet que com entrada rep un executable i no bytecode, com seria habitual.

L'API proporcionada per Pin per desenvolupar Pintools és molt àmplia. Les rutines d'instrumentació defineixen on s'insereix la instrumentació. Les accions definides quan s'activa la instrumentació es defineixen amb les rutines d'anàlisi.

---

[1] Plana web de l'eina Pin: http://www.pintool.org/
[2] Comunitat d'usuaris de Pin: http://tech.groups.yahoo.com/group/pinheads/
[3] Compilació Just In Time: http://en.wikipedia.org/wiki/Just-in-time_compilation/

Les Pintools s'executen en el mateix espai d'adreces que l'eina Pin i que l'aplicació que s'instrumenta. Aquest fet fa que l'eina tingui accés a totes les dades de l'aplicació que està analitzant. També compateix els descriptors d'arxius, informació addicional del procés i l'accés a les llibreries compartides de l'aplicació.

4.2. Descripció del pintool

Per al procés d'instrumentació amb l'eina Pin s'ha fet servin un pintool ja implementat en un projecte anterior, més concretament el que va implementar en Miguel Martínez Ferrer per al seu projecte.

Aquest pintool treu informació sobre l'accés a memòria per a aplicacions multifil. La informació obtinguda durant el procés d'instrumentació queda enregistrada en un fitxer, on cada línia representa cadascun dels accessos a memòria i les diferents columnes representen les diferents opcions escollides.

Algunes de les opcions que permet el pintool són les següents:

- Número d'instrucció.
- Número d'accés a memòria.
- Número d'instrucció del fil.
- Número d'accés a memòria del fil.
- Adreça virtual d'accés a memòria.
- Número de pàgina.
- Adreça física.

A les diferents opcions ja implementades al pintool se li afegirà dues noves. La primera d'ella permetrà fer una traducció d'adreces virtuals a físiques fent servir el mòdul desenvolupat. L'altra opció permetrà fer una traducció de les adreces virtuals amb els dos mètodes (Pagemap i mòdul) i fer una comparació de les adreces físiques obtingudes amb els dos mètodes.

4.3. Descripció del mòdul

L'objectiu principal és el desenvolupament d'un mòdul per al sistema Linux que permeti interactuar entre el nucli del sistema i una aplicació en espai d'usuari. En aquest cas, l'aplicació utilitzada serà un pintool per fer instrumentació amb l'eina Pin.

Durant el procés d'instrumentació, el pintool enviarà al mòdul el PID del procés amb el qual treballa i anirà fent consultes al mòdul per tal de traduir les adreces virtuals durant el procés d'instrumentació.

Per al procés de traducció d'adreces, el mòdul farà servir la tècnica coneguda con "*Page Walk*" i que consisteix en fer un accés de consulta a les taules de pàgina d'un procés determinat. Recordem que cada procés té definides les seves pròpies estructures.

Per tant, l'única tasca que realitza el mòdul és aquesta traducció d'adreces virtuals a físiques, per a un procés concret, a partir del PID d'aquest.

4.4. Entorn de desenvolupament

Donat que estem treballant amb un mòdul per al sistema Linux, és imprescindible fer servir un maquinari amb aquest sistema operatiu.

En principi, qualsevol distribució Linux seria suficient per tal de poder carregar el mòdul i fer servir l'eina Pin per fer instrumentació. L'únic requisit és que la distribució escollida faci servir un nucli de la rama 2.6, ja que el procés de traducció d'adreces amb les taules de pàgina ha sigut desenvolupat específicament per aquesta ramificació del nucli. El procés és lleugerament diferent per a nuclis de la ramificació 2.4.

En aquest cas concret, s'ha decidit fer tot el desenvolupament del mòdul i el procés d'instrumentació sobre un maquinari amb arquitectura Intel(R) x86 i la darrera versió estable de la distribució Debian Linux[1] del moment, més concretament la versió 6.0 coneguda amb el nom de Squeeze[2].

Aquesta distribució té com a avantatge la possibilitat d'instal·lar el programari necessari per fer la compilació i construcció de mòduls. Només és necessari fer la instal·lació d'un meta-paquet, anomenat build-essential, que conté el programari bàsic necessari per desenvolupar mòduls.

A més a més d'aquest petit conjunt de programari, és necessari fer la instal·lació de les capçaleres per al nucli, conegudes com kernel headers. Sota el sistema Debian, el paquet que conté aquestes capçaleres s'anomena "linux-headers-2.6.32-5-common".

Per al procés d'instrumentació és necessària l'eina Pin, que es distribueix de forma gratuïta des de la seva pròpia plana web. L'eina ve en un paquet comprimit que s'ha de descomprimir allà on volem fer la instal·lació.

La versió de l'eina Pin que s'ha fet servir durant aquest projecte ha sigut la darrera versió d'aquesta per al sistema operatiu Linux amb arquitectura Intel(R) x86. Més concretament, la revisió 39599 de Pin.

Com a darrer requeriment, la llibreria zlib[3], que fa servir el pintool. Aquesta llibreria es distribueix en un paquet comprimit que cal compilar, construir e instal·lar al sistema per tal que pugui fer usada pel pintool durant el procés d'instrumentació.

[1] Plana web del projecte Debian Linux: http://www.debian.org
[2] Més informació sobre la versió 6.0 de Debian: http://www.debian.org/News/2011/20110205a
[3] Plana web de la llibreria zlib: http://www.zlib.net/

5. Disseny

5.1. Disseny del mòdul

Els mòduls per al sistema Linux són desenvolupats en llenguatge C.

Per tal de poder fer la traducció d'una adreça virtual a física es farà servir la tècnica anomenada 'Page Walk'. Aquesta tècnica consisteix en consultar les taules de pàgina (Page Tables) per tal de poder obtindre la pàgina corresponent per una determinada adreça virtual.

En cas que hi hagi algun problema amb la traducció de l'adreça virtual fent 'Page Walk' (l'adreça virtual no existeix per al procés donat, per exemple) es genera una excepció i es guarda un registre al log del nucli.

Una vegada obtinguda la pàgina corresponent, només serà necessari obtindre l'adreça física per aquesta pàgina i el identificador del marc, conegut amb el nom de 'Page Frame Number'. El sistema Linux proporciona una macro per realitzar aquestes dues tasques.

És important destacar que cada procés té el seu espai d'adreces virtuals, d'aquí el fet que el mòdul rebi tant l'adreça virtual a traduir com el PID del procés.

De fet, per tal de poder fer 'Page Walk', necessitem l'estructura mm_struct del procés i l'adreça virtual a traduir.

Una altre punt important és la comunicació necessària entre el mòdul i el programa. Aquesta problemàtica s'ha solucionat fent servir un mecanisme que proporcionen els propis mòduls i que s'anomena IOCTL's[1].

Les IOCTL's és un mecanisme per cridar funcions d'un mòdul des d'un programa en espai d'usuari. Gràcies a aquesta tècnica, és possible que el programa envii al mòdul l'adreça virtual i el PID del procés i rebi d'aquest tant la traducció d'aquesta adreça com altres dades addicionals del procés.

Donat que s'han desenvolupat el mòdul per dues arquitectures diferents, és important fer notar que el procés de consulta a les taules de pàgina varia lleugerament entre totes dues. Més endavant es veurà en detall el procés el procés de traducció per cada arquitectura.

[1] De l'anglès, *Input/Output Control.* Més informació: http://en.wikipedia.org/wiki/Ioctl

5.2. Disseny del pintool

Donat que la finalitat del mòdul és fer-lo servir durant el procés d'instrumentació amb l'eina Pin, s'ha fet servir un pintool ja implementat al qual se li ha fet un sèrie de modificacions per tal que faci servir el mòdul per traduir les adreces virtuals a físiques.

El pintool utilitzat es va utilitzar per fer l'anàlisi dels accessos a memòria amb aplicacions amb múltiples fils d'execució. Aquest pintool va ser implementat per Miguel Martínez Ferrer per al seu projecte final.

El pintool original feia servir com a traductor d'adreces virtuals consultes a pagemap[1]. Aquesta característica, que permet la traducció d'adreces des d'espai d'usuari, es més una tècnica més recent que va ser incorporada al nucli de Linux des de la versió 2.6.25. En cas de tindre nuclis anteriors no seria vàlida, mentre que la traducció amb el mòdul desenvolupat seria vàlida per qualsevol versió del nucli 2.6.

L'idea principal ha sigut afegir un nou cas d'ús per tal que el pintool faci servir com a motor de traducció d'adreces el mòdul desenvolupat durant aquest projecte. A més a més, s'afegeix un cas addicional per tal de comparar els dos mètodes de traducció d'adreces virtuals.

D'aquesta manera, es dóna la possibilitat de fer servir el mètode de traducció que ja tenia implementat el pintool original o escollir la traducció amb crides al mòdul.

Aquestes modificacions només afecten a la part dedicada a la traducció d'adreça virtual a física. L'accés i tancament al mòdul només es farà quan es crida el pintool amb aquest nou cas d'ús, de la mateixa manera que les dues crides IOCTL es faran únicament quan es treballi amb aquest cas d'ús.

S'ha decidit que aquest cas d'ús sigui opcional. Per defecte el pintool treballarà com originalment estava desenvolupat. Per tal d'activar l'ús de la traducció d'adreces amb el mòdul es tindrà que executar aquest amb el paràmetre corresponent.

Durant el mètode de comparació de tots dos mètodes, es buscarà trobar el 'Page Frame Number' per a una adreça virtual fent servir els dos mètodes: Pagemap i mòdul.

[1] Més informació sobre Pagemap: http://www.kernel.org/doc/Documentation/vm/pagemap.txt

6. Implementació

6.1. Descripció codi font del mòdul

El mòdul consta de dos arxius:

- module.c
  Fitxers que conté la declaració del mòdul i les dues funcions IOCTL per interactuar amb aquest.

- module.h
  Fitxer de capçaleres (*header file*[1]).

Es pot consultar el codi font dels dos arxius als annexos d'aquesta memòria.

El mòdul es distribueix en un fitxer de tipus tar.gz que conté aquests dos arxius i un makefile, per facilitar la tasca de compilació del mòdul. Només cal cridar aquest fitxer makefile per compilar i construir el mòdul.

6.2. Fitxer de capçaleres

Del fitxer de capçaleres només comentar que conté la declaració del nom del mòdul (DEVICE_FILE_NAME), l'identificador numèric d'aquest (MAJOR_NUM) i les dues crides IOCTL per interactuar amb aquest:

- IOCTL_ENVIAR_PID
  Enviament del PID del procés. Aquest valor és de tipus enter (*int*).
  Retorna error (valor negatiu) si ha hagut algun problema de comunicació amb el mòdul.

- IOCTL_TRADUCTOR
  Envia al mòdul l'adreça virtual que volem traduir a física. Rep aquesta adreça física.
  El format, tant per l'adreça virtual com física, és *unsigned long*.
  En cas que no existeixi l'adreça virtual, rebrà com a resposta el valor 0.

- IOCTL_REBRE_PFN
  Rep del mòdul el Page Frame Number associat a l'adreça física trobada amb l'anterior crida.

6.3. Fitxer declaració del mòdul

Com és habitual en els fitxers del llenguatge C, la primera part del fitxer està dedicada als includes per poder fer servir les diferents biblioteques que el mòdul farà servir. Entre aquestes, només destacar aquelles necessàries per els mòduls, per poder treballar amb les IOCTLs i per poder consultar les taules de pàgina.

Seguidament, tindrem un parell de constants (#define), un per la retornar OK quan no ha hagut erros i l'altre ens defineix el nom del mòdul.

A continuació tenim l'espai per a la declaració de les diferents variables globals necessàries. La primera d'aquestes variables es fa servir per controlar l'accés concurrent al mòdul. Aquest mòdul està desenvolupat amb la idea que no tindrà accessos simultanis, es a dir, només serà accedit per un única aplicació a la vegada.

La resta de variables globals són les necessàries tant per emmagatzemar el PID del procés enviat des de l'espai d'usuari (*userland*) com per cercar el procés i poder fer 'Page Walk' consultant les taules de pàgina.

Totes aquestes variables són de tipus punter. Per evitar problemes de segmentació (*Segmentation Fault*) s'inicialitza la variable PID, per així garantir que conté un valor.

Tenim dues estructures definides:

- mm_struct
- page

La primera d'aquestes estructures és necessària per durant el procés de consulta a les taules de pàgina, per tal de poder fer la traducció de l'adreça virtual a física. La segona estructura serà la pàgina obtinguda després de consultar les taules de pàgina.

El mòdul defineix un total de tres operacions:

- device_open
- device_release
- device_ioctl

Les operacions de lectura i escriptura (conegudes com *read* i *write*, respectivament) no han sigut implementades i tenen el valor nul (NULL).

L'operació d'obertura (device_open) es cridada des de l'espai d'usuari quan un procés vol obrir l'accés a aquest. En cas que el mòdul estigui lliure i accessible, serà permès l'accés i el mòdul passarà a estar ocupat, per així evitar accessos concurrents. Si el mòdul ja està obert per un altre procés, retornarà una excepció (BUSY).

L'operació de tancament (device_release) farà que el mòdul deixi d'estar ocupat per tal que pugui ser accedit per un altre procés.

Dins de l'operació IOCTL (device_ioctl) es defineixen les tres crides que el mòdul té definides per habilitar la comunicació entre els processos a espai d'usuari i el mòdul en espai del nucli.

Les dues crides definides són les següents:

- IOCTL_ENVIAR_PID
- IOCTL_TRADUCTOR
- IOCTL_REBRE_PFN

La crida IOCTL_ENVIAR_PID s'encarrega de passar al mòdul el valor del PID del procés amb el qual es vol treballar. És important tindre el PID del procés ja que cada cadascun dels processos del sistema defineix el seu propi espai d'adreces virtuals.

Amb aquest PID cercarem la seva estructura *mm_struct* dins del conjunt de processos en execució al sistema. Aquesta estructura és necessària per poder iniciar una cerca amb les taules de pàgina.

Aquesta crida només és de lectura (tipus _IOR) i retornarà un error (valor negatiu) si hi ha hagut algun problema de comunicació amb el mòdul.

La segona crida definida, anomenada IOCTL_TRADUCTOR, és l'encarregada de fer la traducció d'una adreça virtual a física, fent 'Page Walk'. Aquesta crida és de lectura/escriptura (tipus _IOWR), ja que ens donarà com a retorn l'adreça física corresponent.

Per últim, la darrera de les crides, anomenada IOCTL_REBRE_PFN, només s'encarrega d'enviar a l'espai d'usuari (userland) el 'Page Frame Number' trobat durant el procés de traducció (crida IOCTL_TRADUCTOR). És important destacar que aquesta no s'ha d'executar de forma independent, ja que va associada a la crida anterior. S'ha definit aquest mètode per evitar traduir per partida doble.

6.4. Consulta a les taules de pàgina - x86

Tal i com ja s'ha vist anteriorment, cada procés defineix les seves pròpies taules de pàgina (més conegudes amb el nom anglès Page Tables) que fa servir per traduir les adreces virtuals que fa servir a físiques. Recordem que encara que cada procés treballi amb el seu conjunt d'adreces virtuals, finalment sempre necessitarà una adreça física per poder accedir a la memòria del sistema.

El procés de traducció d'adreces virtuals a físiques fa servir aquestes taules per tal que, donada una adreça virtual i l'estructura mm_struct del procés, obtindre l'adreça física equivalent.

En l'arquitectura Intel(R) x86 tenim un total de tres taules, com ja s'ha vist anteriorment.

La primera consulta es fa a la primera taula, anomenada Page Global Directory (PGD), fent servir la funció *pgd_offset*. Aquesta funció rep dos paràmetres, l'estructura *mm_struct* del procés i l'adreça virtual que volem traduir.

En cas que el procés hagi sigut correcte, donarà un punter de tipus *pgd_t* necessari per la consulta a la segona taula, coneguda com Page Middle Directory (PMD). Aquesta segona consulta es realitza fent servir la funció *pmd_offset*, que rep per paràmetre el punter *pgd_t* i l'adreça virtual a traduir.

De la mateixa manera que amb la consulta anterior, si el procés ha sigut correcte, tindrem un nou punter de tipus *pmd_t* necessari per la consulta a la darrera taula, anomenada Page Table (PTE).

La consulta a aquesta darrera taula es fa amb la funció *pte_offset_map*, amb el punter de tipus *pmd_t* obtingut anteriorment i l'adreça virtual.

Finalment, i una vegada obtingut el valor *pte*, només cal obtindre la pàgina associada. Aquesta tasca es soluciona fent servir una macro que el propi sistema proporciona, anomenada *pte_page*, que retorna la pàgina associada al valor *pte* passat per paràmetre.

Arribats a aquest punt, tenim la pàgina associada a l'adreça virtual, però encara no tenim el que realment estem cercant, es a dir, l'adreça física. El procés d'obtindre l'adreça física d'una pàgina es resol fent servir una altra macro del sistema, *page_to_phys*.

A més a més de convertir la pàgina a la seva adreça física corresponent, s'ha de trobar el 'Page Frame Number' d'aquesta pàgina. Aquesta tasca es soluciona fent servir una altra macro, anomenada *page_to_pfn*.

Durant el procés de consulta a les diferents taules es comprova que els punters no tinguin un valor nul o incorrecte. Per fer aquesta tasca es fan servir una sèrie de macros que les pròpies taules de pàgina tenen definides:

- *pgd_none* i *pgd_bad*
- *pmd_none* i *pmd_bad*

En cas que donin un resultat nul o incorrecte, no es continua amb el procés 'Page Walk' i guarda una excepció al log del nucli.

És important destacar la tasca de la macro *pte_unmap,* que ha de ser cridada tan aviat com es tingui el valor del punter a la darrera taula. Aquesta macro s'encarrega de desmapejar el punter a la taula PTE i així evitar errors de segmentació (*Segmentation Fault*) quan es fan crides al mòdul per fer traduccions d'adreces.

6.5. Consulta a les taules de pàgina - x86_64

El procés de consulta a les taules de pàgina varia lleugerament en l'arquitectura x86_64.

En aquesta arquitectura tenim un total de quatre taules. Les mateixes taules que en l'arquitectura x86 i una addicional, anomenada Page Upper Directory (PUD). Aquesta es troba entre la taula Page Global Directory i Page Middle Directory.

La consulta a aquesta taula es fa amb la funció *pud_offset* i el punter *pgd_t* obtingut al consulta a la taula Page Global Directory.

Per tant, en aquesta arquitectura el procés de 'Page Walk' segueix el següent ordre de consulta:

- Page Global Directory: amb la funció *pgd_offset*.
- Page Upper Directory: amb la funció *pud_offset*.
- Page Middle Directory: amb la funció *pmd_offset*.
- Page Table: amb la funció *pte_offset_map*.

De la mateixa manera que en l'arquitectura x86, es comprova que els punters no tinguin un valor nul o incorrecte amb la nova taula. Les macros de control per la taula Page Upper Directory són les següents:

- *pud_none* i *pud_bad*

6.6. Pintool

El pintool es desenvolupa en un únic fitxer, anomenat mtaptrace.cpp, en llenguatge C++. La distribució d'aquest es fa en un paquet de tipus tar.gz que conté el codi font d'aquest i makefile per la seva compilació.

Per mantenir l'espai d'emmagatzemament local per cada fil d'execució es fa servir una utilitat de l'eina Pin, anomenada Thread Local Storage o TLS. Es defineixen una variable de tipus TLS_KEY i una funció per accedir a aquesta.

Pintool defineix un descriptor d'arxiu per poder accedir al pagemap i un conjunt de variables per guardar els llindars.

Un altre descriptor d'arxiu es defineix per poder accedir al fitxer de sortida i una variable de bloqueig per controlar els accessos dels diferents fils a les variables globals o tractin d'imprimir el mateix arxiu a la vegada.

El format de l'arxiu de sortida es defineix fent servir un conjunt de funcions auxiliars. Altres funcions es defineixen per extreure informació sobre el sistema de memòria, com la mida de la pàgina (habitualment 4K) o el nivell més gran de la memòria cau.

Les diferents impressions per pantalla que realitza el pintool durant la seva execució també es defineixen fent servir funcions, com la impressió de l'arxiu de log o un resum dels comptadors per fi i globals.

Finalment, es defineix una darrera funció addicional que mostra un missatge d'ajuda. Quan es crida el pintool amb l'argument -h o -help, es mostra aquest missatge per pantalla i es para l'execució.

La rutina d'instrumentació insereix una crida a la rutina d'anàlisi. Seguidament, comprova si la instrucció té accessos a memòria i per cada accés insereix una crida a les rutines d'anàlisi. Finalment, comprova si es vol avortar l'execució en arribar al llindar superior.

El pintool original traduïa les adreces virtuals amb consultes al pagemap, tal i com ja s'ha vist. Amb el nou cas d'ús afegit, les traduccions d'adreces es faran amb crides al mòdul desenvolupat.

Per tal que el pintool pugui tindre accés al mòdul, primerament, s'ha d'afegir el fitxer de capçaleres del mòdul (module.h). D'aquesta manera, el pintool ja tindrà les dues crides IOCTL definides per accedir al mòdul, així com el nom i MAJOR_NUM d'aquest.

S'han declarat un total de tres variables globals, necessàries quan el pintool fa accessos al mòdul. La primera d'aquestes, de tipus enter (int), necessària quan s'obri l'accés al mòdul per part del pintool. Les altres dues, de tipus unsigned long, es faran servir per emmagatzemar l'adreça virtual i física, respectivament.

El tractament dels arguments que es podem passar per línia de comandes al pintool es fa utilitzant una eina, anomenada Knob, que defineix el nom del paràmetre, el seu valor per defecte, i una petita descripció.

S'han afegit un parell de nous Knob's de tipus booleà al pintool:
- KnobUseModuleAddressTranslator
- KnobCompareMethods

El primer d'aquests indica al pintool que ha de traduir les adreces virtuals a físiques fent servir el mòdul desenvolupat i prèviament carregat al sistema.

El segon és un Knob addicional i lligat a l'anterior. Dóna la possibilitat de comparar el marc de pàgina (conegut com 'Page Frame Number') obtingut fent servir el dos mètodes: consultes al pagemap i consultes al mòdul. El resultat d'aquesta comparació es mostrarà per la sortida estàndard durant el procés d'instrumentació.

Tots dos Knob's s'ha decidit que tinguin per defecte un valor igual a 0, es a dir, no es cridaran. Per tal d'utilitzar-los, s'haurà d'indicar explícitament per paràmetre quan es llenci l'execució de l'eina Pin amb el pintool.

L'obertura i el tancament de l'accés al mòdul es farà únicament quan algun dels dos Knob's anteriors tinguin valor 1. Totes dues operacions es defineixen al programa principal (main) del pintool. En cas que no es pugui obrir l'accés al mòdul, s'imprimirà un missatge d'error per la sortida estàndard i s'avortarà la instrumentació.

El tancament de l'accés al mòdul s'ha definit com el darrer pas del programa principal, una vegada ja hagi quedat finalitzada la instrumentació.
Abans de poder fer una traducció, el mòdul ha d'haver rebut el PID del procés sobre el qual farà la traducció d'adreces virtuals. Per tant, cada vegada que es crei un fil, es passarà el PID del procés al mòdul. Per tal d'obtindre el PID, es fa servir la funció PIN_GetPid().

Una vegada tenim el valor del PID emmagatzemat en una variable, només és necessari fer una crida IOCTL al mòdul amb aquesta, per tal que el mòdul tingui el PID amb el qual treballarà durant la traducció.

El pintool té definit un constructor que per agafar la informació d'una adreça física, donada una virtual. És en aquest constructor on s'ha definit l'accés al mòdul per fer la traducció, sempre i quan es faci servir aquest.

Només cal fer una crida IOCTL al mòdul amb l'adreça virtual per tal d'obtindre la traducció d'aquesta. Una vegada obtinguda aquesta, només cal fer un casting per tal d'emmagatzemar-la en una variable de tipus off64_t.

En cas que d'estar treballant amb el Knob "KnobCompareMethods" es farà el procés de traducció fent servir el dos mètodes ja comentats i comparant el resultat del 'Page Frame Number' obtingut amb cadascun d'ells. Només s'imprimirà informació per la sortida estàndard en cas que no hi hagi una coincidència en les dues adreces. En cap cas s'avortarà el procés d'instrumentació.

7. Proves

7.1. Proves de comunicació amb el mòdul

Les proves inicials que es van fer amb el mòdul van ser de comunicació entre aquest i una aplicació en espai d'usuari (userland) fent servir les crides IOCTL.

Es va desenvolupar una aplicació molt senzilla en llenguatge C amb l'única finalitat de provar que la comunicació entre el mòdul ja carregat al sistema i aquesta aplicació era correcta. Amb comunicació es vol donar a entendre enviament i recepció de dades entre totes dues.

Per tal de comprovar que la recepció era correcta, es van fer servir impressions per pantalla per a l'aplicació i guardar registre al log del sistema per al mòdul. Les impressions per pantalla es poden veure en el moment d'executar l'aplicació des de la consola del sistema. Per tal de poder accedir al log del sistema, es executar la comanda 'dmesg' (no cal tindre permisos de root) o bé consultar directament l'arxiu de log ('/var/log/syslog' en sistemes Debian). Per aquesta darrera opció si cal tindre permisos de root.

Després de tota una sèrie de proves per tal de provar les crides IOCTL es va poder veure que la comunicació era correcta i l'aplicació enviava correctament un PID al mòdul. Aquesta prova va ser feta per provar la primera crida IOCTL, anomenada 'IOCTL_ENVIAR_PID'.

Per provar la segona crida, anomenada 'IOCTL_TRADUCTOR' es va provar que l'adreça virtual introduïda era correctament rebuda pel mòdul i que l'adreça física obtinguda s'enviava sense errors del mòdul a l'aplicació. Totes dues proves van ser satisfactòries. De la mateixa manera que amb la crida anterior, es va fer servir tant el log del sistema com la sortida per pantalla per comprovar que la comunicació era correcta.

Per últim, es va fer una darrera prova per la darrera crida, IOCT_REBRE_PFN, per comprovar que es rebia correctament l'identificador de pàgina per una traducció concreta. De la mateixa manera que amb les proves anteriors, la comunicació va ser correcta entre mòdul i espai d'usuari.

Les mateixes proves es van fer tant per la versió x86 com per x86_64 del mòdul.

7.2. Proves amb la traducció d'adreces

Les proves amb la traducció d'adreces no van començar a fer-se fins que es va comprovar que la comunicació entre el mòdul i l'aplicació eren correctes.

Per al procés de traducció es van fer dues proves ben diferenciades, fent servir una adreça virtual existent per al procés i una altra amb una adreça virtual no existent.

La primera de les proves ha de permetre poder consultar sense errors les taules de pàgina per tal de poder fer la traducció. Donada una adreça virtual existent per a l'espai d'adreces del procés s'ha de permetre fer una consulta a la taula de pàgines per tal de poder trobar l'adreça física corresponent.

Si durant el procés de traducció s'arriba fins a obtindre la pàgina corresponent a l'adreça virtual donada, es considera que el procés de traducció ha sigut correcte. Convertir aquesta pàgina a direcció física i obtindre el marc de pàgina és un procés trivial fent servir una macro, com ja s'ha vist.

En cas que l'adreça virtual no existeixi per al procés indicat, el proces de 'Page Walk' generarà una excepció, que quedarà enregistrada al log del sistema. Només cal, doncs, fer consultes a aquest log per tal de comprovar si el procés de traducció ha sigut correcte.

A més a més, quan el procés de traducció no és correcte, el valor de l'adreça física retornada pel mòdul té valor 0. Des de l'aplicació en espai d'usuari també s'ha comprovat aquest fet, consultant el valor de retorn obtingut després de fer una consulta de traducció al mòdul.

El procés de traducció va generar tot una sèrie de problemes degut al fet que era imprescindible un 'desmapeig' del punter a la darrera taula, amb la comanda '*pte_unmap*'. El fet de no fer aquest 'demapeig' generava una excepció (*Segmentation Fault*) durant el procés de traducció i no es podia obtindre l'adreça física des de l'espai d'usuari.

Les proves de traducció d'adreces van ser realitzades per les dues versions del mòdul. Recordem que el procés de traducció difereix lleugerament entre les dues arquitectures.

El resultat de les proves de traducció d'adreces va ser satisfactori en totes dues arquitectures.

7.3. Proves inicials amb el pintool

Després de fer les modificacions ja comentades al pintool per tal d'afegir casos d'ús vistos, es passar a provar a fer la instrumentació amb l'eina Pin fent servir com a motor de traducció d'adreces el mòdul carregat prèviament al sistema.

Donat que el mòdul genera una excepció al log del sistema, es va fer servir aquesta característica per provar que la traducció d'adreces era correcta durant el procés d'instrumentació.

La comanda feta servir per a fer la instrumentació activant l'ús del mòdul va ser la següent:
$ pin -t mtaptrace -pa 1 -mat 1 -- /bin/ls

Després de fer aquesta instrumentació es va consultar el log per tal de veure si s'havia produït alguna excepció durant el procés de traducció. Es resultat va ser negatiu, cosa que indica que totes les adreces virtuals a traduir eren correctes.

Per altra banda, el propi mòdul guarda al log una entrada cada vegada que es fa un accés a ell, amb les crides IOCTL. Després del procés d'instrumentació també es va comprovar aquest fet, sent visible que el pintool havia accedit contínuament al mòdul per fer traduccions.

D'altra banda, i per veure els accessos al mòdul durant el propi procés d'instrumentació, es van fer servir dues consoles diferents. Amb la primera, es va llençar el procés d'instrumentació de forma normal, des d'espai d'usuari amb un usuari amb permisos estàndards i, amb l'altra consola, es va carregar el log del sistema amb l'opció de refrescar aquest de forma continua, per tal de veure en temps reals els registres que s'anessin produint en ell.

La comanda per fer aquesta tasca és la següent:
$ tail -f /var/log/syslog

Per poder executar aquesta comanda, cal que l'usuari tingui permisos de root. D'aquesta manera es va comprovar, en temps real, com el pintool anava accedint al mòdul per fer traduccions durant tot el procés d'instrumentació, sense generar cap error.

Aquest primer conjunt de proves va ser fet amb les dues versions del mòdul, x86 i x86_64.

El resultat de les proves inicials ha sigut satisfactori en les dues arquitectures.

7.4. Proves amb NAS Parallel Benchmarks

Donat que el funcionament de la traducció d'adreces proporcionada amb el mòdul era correcte, es va passar a fer unes proves de rendiment per tal de veure la millora en el rendiment de la traducció en comparació amb el mètode Pagemap ja implementat.

Per fer aquestes proves de rendiment s'ha fet servir una eina desenvolupada per la NASA per avaluar el rendiment en sistemes HPC[1]. Aquesta eina són una sèrie de tests de càlcul intensiu, anomenats NAS Parallel Benchmarks[2][3].

La versió dels benchmarks feta servir ha sigut la 3.3.1, amb els tipus OpenMP i Serial Versions dels tests. Tots els tests amb classe B[3].

El pintool fet servir incorpora la possibilitat de limitar la instrumentació a un número determinat d'instruccions. S'ha fet servir aquesta característica per executar els diferents tests amb un determinat nombre d'instruccions, més concretament amb 100.000, 1.000.000 i 10.000.000 d'instruccions, respectivament.

Per altra banda, el fitxer resultant que genera el procés d'instrumentació registra amb unes marques de temps (anomenades 'timestamps') el moment d'execució de cada instrucció. D'aquesta manera és senzill saber el temps trigat en instrumentar un determinat número d'instruccions, només cal restar el timestamp de la darrera i la primera instrucció.

Aquestes tests d'estrès s'han fet sobre una arquitectura Intel(R) x86_64 amb aquesta versió del mòdul. Aquests tests de rendiment no han sigut provats amb la versió x86.

Característiques del maquinari fet servir per aquest conjunt de proves de rendiment:

- Processador: Intel(R) Xeon E5410 2,33GHz. - 4 nuclis
- Memòria RAM: 4GB.
- Distribució Linux: Fedora 14
- Versió del nucli Linux: 2.6.35

Aquest maquinari ha sigut proporcionat per la Universitat Oberta de Catalunya.

---

[1] Més informació sobre sistemes HPC: http://en.wikipedia.org/wiki/High-performance_computing
[2] Plana web oficial: http://www.nas.nasa.gov/Resources/Software/npb.html
[3] Informació detallada sobre els diferents tests: http://en.wikipedia.org/wiki/NAS_Parallel_Benchmarks

7.4.1. Instrumentació amb 100.000 instruccions

Benchmarks amb test tipus OpenMP

Benchmark Type	Class	# instructions	Pagemap	Module	% Improvement	Pagemap / Module
BT	B	100,000	1176030	946740	19.4969516084	1.2421889854
CG	B	100,000	1149800	947240	17.6169768655	1.2138423208
DC	B	100,000	1178390	952880	19.137127776	1.2366614894
EP	B	100,000	1172600	946450	19.2862016033	1.2389455333
FT	B	100,000	1160160	945110	18.5362363812	1.2275396515
IS	B	100,000	1183760	954100	19.4008920727	1.2407085211
LU	B	100,000	1181500	947300	19.8222598392	1.2472289665
MG	B	100,000	1159860	947830	18.2806545618	1.2237004526
SP	B	100,000	1150120	947420	17.6242479046	1.2139494628

Benchmarks amb test tipus Serial Version

Benchmark Type	Class	# instructions	Pagemap	Module	% Improvement	Pagemap / Module
BT	B	100,000	1113500	902430	18.955545577	1.2338907173
CG	B	100,000	1103070	901090	18.3107146419	1.2241507508
DC	B	100,000	1232400	1034050	16.0946121389	1.1918185774
EP	B	100,000	1107060	902450	18.4822864163	1.2267272425
FT	B	100,000	1125610	910230	19.1345137303	1.2366215132
IS	B	100,000	1241010	1031960	16.8451503211	1.2025756812
LU	B	100,000	1119310	901600	19.4503756779	1.2414707187
MG	B	100,000	1119010	901690	19.4207379737	1.2410140958
SP	B	100,000	1118030	901520	19.3653122009	1.2401610613

Per cada taula, es pot veure el test I classe d'aquest. NAS Parallel Bechmarks són un total de nou tests. S'han executat tots.

Les columnes 'Pagemap' I 'Module' mostrem el temps trigat en l'execució. Aquest temps es mostra en microsegons (10^{-6} segons).

La columna '% Improvement' indica el percentatge de millora en els temps comparant el mòdul amb el procés de Pagemap.

La millora obtinguda per aquest conjunt d'instruccions és d'un 18% menys de temps (de mitjana).

7.4.2. Instrumentació amb 1.000.000 instruccions

Benchmarks amb test tipus OpenMP

Benchmark Type	Class	# instructions	Pagemap	Module	% Improvement	Pagemap / Module
BT	B	1,000,000	7553240	6553240	13.2393515895	1.152596273
CG	B	1,000,000	7789360	6789360	12.8380252036	1.1472892879
DC	B	1,000,000	5408120	4408120	18.4907139634	1.2268540784
EP	B	1,000,000	10650100	9650100	9.3895831964	1.1036258692
FT	B	1,000,000	7954730	6954730	12.5711369211	1.1437870341
IS	B	1,000,000	5752660	4752660	17.3832626993	1.210408487
LU	B	1,000,000	8047590	7047590	12.4260803545	1.1418924767
MG	B	1,000,000	5687270	4687270	17.5831286364	1.2133438014
SP	B	1,000,000	11018010	10018010	9.0760491232	1.0998202238

Benchmarks amb test tipus Serial Version

Benchmark Type	Class	# instructions	Pagemap	Module	% Improvement	Pagemap / Module
BT	B	1,000,000	6324670	4301900	31.9822219974	1.4702038634
CG	B	1,000,000	5059540	3608690	28.6755317677	1.4020434008
DC	B	1,000,000	5032070	3599560	28.4676087574	1.397968085
EP	B	1,000,000	4101030	3088120	24.6989171013	1.3280021502
FT	B	1,000,000	3495030	2672150	23.5442900347	1.3079467844
IS	B	1,000,000	4490730	2950310	34.3022181249	1.5221214042
LU	B	1,000,000	6978530	4377220	37.275901945	1.5942835864
MG	B	1,000,000	4577790	3166130	30.8371506775	1.4458629305
SP	B	1,000,000	8285900	5279190	36.2870659796	1.5695400241

Per cada taula, es pot veure el test I classe d'aquest. NAS Parallel Bechmarks són un total de nou tests. S'han executat tots.

Les columnes 'Pagemap' I 'Module' mostrem el temps trigat en l'execució. Aquest temps es mostra en microsegons (10^{-6} segons).

La columna '% Improvement' indica el percentatge de millora en els temps comparant el mòdul amb el procés de Pagemap.

En aquest cas, la millora de temps és d'un 13% per a la versió OpenMP i d'un 30% amb Serial Version, de mitjana global per tots els tests.

7.4.3. Instrumentació amb 10.000.000 instruccions

Benchmarks amb test tipus OpenMP

Benchmark Type	Class	# instructions	Pagemap	Module	% Improvement	Pagemap / Module
BT	B	10,000,000	53437910	32176200	39.78769005	1.6607899628
CG	B	10,000,000	67037860	35592800	46.9064197455	1.8834668809
DC	B	10,000,000	43127590	27399390	36.4689981518	1.5740346774
EP	B	10,000,000	43550530	23072000	47.0224587393	1.8875923197
FT	B	10,000,000	66620170	39283960	41.0329334194	1.6958618734
IS	B	10,000,000	39220090	22098150	43.6560446445	1.7748132762
LU	B	10,000,000	67420180	34682350	48.5579095161	1.9439334416
MG	B	10,000,000	31852580	19691010	38.180800425	1.6176204268
SP	B	10,000,000	77985800	45571120	41.5648489853	1.7112987348

Benchmarks amb test tipus Serial Version

Benchmark Type	Class	# instructions	Pagemap	Module	% Improvement	Pagemap / Module
BT	B	10,000,000	47537660	27219350	42.7415022111	1.7464656577
CG	B	10,000,000	32067730	20457940	36.2039657936	1.5674955543
DC	B	10,000,000	43904420	27186150	38.0787856895	1.6149554093
EP	B	10,000,000	24651050	16621490	32.5728924326	1.4830830449
FT	B	10,000,000	12446690	8253370	33.6902421447	1.5080736717
IS	B	10,000,000	32388030	18870410	41.7364686892	1.7163394966
LU	B	10,000,000	58232730	30787470	47.1302994038	1.8914425252
MG	B	10,000,000	26780980	15492800	42.1499885366	1.7286081276
SP	B	10,000,000	71016440	40214310	43.3732386473	1.7659494842

Per cada taula, es pot veure el test I classe d'aquest. NAS Parallel Bechmarks són un total de nou tests. S'han executat tots.

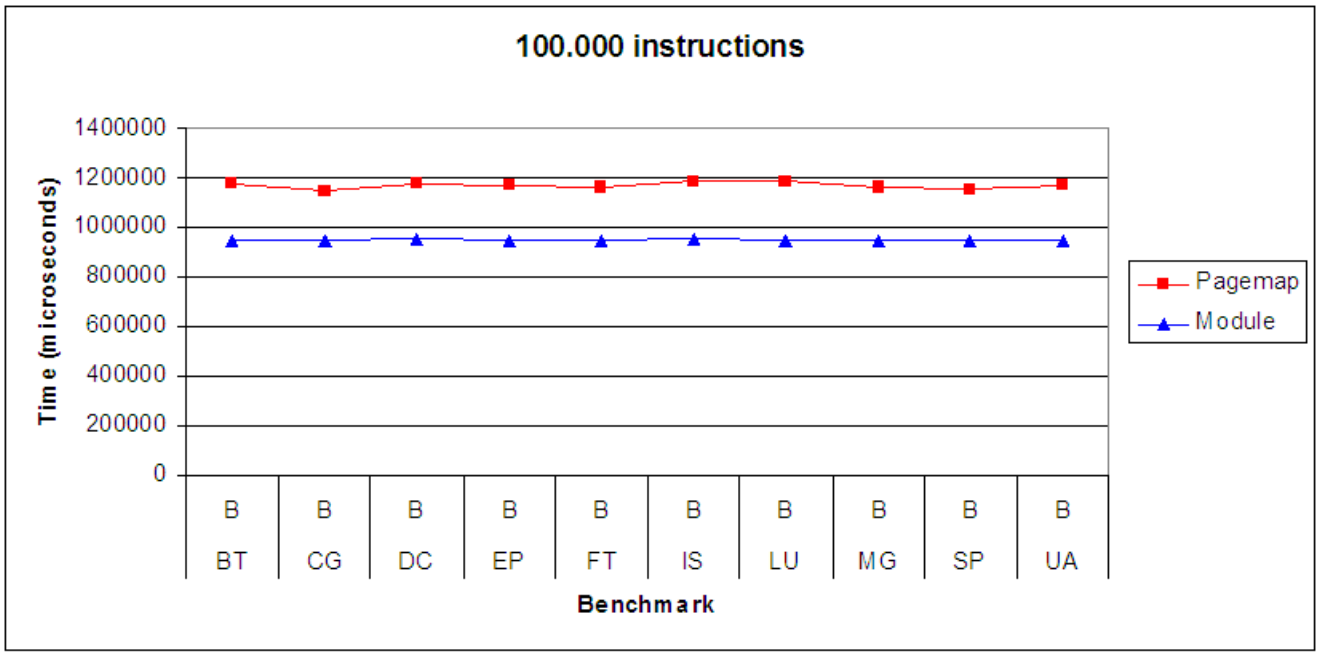
Les columnes 'Pagemap' I 'Module' mostrem el temps trigat en l'execució. Aquest temps es mostra en microsegons (10^{-6} segons).

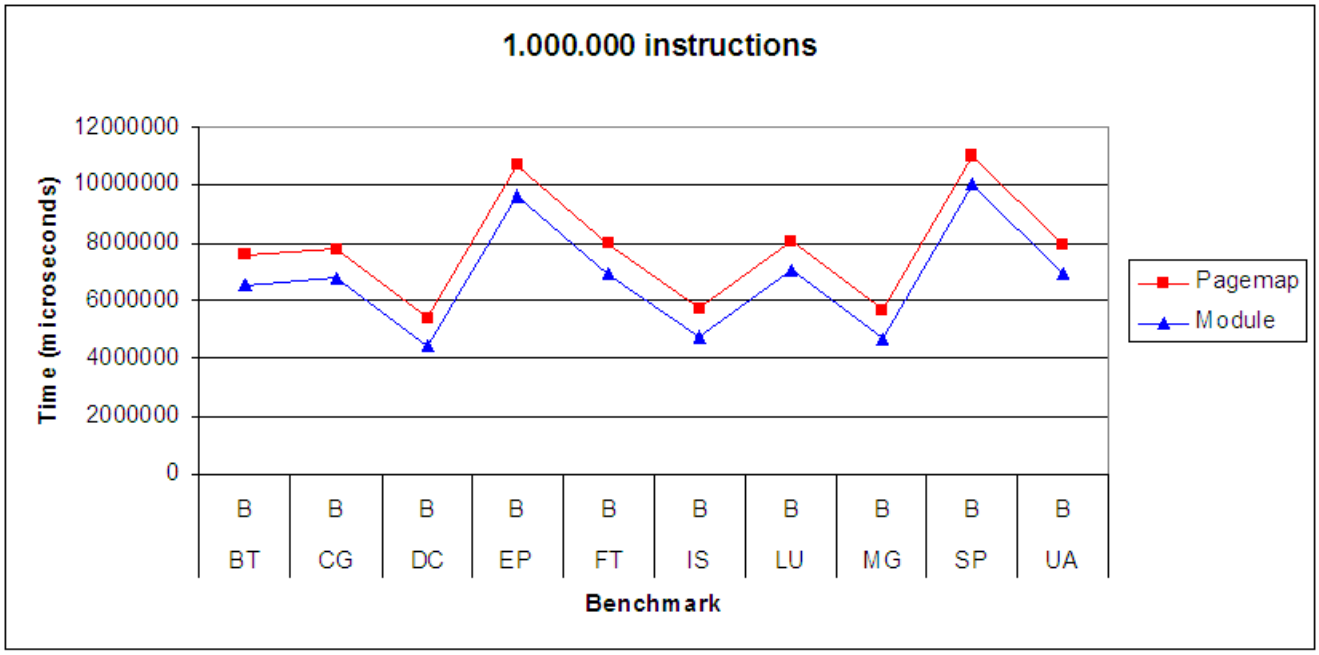
La columna '% Improvement' indica el percentatge de millora en els temps comparant el mòdul amb el procés de Pagemap.

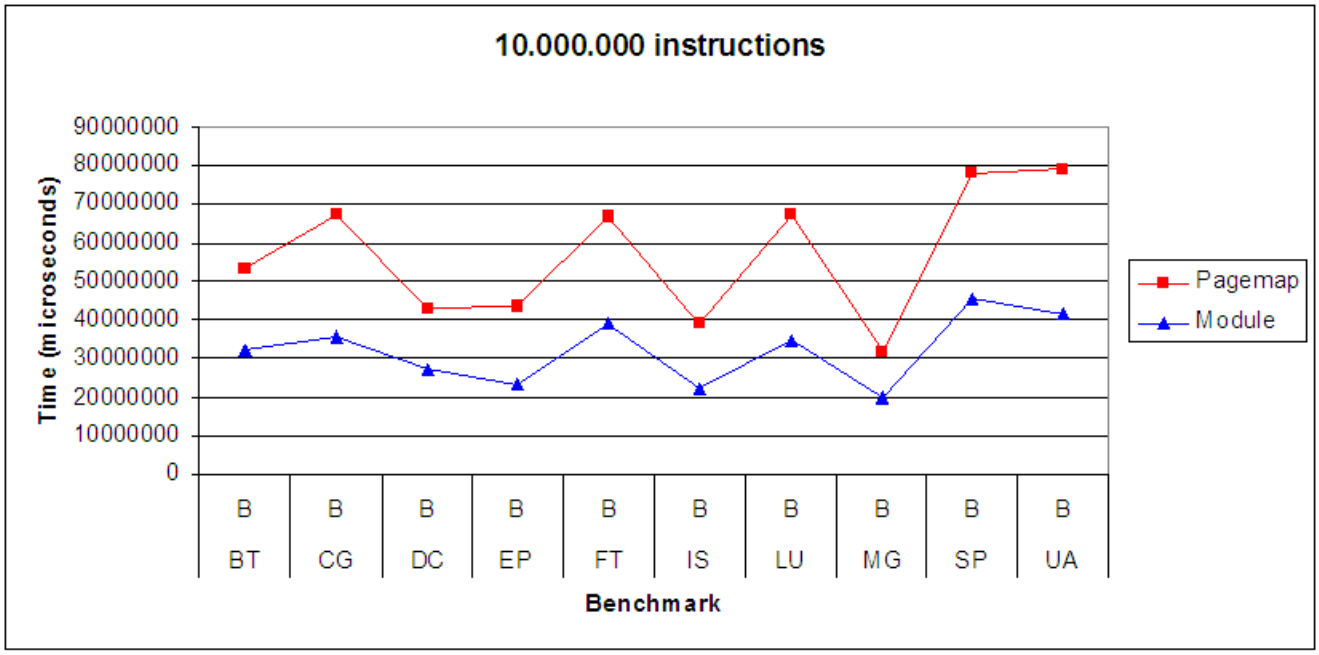
La millora global és, en aquest cas, d'un 40% de mitjana per al conjunt de tests en tots dos tipus (OpenMP i Serial Version).

7.4.4. Gràfiques comparatives

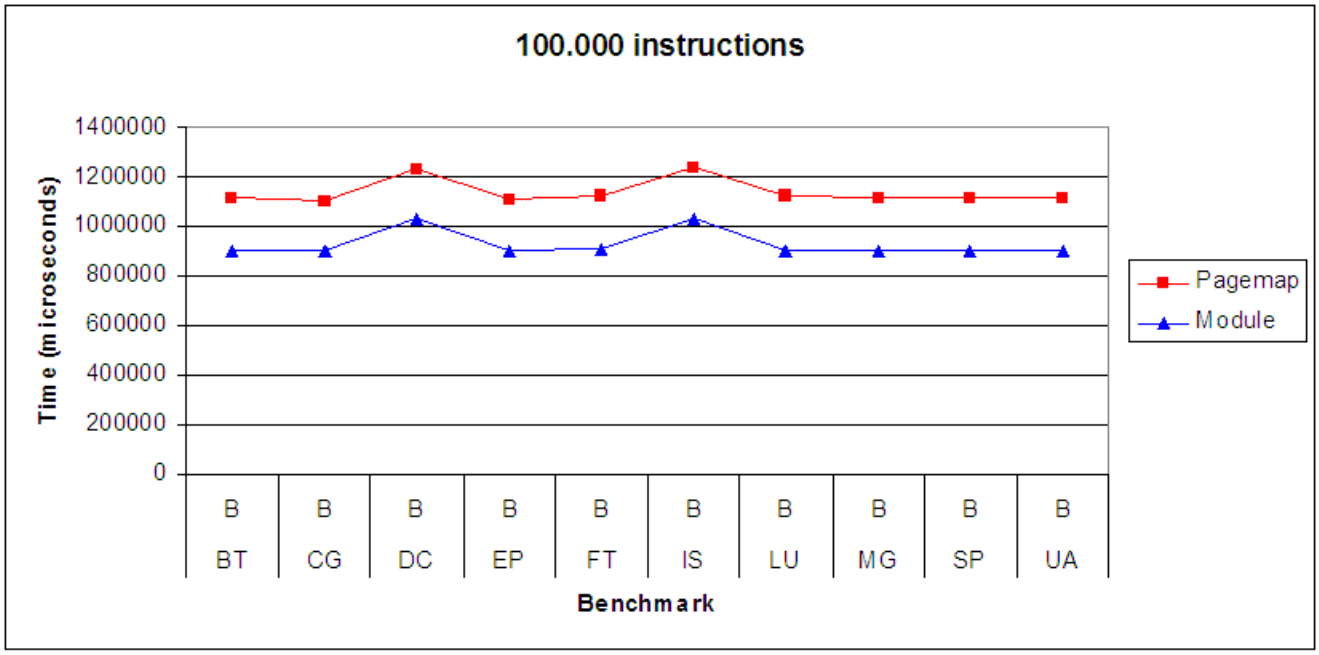
OpenMP

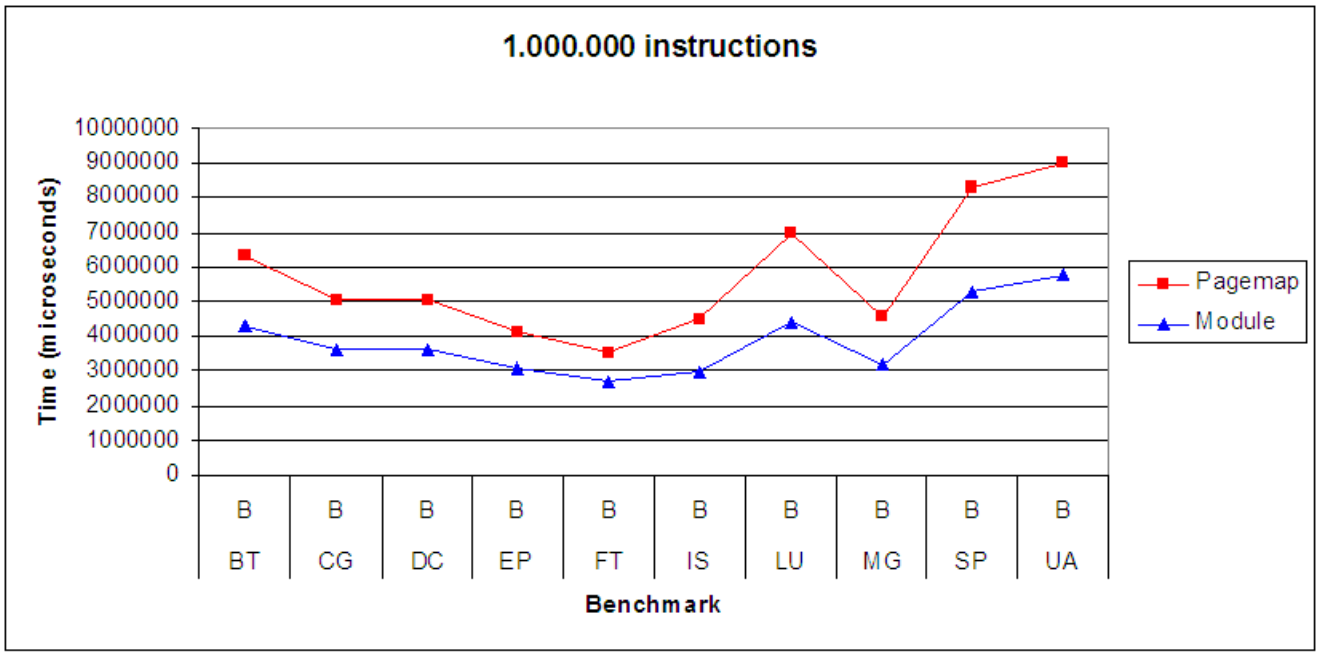


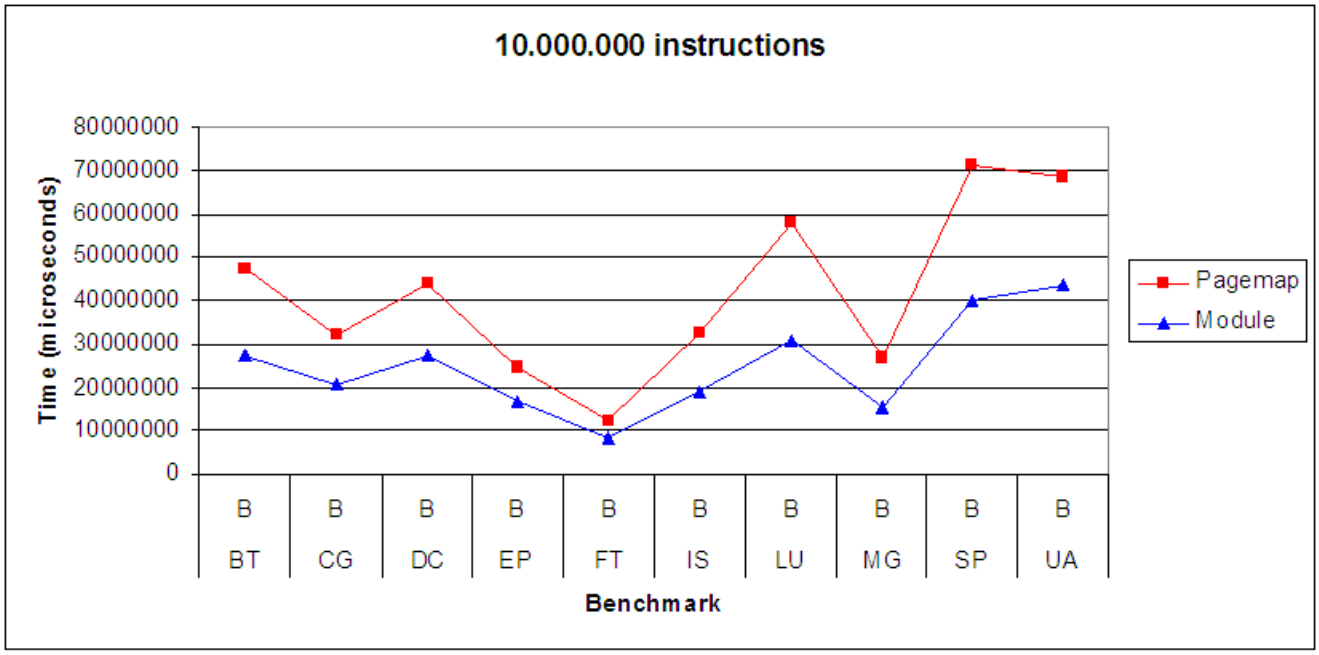




Serial Version




1.000.000 instructions
Time (microseconds)
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
B BT
B CG
B DC
B EP
B FT
B IS
B LU
B MG
B SP
B UA
Benchmark
Pagemap
Module


10.000.000 instructions
Time (microseconds)
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
B BT
B CG
B DC
B EP
B FT
B IS
B LU
B MG
B SP
B UA
Benchmark
Pagemap
Module

7.5. Conclusions

Tot el conjunt de gràfiques mostren, de manera molt gràfica, el rang de millora en el procés de traducció d'adreces fent servir el mòdul desenvolupat, en comparació amb el mètode anterior (Pagemap).

La millora és indiscutible i a mida que augmenta el número d'instruccions durant el procés d'instrumentació la millora de temps és més significativa, en termes generals.

Amb l'augment del número d'instruccions a executar, també es fa present la diferència entre els diferents tests. La millora de rendiment amb el test FT en una execució més llarga (10.000.000 instruccions) és més lleugera que amb el test LU, per posar un exemple.

Aquest mètode de traducció, a més a més de donar un rendiment major que el mètode Pagemap, també proporciona una traducció de les adreces virtuals més fidedigne i proper al que fa servir el nucli per fer les traduccions i manegar la memòria del sistema.

L'únic desavantatge, si es pot considerar així, és el fet que és un mètode més intrusiu ja que treballa directament amb les estructures del nucli, encara que només en mode consulta. D'altra banda, cal tindre permisos d'usuari root per tal de poder construir i carregar el mòdul al sistema i, en molts cosos, aquest fet pot ser problemàtic.

8. Bibliografia

Llibres

Mel Gorman (2004). Understanding the Linux Virtual Memory Manager (1st Ed.) New Jersey: Pearson Education, Inc. Prentice Hall Professional Technical Reference.

Daniel P. Bovet , Marco Cesati (2005). Understanding the Linux Kernel (3rd Ed.) Sebastopol CA: O'Reilly Media.

Robert Love (2010). Linux Kernel Development: A thorough guide to the design and implementation of the Linux kernel (3rd Ed.) Crawfordsville, Indiana: Pearson Education, Inc. Addison Wesley.

Jonathan Corbet , Alessandro Rubini (2005). Linux Device Drivers (3rd Ed.). Sebastopol CA: O'Reilly Media.

Peter Jay Salzman , Michael Burian , Ori Pomerantz (2009). The Linux Kernel Module Programming Guide (1st Ed.). Createspace.

Web

Anatomy of Linux Kernel
http://www.makelinux.net/reference?n=virtual%20memory

Linux Memory Management Project
http://linux-mm.org

Pin – A Dynamic Binary Instrumentation Tool
http://www.pintool.org

Pin 2.8 User Guide
http://www.pintool.org/docs/39599/Pin/html/

PinHead – Pin Users Space
http://tech.groups.yahoo.com/group/pinheads/

NAS Parallel Benchmarks
http://www.nas.nasa.gov/Resources/Software/npb.html

9. Annexos

9.1. Codi font module.c - x86

```
#include <linux/module.h>         /*   Needed by modules                          */
#include <linux/kernel.h>         /*   Needed by KERN_INFO                        */
#include <linux/init.h>           /*   Needed by macros                           */
#include <asm/page.h>             /*   Needed by __pa() macro                     */
#include <linux/sched.h>          /*   Needed by "for_each_process" macro         */
#include <linux/moduleparam.h>    /*   Needed by module params                    */
#include <linux/mm.h>
#include <linux/fs.h>
#include <asm/pgtable.h>
#include <linux/hardirq.h>
#include <linux/highmem.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <linux/spinlock.h>
#include "module.h"

#define OK 0
#define NOM_MODUL "traductor_adreces"


//  Concurrent access control
static int modul_obert = 0;


//  Global Variables declaration
struct mm_struct * mm;
struct page *pagina;

pgd_t *pgd;
pmd_t *pmd;
pte_t *ptep , pte;

int pid_tmp = 0;
int *pid = &pid_tmp;

int pfn;



//  Module Functions
//  ----------------

//  Function called when a process try to open a module to access it
static int device_open(struct inode *inode, struct file *file)
{

      //  Only one process can access the module
      //  Prevent that other process can access it when it's opened by a process
      if (modul_obert)
```

```
            return -EBUSY;

      modul_obert++;

      try_module_get(THIS_MODULE);

      return OK;
}


//  Function called when the process close the access to the module
static int device_release(struct inode *inode, struct file *file)
{

      //  Now the module can be accessed by other processes
      modul_obert--;

      module_put(THIS_MODULE);

      return OK;
}


//  Function called when the process try to access the module ops using IOCTL
mechanism
int device_ioctl(struct inode *inode, struct file *file, unsigned int ioctl_num,
unsigned long * ioctl_param)
{
    unsigned long va = 0;
      unsigned long pa;

      //  Change to IOCTL called
      switch (ioctl_num) {

        case IOCTL_ENVIAR_PID:
            pid = (int *)ioctl_param;
            printk(KERN_INFO "PID process:     %i \n", pid );
            struct task_struct *task;
        for_each_process(task)  {
          if  ( task->pid == pid )
              {
              mm = task->mm;
            }
        }
            break;

        case IOCTL_TRADUCTOR:
            va = (unsigned long *)ioctl_param;
            pa = 0;

            //  Variable initialization
            pagina = NULL;
          pgd = NULL;
          pmd = NULL;
          ptep = NULL;
```

```
          // Using Page Tables (this mechanism is known as "Page Walk"), we find
the page that corresponds to Virtual Address
          pgd  = pgd_offset(mm, va);
          if ( !pgd_none(*pgd) || !pgd_bad(*pgd) )
          {
            pmd  = pmd_offset(pgd, va);
            if ( !pmd_none(*pmd) || !pmd_bad(*pmd) )
            {
                ptep = pte_offset_map(pmd, va);
                if (ptep)
                {
                pte  = *ptep;
                  pte_unmap(ptep);
              pagina = pte_page(pte);

                    //  The page has been found
                    //  Seek "Page Frame Number" for this page
                    pfn = page_to_pfn(pagina);

                    //  Seek Physical Address for this page, using "page_to_phys()"
macro
                pa = page_to_phys(pagina);
                }
                  else   printk(KERN_INFO "Page Walk exception.  The Virtual
Address has not been found for this process. \n" );
            }
              else   printk(KERN_INFO "Page Walk exception.  The Virtual Address
has not been found for this process. \n" );
          }
            else   printk(KERN_INFO "Page Walk exception.  The Virtual Address has
not been found for this process. \n" );

            return pa;
            break;

        case IOCTL_REBRE_PFN:
          va = (unsigned long *)ioctl_param;
            return pfn;

        default:
          return -ENOTTY;
      }

      return OK;
}


//   Module ops
struct file_operations ops = {
      .owner   =  THIS_MODULE,
      .read    =  NULL,
      .write   =  NULL,
      .ioctl   =  device_ioctl,
      .open    =  device_open,
```

```
      .release =  device_release,
};



static int __init module_start(void)
{
  printk(KERN_INFO "Loading 'traductor_adreces' module ... \n");

  int init;
  init = register_chrdev(MAJOR_NUM, NOM_MODUL, &ops);

  if (init < 0)
  {
    printk(KERN_ALERT "Error loading module 'traductor_adreces' \n");
      printk(KERN_ALERT "Error code:  %d \n" , init );
      return init;
  }
  else
  {
    printk(KERN_INFO "Module loaded successfully \n");
      return 0;
  }
}



static void __exit module_end(void)
{
  printk(KERN_INFO "Unloading module 'traductor_adreces' ...\n");

  unregister_chrdev(MAJOR_NUM, NOM_MODUL);

}


module_init(module_start);
module_exit(module_end);


/*   Info sobre el modul   */
MODULE_AUTHOR ("J.A.Martin   <jamartin.dev@gmail.com>");
MODULE_DESCRIPTION ("Translate a Virtual Address to Physical one, using Page
Tables");
MODULE_LICENSE("GPL");
```

9.2. Codi font module.c - x86_64

```
#include <linux/module.h>         /*   Needed by modules                          */
#include <linux/kernel.h>         /*   Needed by KERN_INFO                        */
#include <linux/init.h>           /*   Needed by macros                           */
#include <asm/page.h>             /*   Needed by __pa() macro                     */
#include <linux/sched.h>          /*   Needed by "for_each_process" macro         */
#include <linux/moduleparam.h>    /*   Needed by module params                    */
#include <linux/mm.h>
#include <linux/fs.h>
#include <asm/pgtable.h>
#include <linux/hardirq.h>
#include <linux/highmem.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <linux/spinlock.h>
#include "module.h"

#define OK 0
#define NOM_MODUL "traductor_adreces"


//  Concurrent access control
static int modul_obert = 0;


//  Global Variables declaration
struct mm_struct * mm;
struct page *pagina;

pgd_t *pgd;
pud_t *pud;
pmd_t *pmd;
pte_t *ptep , pte;

int pid_tmp = 0;
int *pid = &pid_tmp;

int pfn;



//  Module Functions
//  ----------------

//  Function called when a process try to open a module to access it
static int device_open(struct inode *inode, struct file *file)
{

      //  Only one process can access the module
      //  Prevent that other process can access it when it's opened by a process
      if (modul_obert)
            return -EBUSY;
```

```
      modul_obert++;

      try_module_get(THIS_MODULE);

      return OK;
}


//  Function called when the process close the access to the module
static int device_release(struct inode *inode, struct file *file)
{

      //  Now the module can be accessed by other processes
      modul_obert--;

      module_put(THIS_MODULE);

      return OK;
}


//  Function called when the process try to access the module ops using IOCTL
mechanism
int device_ioctl(struct inode *inode, struct file *file, unsigned int ioctl_num,
unsigned long * ioctl_param)
{
    unsigned long va = 0;
      unsigned long pa;

      //  Change to IOCTL called
      switch (ioctl_num) {

        case IOCTL_ENVIAR_PID:
            pid = (int *)ioctl_param;
            printk(KERN_INFO "PID process: %i \n", pid );
            struct task_struct *task;
        for_each_process(task)  {
          if  ( task->pid == pid )
              {
              mm = task->mm;
             }
        }
            break;

        case IOCTL_TRADUCTOR:
            va = (unsigned long *)ioctl_param;
            pa = 0;

            //  Variable initialization
            pagina = NULL;
          pgd = NULL;
          pmd = NULL;
          ptep = NULL;

          // Using Page Tables (this mechanism is known as "Page Walk"), we find
```

```
the page that corresponds to Virtual Address
          pgd  = pgd_offset(mm, va);
          if ( !pgd_none(*pgd) || !pgd_bad(*pgd) )
          {
              pud = pud_offset(pgd , va);
              if ( !pud_none(*pud) || !pud_bad(*pud) )
              {
              pmd  = pmd_offset(pud, va);
              if ( !pmd_none(*pmd) || !pmd_bad(*pmd) )
              {
                  ptep = pte_offset_map(pmd, va);
                  if (ptep)
                  {
                  pte  = *ptep;
                    pte_unmap(ptep);
                pagina = pte_page(pte);

                      //  The page has been found
                      //  Seek Page Frame Number for this page
                      pfn = page_to_pfn(pagina);

                      //  Seek Physical Address for this page, using
"page_to_phys()" macro
                  pa = page_to_phys(pagina);

                  }
                    else   printk(KERN_INFO "Page Walk exception.  The Virtual
Address has not been found for this process. \n" );
              }
                else   printk(KERN_INFO "Page Walk exception.  The Virtual Address
has not been found for this process. \n" );
            }
              else   printk(KERN_INFO "Page Walk exception.  The Virtual Address
has not been found for this process. \n" );
            }
            else   printk(KERN_INFO "Page Walk exception.  The Virtual Address has
not been found for this process. \n" );

            return pa;
            break;

        case IOCTL_REBRE_PFN:
          va = (unsigned long *)ioctl_param;
            return pfn;

        default:
          return -ENOTTY;
      }

      return OK;
}


//   Module ops
struct file_operations ops = {
```

```
      .owner   =  THIS_MODULE,
      .read    =  NULL,
      .write   =  NULL,
      .ioctl   =  device_ioctl,
      .open    =  device_open,
      .release =  device_release,
};



static int __init module_start(void)
{
  printk(KERN_INFO "Loading 'traductor_adreces' module ... \n");

  int init;
  init = register_chrdev(MAJOR_NUM, NOM_MODUL, &ops);

  if (init < 0)
  {
    printk(KERN_ALERT "Error loading module 'traductor_adreces' \n");
      printk(KERN_ALERT "Error code:  %d \n" , init );
      return init;
  }
  else
  {
    printk(KERN_INFO "Module loaded successfully \n");
      return 0;
  }
}



static void __exit module_end(void)
{
  printk(KERN_INFO "Unloading module 'traductor_adreces' ...\n");

  unregister_chrdev(MAJOR_NUM, NOM_MODUL);

}


module_init(module_start);
module_exit(module_end);


/*   Info sobre el modul   */
MODULE_AUTHOR ("J.A.Martin   <jamartin.dev@gmail.com>");
MODULE_DESCRIPTION ("Translate a Virtual Address to Physical one, using Page
Tables");
MODULE_LICENSE("GPL");
```

9.3. Codi font module.h - x86 i x86_64

```
#ifndef MODULE_H
#define MODULE_H

#include <linux/ioctl.h>

#define MAJOR_NUM 100

#define IOCTL_ENVIAR_PID _IOR(MAJOR_NUM,  0 , int * )
#define IOCTL_TRADUCTOR  _IOWR(MAJOR_NUM, 1 , unsigned long * )
#define IOCTL_REBRE_PFN  _IOWR(MAJOR_NUM, 2 , unsigned long * )

#define DEVICE_FILE_NAME "traductor_adreces"

#endif
```

9.4. Codi font mtaptrace.cpp

```
#define _LARGEFILE64_SOURCE
#include "pin.H"
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/sysinfo.h>
#include <dirent.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <iostream>
#include <string>
#include <set>
#include <map>
#include <sys/ioctl.h>
#include "zlib.h"
#include "module.h"


/******   Code added by Jose Antonio Martin    ******/
//  Variable for Module
int module = 0;
unsigned long virtual_address , physical_address;

int page_success = 0;
int page_fail = 0;
/************************************************/
```

```
// Number of threads count
INT32 numThreads = 0;

// Set to hold the threads to print
set<THREADID> threads;

// A boolean to signal when upper thresholds reached
static bool topReached = false;

// Total running count of instructions, memory accesses, reads and writes
static UINT64 icount = 0;
static UINT64 rds_wrts = 0;
static UINT64 rds = 0;
static UINT64 wrts = 0;

// Pointer to the global map of access counters in analytic mode
map<UINT64, UINT64>* am_map;

// Force each thread's data to be in its own data cache line so that
// multiple threads do not contend for the same data cache line.
// This avoids the false sharing problem. 64 byte line size.
#define PADSIZE (sizeof(UINT64) * 4 + sizeof(FILE *) + sizeof(map<UINT64,
UINT64>*)) % 64

// Per thread running count of instructions, memory accesses, reads and writes
class thread_data_t
{
  public:
    thread_data_t() : _icount(0), _rds_wrts(0), _rds(0), _wrts(0) {}
    UINT64 _icount;
    UINT64 _rds_wrts;
    UINT64 _rds;
    UINT64 _wrts;
    map<UINT64, UINT64>* _am_map;
    FILE * _trace;
    UINT8 _pad[PADSIZE];
};

// Key for accessing TLS storage in the threads. initialized once in main()
static  TLS_KEY tls_key;

// Function to access thread-specific data
thread_data_t* get_tls(THREADID threadid)
{
    thread_data_t* tdata =
          static_cast<thread_data_t*>(PIN_GetThreadData(tls_key, threadid));
    return tdata;
}

// Memory info
int cache_top_level = 0;
long int page_size = 0;
long int cache_size = 0;
```

```
long int cache_linesize = 0;
long int cache_associativity = 0;
long int number_of_sets = 0;

// File descriptor for pagemap
int pagemap = -1;

// Print thresholds & sampling
static UINT64 lthreshold = 1;
static UINT64 uthreshold = 18446744073709551615LLU;
static UINT64 sampling = 1;

/* ================================================================== */
/* Commandline Switches */
/* ================================================================== */

static KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
   "mtaptrace.out", "Specify output file name to store the application trace");
static KNOB<bool> KnobUniqueOutputFile(KNOB_MODE_WRITEONCE, "pintool",
   "unique_outputfile", "0", "The output file names will contain the pid");
static KNOB<bool> KnobThFiles(KNOB_MODE_WRITEONCE, "pintool", "tf", "0",
   "Creates one file per thread appending thread id to output filename");
static KNOB<bool> KnobIcount(KNOB_MODE_WRITEONCE, "pintool", "ic", "1",
   "Print to output file the instruction counter (icount)");
static KNOB<bool> KnobMcount(KNOB_MODE_WRITEONCE, "pintool", "mc", "1",
   "Print to output file the memory access counter (mcount)");
static KNOB<bool> KnobThIcount(KNOB_MODE_WRITEONCE, "pintool", "tic", "0",
   "Print to output file the thread instruction counter (ticount)");
static KNOB<bool> KnobThMcount(KNOB_MODE_WRITEONCE, "pintool", "tmc", "0",
   "Print to output file the thread memory access counter (tmcount)");
static KNOB<bool> KnobTimestamp(KNOB_MODE_WRITEONCE, "pintool", "ts", "1",
   "Print to output file the time stamp of the memory access (timestamp)");
static KNOB<bool> KnobThreadid(KNOB_MODE_WRITEONCE, "pintool", "th", "0",
   "Print to output file the thread identifier assigned by PIN (threadid)");
static KNOB<bool> KnobIaddr(KNOB_MODE_WRITEONCE, "pintool", "ia", "0",
   "Print to output file the instruction address (iaddr)");
static KNOB<bool> KnobTypeop(KNOB_MODE_WRITEONCE, "pintool", "to", "1",
   "Print to output file the type of operation (R)ead or (W)rite (typeop)");
static KNOB<bool> KnobNrops(KNOB_MODE_WRITEONCE, "pintool", "no", "0",
   "Print to output file the number of operands of the instruction (nrops)");
static KNOB<bool> KnobOpsize(KNOB_MODE_WRITEONCE, "pintool", "os", "0",
   "Print to output file the operation size in bytes (opsize)");
static KNOB<bool> KnobVaddr(KNOB_MODE_WRITEONCE, "pintool", "va", "1",
   "Print to output file the virtual address of the memory access (vaddr)");
static KNOB<bool> KnobNcacheline(KNOB_MODE_WRITEONCE, "pintool", "nc", "0",
   "Print to output file the number of cache line (ncacheline)");
static KNOB<bool> KnobNset(KNOB_MODE_WRITEONCE, "pintool", "ns", "0",
   "Print to output file the number of cache set (nset)");
static KNOB<bool> KnobNpage(KNOB_MODE_WRITEONCE, "pintool", "np", "0",
   "Print to output file the number of page (npage)");
static KNOB<bool> KnobPaddr(KNOB_MODE_WRITEONCE, "pintool", "pa", "0",
   "Print to output file the physical address info (several fields)");
static KNOB<bool> KnobRead(KNOB_MODE_WRITEONCE, "pintool", "r", "1",
   "Print to output file the read accesses");
static KNOB<bool> KnobWrite(KNOB_MODE_WRITEONCE, "pintool", "w", "1",
```

```
   "Print to output file the write accesses");
static KNOB<string> KnobThList(KNOB_MODE_WRITEONCE, "pintool", "tl", "",
   "Only print to file the threads specified as a comma separated list");
static KNOB<string> KnobLowerThreshold(KNOB_MODE_WRITEONCE,
   "pintool", "lt", "1",
   "Do not print instructions or accesses below that lower threshold");
static KNOB<string> KnobUpperThreshold(KNOB_MODE_WRITEONCE,
   "pintool", "ut", "18446744073709551615",
   "Do not print instructions or accesses over that upper threshold");
static KNOB<string> KnobSampling(KNOB_MODE_WRITEONCE, "pintool", "s", "1",
   "Sampling. Only print every specied number of nstructions or accesses");
static KNOB<bool> KnobRelThres(KNOB_MODE_WRITEONCE, "pintool", "rt", "1",
   "Thresholds and sampling relative to instruction(1) or memory access(0)");
static KNOB<bool> KnobForceExit(KNOB_MODE_WRITEONCE, "pintool", "fx", "1",
   "The pintool exits when the upper threshold is reached");
static KNOB<bool> KnobPad(KNOB_MODE_WRITEONCE, "pintool", "pz", "0",
   "Print numbers padded with zeroes to the left");
static KNOB<bool> KnobHex(KNOB_MODE_WRITEONCE, "pintool", "0x", "0",
   "Print hex numbers with 0x prefix");
static KNOB<bool> KnobGzip(KNOB_MODE_WRITEONCE, "pintool", "gz", "0",
   "Compress output file on-the-fly appending .gz to file name(s)");
static KNOB<bool> KnobStd(KNOB_MODE_WRITEONCE, "pintool", "std", "1",
   "Print info and error messages to stdout");
static KNOB<bool> KnobAnalyticMode(KNOB_MODE_WRITEONCE, "pintool", "am", "0",
   "In analytic mode keeps count of accesses to each page frame number");

/******   Code added by Jose Antonio Martin    ******/
static KNOB<bool> KnobUseModuleAddressTranslator(KNOB_MODE_WRITEONCE, "pintool",
"mat", "0",
   "Use Module Address Translator to convert Virtual Addresses to Physical ones");
static KNOB<bool> KnobCompareMethods(KNOB_MODE_WRITEONCE, "pintool", "cm", "0",
   "Compare two methods to get Physical Address");
/************************************************/


// Format strings
static string countFormat = "%llu ";
static string addrFormat = "%llx ";
static string pageFormat = "%llx ";
static string opsFormat = "%u ";
static string tsFormat = "%llu ";
static string nsetFormat = "%lu ";
static string tidFormat = "%u ";
static string phnpFormat = "NP %llx ";
static string phprFormat = "PR %llu %llx ";
static string phswFormat = "SW %llu %llx %llx ";

// Output file
FILE * trace;

// Lock for threads
PIN_LOCK lock;

// Function to set formats for printf
void set_print_formats()
```

```
{
    if (KnobPad)
    {
        countFormat = "%010llu ";
        addrFormat = "%012llx ";
        pageFormat = "%09llx ";
        opsFormat = "%u ";
        tsFormat = "%010llu ";
        nsetFormat = "%04lu ";
        tidFormat = "%03u ";
        phnpFormat = "NP %012llx ";
        phprFormat = "PR %llu %09llx ";
        phswFormat = "SW %llu %llx %012llx ";
    }
    if (KnobHex)
    {
        phnpFormat = "NP 0x" + addrFormat;
        phprFormat = "PR %llu 0x" + pageFormat;
        phswFormat = "SW %llu %llx 0x" + addrFormat;
        addrFormat = "0x" + addrFormat;
        pageFormat = "0x" + pageFormat;
    }
}

// Function to print header in the output file
void print_header(FILE * trace)
{
    if (!KnobAnalyticMode)
    {
        char header[256];
        sprintf(header, "#");
        if (KnobIcount) strcat(header, "icount ");
        if (KnobMcount) strcat(header, "mcount ");
        if (KnobThIcount) strcat(header, "ticount ");
        if (KnobThMcount) strcat(header, "tmcount ");
        if (KnobTimestamp) strcat(header, "timestamp ");
        if (KnobThreadid) strcat(header, "threadid ");
        if (KnobIaddr) strcat(header, "iaddr ");
        if (KnobTypeop) strcat(header, "typeop ");
        if (KnobNrops) strcat(header, "nrops ");
        if (KnobOpsize) strcat(header, "opsize ");
        if (KnobVaddr) strcat(header, "vaddr ");
        if (KnobNcacheline) strcat(header, "ncacheline ");
        if (KnobNset) strcat(header, "nset ");
        if (KnobNpage) strcat(header, "npage ");
        if (KnobPaddr) strcat(header,
                              "pa_stat pmword|pgsize pfn|swtype swoffset ");
        char* last = header + strlen(header) - 1;
        if (*last == ' ') *last = '\0';
        if (KnobGzip)
        {
             gzprintf(trace, "%s\n", header);
        }
        else
        {
```

```
            fprintf(trace, "%s\n", header);
        }
    }
    else
    {
        if (KnobGzip)
        {
             gzprintf(trace,
 "#pfn rds_wrts\n");
        }
        else
        {
            fprintf(trace, "#pfn rds_wrts\n");
        }
    }
}

// Function to check if a string is numeric
static bool is_numeric(const char *str)
{
    while (*str != '\0')
    {
        if (!isdigit (*str))
        {
            return false;
        }
        str++;
    }
    return true;
}

// Function to check thresholds and sampling
int check_threshold_sampling()
{
    char logThrs[80] = "";
    // Check lower threshold
    if (is_numeric(KnobLowerThreshold.Value().c_str())
        && ((lthreshold = Uint64FromString(KnobLowerThreshold.Value())) > 0))
    {
        sprintf(logThrs, "Lower threshold: %llu\n",
                   (unsigned long long)lthreshold);
        if (KnobStd) printf("%s", logThrs); LOG(logThrs);
    }
    else
    {
        sprintf(logThrs, "Invalid lower threshold specified: %s\n",
                              KnobLowerThreshold.Value().c_str());
        if (KnobStd) printf("%s", logThrs); LOG(logThrs);
        return 0;
    }

    // Check sampling
    if (is_numeric(KnobSampling.Value().c_str())
        && ((sampling = Uint64FromString(KnobSampling.Value())) > 0))
    {
```

```
        sprintf(logThrs, "Sampling: %llu\n", (unsigned long long)sampling);
        if (KnobStd) printf("%s", logThrs); LOG(logThrs);
    }
    else
    {
        sprintf(logThrs, "Invalid sampling specified: %s\n",
                              KnobSampling.Value().c_str());
        if (KnobStd) printf("%s", logThrs); LOG(logThrs);
        return 0;
    }

   // Check upper threshold
    if (is_numeric(KnobUpperThreshold.Value().c_str())
        && ((uthreshold = Uint64FromString(KnobUpperThreshold.Value())) > 0))
    {
        sprintf(logThrs, "Upper threshold: %llu\n",
                    (unsigned long long)uthreshold);
        if (KnobStd) printf("%s", logThrs); LOG(logThrs);
    }
    else
    {
        sprintf(logThrs, "Invalid upper threshold specified: %s\n",
                               KnobUpperThreshold.Value().c_str());
        if (KnobStd) printf("%s", logThrs); LOG(logThrs);
        return 0;
    }
    return 1;
}

// Function to check thread list
int check_thlist()
{
    if (!KnobThList.Value().empty())
    {
        size_t n = KnobThList.Value().length();
        size_t start = KnobThList.Value().find_first_not_of(",");
        while (start < n)
        {
            size_t stop = KnobThList.Value().find_first_of (",", start);
            if (stop > n) stop = n;
            string th = KnobThList.Value().substr(start, stop - start);
            if (is_numeric(th.c_str()))
                threads.insert(Int32FromString(th));
            start = KnobThList.Value().find_first_not_of(",", stop + 1);
        }
        if (!threads.empty())
        {
            string logThList = "Threads to print:";
            char logThread[32];
            set<THREADID>::iterator it;
            for (it=threads.begin(); it!=threads.end(); it++)
            {
                sprintf(logThread, " %d", *it);
                logThList += logThread;
            }
```

```
            if (KnobStd) printf("%s\n", logThList.c_str()); LOG(logThList + "\n");
        }
        else
        {
            if (KnobStd) printf("Switch -tl specified but no thread id found.\n");
            LOG("Switch -tl specified but no thread id found.\n");
            return 0;
        }
    }
    return 1;
}

// Function to check if a thread is selected to print
bool thPrint(THREADID tid)
{
    if (threads.empty()) return true;
    if (threads.find(tid) != threads.end()) return true;
    return false;
}

// Function to get page size
int get_page_size()
{
    page_size = sysconf(_SC_PAGE_SIZE);
    char logMinfo[64] = "";
    if (page_size == 0)
    {
        sprintf(logMinfo, "Can't find page size.");
        if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
        return 0;
    }
    sprintf(logMinfo, "Page size: %ld\n", page_size);
    if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
    return 1;
}

// Function to get cache info
int get_cache_info()
{
    if ((cache_size = sysconf(_SC_LEVEL4_CACHE_SIZE)) > 0L)
    {
        cache_top_level = 4;
        cache_linesize = sysconf(_SC_LEVEL4_CACHE_LINESIZE);
        cache_associativity = sysconf(_SC_LEVEL4_CACHE_ASSOC);
        number_of_sets = (cache_size/(cache_associativity*cache_linesize));
    }
    else if ((cache_size = sysconf(_SC_LEVEL3_CACHE_SIZE)) > 0L)
    {
        cache_top_level = 3;
        cache_linesize = sysconf(_SC_LEVEL3_CACHE_LINESIZE);
        cache_associativity = sysconf(_SC_LEVEL3_CACHE_ASSOC);
        number_of_sets = (cache_size/(cache_associativity*cache_linesize));
    }
    else if ((cache_size = sysconf(_SC_LEVEL2_CACHE_SIZE)) > 0L)
    {
```

```
        cache_top_level = 2;
        cache_linesize = sysconf(_SC_LEVEL2_CACHE_LINESIZE);
        cache_associativity = sysconf(_SC_LEVEL2_CACHE_ASSOC);
        number_of_sets = (cache_size/(cache_associativity*cache_linesize));
    }
    else if ((cache_size = sysconf(_SC_LEVEL1_DCACHE_SIZE)) > 0L)
    {
        cache_top_level = 1;
        cache_linesize = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
        cache_associativity = sysconf(_SC_LEVEL1_DCACHE_ASSOC);
        number_of_sets = (cache_size/(cache_associativity*cache_linesize));
    }

    char logMinfo[64] = "";
    if (cache_top_level == 0)
    {
        sprintf(logMinfo, "Can't find cache memory info.");
        if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
        return 0;
    }
    sprintf(logMinfo, "Cache top level: L%d\n", cache_top_level);
    if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
    sprintf(logMinfo, "Cache size: %ld\n", cache_size);
    if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
    sprintf(logMinfo, "Cache line size: %ld\n", cache_linesize);
    if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
    sprintf(logMinfo, "Cache associativity: %ld\n", cache_associativity);
    if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
    sprintf(logMinfo, "Number of sets: %ld\n", number_of_sets);
    if (KnobStd) printf("%s", logMinfo); LOG(logMinfo);
    return 1;
}

// Function to print counters to stdout and logfile
void print_counters()
{
    char logCounts[120] = "Mtaptrace resuming...\n";
    if (KnobStd) printf("%s", logCounts); LOG(logCounts);
    sprintf(logCounts, "Number of threads: %d\n", numThreads);
    if (KnobStd) printf("%s", logCounts); LOG(logCounts);
    sprintf(logCounts, "%s %s %s %s %s\n",
              "#thread_id", "(t)icount", "(t)rds_wrts", "(t)rds", "(t)wrts");
    if (KnobStd) printf("%s", logCounts); LOG(logCounts);
    for (INT32 t=0; t<numThreads; t++)
    {
        thread_data_t* tdata = get_tls(t);
        sprintf(logCounts, "%s %llu %llu %llu %llu\n", decstr(t).c_str(),
                                      (unsigned long long)tdata->_icount,
                                   (unsigned long long)tdata->_rds_wrts,
                                        (unsigned long long)tdata->_rds,
                                     (unsigned long long)tdata->_wrts);
        if (KnobStd) printf("%s", logCounts); LOG(logCounts);
    }
    sprintf(logCounts, "%s %llu %llu %llu %llu\n", "all",
                    (unsigned long long)icount, (unsigned long long)rds_wrts,
```

```
                            (unsigned long long)rds, (unsigned long long)wrts);
    if (KnobStd) printf("%s", logCounts); LOG(logCounts);
}

// Function to print map in analytic mode with thread files
void print_am_map_th(thread_data_t* tdata)
{
    string mapFormat = pageFormat + countFormat + "\n";
    mapFormat.erase(mapFormat.end() - 2);
    //thread_data_t* tdata = get_tls(threadid);
    map<UINT64, UINT64>::const_iterator end = (*tdata->_am_map).end();
    map<UINT64, UINT64>::const_iterator it;
    for (it = (*tdata->_am_map).begin(); it != end; it++)
    {
        if (KnobGzip)
        {
            gzprintf(tdata->_trace, mapFormat.c_str(),
                                    (unsigned long long)(*it).first,
                                   (unsigned long long)(*it).second);
        }
        else
        {
            fprintf(tdata->_trace, mapFormat.c_str(),
                                    (unsigned long long)(*it).first,
                                   (unsigned long long)(*it).second);
        }
    }
    if (KnobGzip)
    {
        gzflush(tdata->_trace, Z_SYNC_FLUSH);
    }
    else
    {
        fflush(tdata->_trace);
    }
}

// Function to print map in analytic mode
void print_am_map()
{
    string mapFormat = pageFormat + countFormat + "\n";
    mapFormat.erase(mapFormat.end() - 2);
    map<UINT64, UINT64>::const_iterator end = (*am_map).end();
    map<UINT64, UINT64>::const_iterator it;
    for (it = (*am_map).begin(); it != end; it++)
    {
        if (KnobGzip)
        {
            gzprintf(trace, mapFormat.c_str(),
                                    (unsigned long long)(*it).first,
                                   (unsigned long long)(*it).second);
        }
        else
        {
            fprintf(trace, mapFormat.c_str(),
```

```
                                        (unsigned long long)(*it).first,
                                       (unsigned long long)(*it).second);
        }
    }
    if (KnobGzip)
    {
        gzflush(trace, Z_SYNC_FLUSH);
    }
    else
    {
        fflush(trace);
    }
}

// Function to get time stamp
unsigned long long get_ts()
{
    timeval t;
    gettimeofday(&t, NULL);
    return t.tv_sec + t.tv_usec;
}

// Function to get page number
ADDRINT get_page(ADDRINT addr)
{
    return addr / page_size;
}

// Function to get cache line number
ADDRINT get_cacheline(ADDRINT addr)
{
    return addr / cache_linesize;
}

// Function to get cache set number
long int get_nset(ADDRINT addr)
{
    return get_cacheline(addr) % number_of_sets;
}

// Class to hold physical address info
class paddr_t {
  public:
    unsigned long physical_address;
      unsigned
 long long pm_word, pagesize, pfn, swaptype, swapoffs;
    bool page_present, page_swapped, seek_error, read_error, no_module;
    off64_t index , index2;
    paddr_t(ADDRINT);
    const char* c_str();
};

// Constructor that gets physical address info
paddr_t::paddr_t(ADDRINT vaddr)
{
```

```
page_present = false;
page_swapped = false;
seek_error = false;
read_error = false;
  unsigned long virtual_address;

  /******   Code added by Jose Antonio Martin    ******/

/* Seek to appropriate index of pagemap file. */

  if ( KnobCompareMethods )
  {
      // 1st - Seek page using Pagemap
        // ----------------------------
        index = (vaddr / page_size) * sizeof(unsigned long long);

    off64_t o;
    o = lseek64(pagemap, index, SEEK_SET);
    if (o != index)
    {
        seek_error = true;
    }
    else
    {
        /* Read a 64-bit word from the pagemap file. */
        ssize_t t;
        t = read(pagemap, &pm_word, sizeof(unsigned long long));
        if (t < 0)
        {
            read_error = true;
        }
    }

    if (!seek_error && !read_error)
    {
        if ((pm_word & (1ULL << 63)) > 0)
        {
            page_present = true;
        }
        if ((pm_word & (1ULL << 62)) > 0)
        {
            page_swapped = true;
        }
        if (page_present || page_swapped)
        {
            pagesize = 1ULL << ((pm_word & ((1ULL << 61) - 1)) >> 55);
        }
        if (page_present)
            {
            pfn = (pm_word & ((1ULL << 55) - 1));
        }
        if (page_swapped)
        {
            swapoffs = ((pm_word & ((1ULL << 55) - 1)) >> 5);
            swaptype = (pm_word & ((1ULL << 5) - 1));
```

```
            }
        }

            // 2nd - Seek page unsing module (Page Walk process)
            // -------------------------------------------------
            virtual_address  = (ADDRINT)vaddr;
            physical_address = ioctl(module, IOCTL_TRADUCTOR , virtual_address );
            int pfn2 = ioctl(module, IOCTL_REBRE_PFN , virtual_address );

            // 3rd - Compare pages found with each method (using Page Frame Number)
            //
-----------------------------------------------------------------------
            int pfn1 = (unsigned long long)pfn;
            if ( pfn1 != pfn2 )   page_fail++;
            else   page_success++;

            printf("Page success: %d  --  Page fail: %d \n" , page_success ,
page_fail);
      }
      else
      if ( KnobUseModuleAddressTranslator )
      {
            if (module < 0)
            {
                printf("Error opening module:  %s \n", DEVICE_FILE_NAME);
                printf("Error code:  %d \n", module );
                exit(-1);
          }
            else
        {
                virtual_address = (ADDRINT)vaddr;
            physical_address = 0;
                physical_address = ioctl(module, IOCTL_TRADUCTOR ,
virtual_address );

                  if ( physical_address != 0 )
                  {
                      pfn = (int)ioctl(module, IOCTL_REBRE_PFN , virtual_address );
                        pagesize = sysconf(_SC_PAGE_SIZE);
                  }
            }
      }

      /************************************************/

      else
      {
        index = (vaddr / page_size) * sizeof(unsigned long long);

        off64_t o;
        o = lseek64(pagemap, index, SEEK_SET);
        if (o != index)
        {
            seek_error = true;
        }
```

```
        else
        {
            /* Read a 64-bit word from the pagemap file. */
            ssize_t t;
            t = read(pagemap, &pm_word, sizeof(unsigned long long));
            if (t < 0)
            {
                read_error = true;
            }
        }

        if (!seek_error && !read_error)
        {
            /* Bits 0-54  page frame number (PFN) if present
             * Bits 0-4   swap type if swapped
             * Bits 5-54  swap offset if swapped
             * Bits 55-60 page shift (page size = 1<<page shift)
             * Bit  61    reserved for future use
             * Bit  62    page swapped
             * Bit  63    page present
             */

                //unsigned long long tmp = (unsigned long)physical_address;
            //printf("pm_word: %llu  ---  physical_address: %llu \n" , pm_word,
tmp);

            if ((pm_word & (1ULL << 63)) > 0)
            {
                page_present = true;
            }
            if ((pm_word & (1ULL << 62)) > 0)
            {
                page_swapped = true;
            }
            if (page_present || page_swapped)
            {
                pagesize = 1ULL << ((pm_word & ((1ULL << 61) - 1)) >> 55);
            }
            if (page_present)
                {
                pfn = (pm_word & ((1ULL << 55) - 1));
            }
            if (page_swapped)
            {
                swapoffs = ((pm_word & ((1ULL << 55) - 1)) >> 5);
                swaptype = (pm_word & ((1ULL << 5) - 1));
            }
        }
      }
}

// Function to get physical address info in string form
const char* paddr_t::c_str()
{
    char result[128] = "ERROR";
```

```
    if (!page_present && !page_swapped) {
        sprintf(result, phnpFormat.c_str(), pm_word);
    }
    else
    {
        if (!page_swapped)
        {
            sprintf(result, phprFormat.c_str(), pagesize, pfn);
        }
        else
        {
            sprintf(result, phswFormat.c_str(), pagesize, swaptype, swapoffs);
        }
    }
    string res = result;
    return res.c_str();
}

// Function to get physical address info
paddr_t get_physical_address(ADDRINT vaddr)
{
        char logErr[128] = "";
        paddr_t paddr(vaddr);

        if (paddr.seek_error)
        {
            sprintf(logErr, "Error seeking to %ld in pagemap file.\n",
paddr.index);
            if (KnobStd) printf("%s", logErr); LOG(logErr);
        }
        if (paddr.read_error)
        {
            sprintf(logErr, "Error reading pagemap file.\n");
            if (KnobStd) printf("%s", logErr); LOG(logErr);
        }

        return paddr;
}

// This function is called before every instruction is executed
VOID docount(THREADID threadid)
{
    GetLock(&lock, threadid+1);
    icount++;
    ReleaseLock(&lock);
    thread_data_t* tdata = get_tls(threadid);
    tdata->_icount++;
}

// Print a memory read record when thread separated files are required
VOID RecordMemReadTh(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                        ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; rds++;
```

```
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_rds++;
    UINT64 c;
    if (KnobRelThres) c = tdata->_icount;
    else c = tdata->_rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobRead && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            char line[256] = "\0";
            if (KnobIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                    (unsigned long long)icount);
            if (KnobMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                   (unsigned long long)rds_wrts);
            if (KnobThIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                        (unsigned long long)tdata->_icount);
            if (KnobThMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                      (unsigned long long)tdata->_rds_wrts);
            if (KnobTimestamp) sprintf(strchr(line,0), tsFormat.c_str(),
                                                                 get_ts());
            if (KnobThreadid) sprintf(strchr(line,0), tidFormat.c_str(), tid);
            if (KnobIaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                     (unsigned long long)ip);
            if (KnobTypeop) sprintf(strchr(line,0), "R ");
            if (KnobNrops) sprintf(strchr(line,0), opsFormat.c_str(), ops);
            if (KnobOpsize) sprintf(strchr(line,0), opsFormat.c_str(), sz);
            if (KnobVaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                    (unsigned long long)addr);
            if (KnobNcacheline) sprintf(strchr(line,0), addrFormat.c_str(),
                                    (unsigned long long)get_cacheline(addr));
            if (KnobNset) sprintf(strchr(line,0), nsetFormat.c_str(),
                                                           get_nset(addr));
            if (KnobNpage) sprintf(strchr(line,0), pageFormat.c_str(),
                                        (unsigned long long)get_page(addr));
            if (KnobPaddr) sprintf(strchr(line,0), "%s",
                                       get_physical_address(addr).c_str());
            char* last = line + strlen(line) - 1;
            if (*last == ' ') *last = '\0';
            if (KnobGzip)
            {
                gzprintf(tdata->_trace, "%s\n", line);
                gzflush(tdata->_trace, Z_SYNC_FLUSH);
            }
            else
            {
                fprintf(tdata->_trace, "%s\n", line);
                fflush(tdata->_trace);
            }
        }
    if ((c >= uthreshold) && thPrint(tid)) topReached = true;
    ReleaseLock(&lock);
}

// Print a memory read record
VOID RecordMemRead(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                       ADDRINT addr)
```

```
{
    GetLock(&lock, tid+1);
    rds_wrts++; rds++;
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_rds++;
    UINT64 c;
    if (KnobRelThres) c = icount;
    else c = rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobRead && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            char line[256] = "\0";
            if (KnobIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                    (unsigned long long)icount);
            if (KnobMcount) sprintf(strchr(line,0), countFormat.c_str(),

                                                (unsigned long long)rds_wrts);
            if (KnobThIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                            (unsigned long long)tdata->_icount);
            if (KnobThMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                          (unsigned long long)tdata->_rds_wrts);
            if (KnobTimestamp) sprintf(strchr(line,0), tsFormat.c_str(),
                                                              get_ts());
            if (KnobThreadid) sprintf(strchr(line,0), tidFormat.c_str(), tid);
            if (KnobIaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                     (unsigned long long)ip);
            if (KnobTypeop) sprintf(strchr(line,0), "R ");
            if (KnobNrops) sprintf(strchr(line,0), opsFormat.c_str(), ops);
            if (KnobOpsize) sprintf(strchr(line,0), opsFormat.c_str(), sz);
            if (KnobVaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                    (unsigned long long)addr);
            if (KnobNcacheline) sprintf(strchr(line,0), addrFormat.c_str(),
                                       (unsigned long long)get_cacheline(addr));
            if (KnobNset) sprintf(strchr(line,0), nsetFormat.c_str(),
                                                           get_nset(addr));
            if (KnobNpage) sprintf(strchr(line,0), pageFormat.c_str(),
                                            (unsigned long long)get_page(addr));
            if (KnobPaddr) sprintf(strchr(line,0), "%s",
                                           get_physical_address(addr).c_str());
            char* last = line + strlen(line) - 1;
            if (*last == ' ') *last = '\0';
            if (KnobGzip)
            {
                gzprintf(trace, "%s\n", line);
                gzflush(trace, Z_SYNC_FLUSH);
            }
            else
            {
                fprintf(trace, "%s\n", line);
                fflush(trace);
            }
        }
    if (c >= uthreshold) topReached = true;
    ReleaseLock(&lock);
}
```

```
// Count a memory read when thread separated files are required
VOID CountMemReadTh(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                        ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; rds++;
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_rds++;
    UINT64 c;
    if (KnobRelThres) c = tdata->_icount;
    else c = tdata->_rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobRead && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            paddr_t pa = get_physical_address(addr);
            if (pa.page_present) (*tdata->_am_map)[pa.pfn]++;
        }
    if ((c >= uthreshold) && thPrint(tid)) topReached = true;
    ReleaseLock(&lock);
}

// Count a memory read
VOID CountMemRead(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                        ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; rds++;
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_rds++;
    UINT64 c;
    if (KnobRelThres) c = icount;
    else c = rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobRead && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            paddr_t pa = get_physical_address(addr);
            if (pa.page_present) (*am_map)[pa.pfn]++;
        }
    if (c >= uthreshold) topReached = true;
    ReleaseLock(&lock);
}

// Print a memory write record when thread separated files are required
VOID RecordMemWriteTh(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                         ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; wrts++;
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_wrts++;
    UINT64 c;
    if (KnobRelThres) c = tdata->_icount;
    else c = tdata->_rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobWrite && thPrint(tid))
```

```
        if (((c - lthreshold) % sampling) == 0)
        {
            char line[256] = "\0";
            if (KnobIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                    (unsigned long long)icount);
            if (KnobMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                  (unsigned long long)rds_wrts);
            if (KnobThIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                       (unsigned long long)tdata->_icount);
            if (KnobThMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                     (unsigned long long)tdata->_rds_wrts);
            if (KnobTimestamp) sprintf(strchr(line,0), tsFormat.c_str(),
                                                              get_ts());
            if (KnobThreadid) sprintf(strchr(line,0), tidFormat.c_str(), tid);
            if (KnobIaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                    (unsigned long long)ip);
            if (KnobTypeop) sprintf(strchr(line,0), "W ");
            if (KnobNrops) sprintf(strchr(line,0), opsFormat.c_str(), ops);
            if (KnobOpsize) sprintf(strchr(line,0), opsFormat.c_str(), sz);
            if (KnobVaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                     (unsigned long long)addr);
            if (KnobNcacheline) sprintf(strchr(line,0), addrFormat.c_str(),
                                  (unsigned long long)get_cacheline(addr));
            if (KnobNset) sprintf(strchr(line,0), nsetFormat.c_str(),
                                                          get_nset(addr));
            if (KnobNpage) sprintf(strchr(line,0), pageFormat.c_str(),
                                       (unsigned long long)get_page(addr));
            if (KnobPaddr) sprintf(strchr(line,0), "%s",
                   get_physical_address(addr).c_str());
            char* last = line + strlen(line) - 1;
            if (*last == ' ') *last = '\0';
            if (KnobGzip)
            {
                gzprintf(tdata->_trace, "%s\n", line);
                gzflush(tdata->_trace, Z_SYNC_FLUSH);
            }
            else
            {
                fprintf(tdata->_trace, "%s\n", line);
                fflush(tdata->_trace);
            }
        }
    if ((c >= uthreshold) && thPrint(tid)) topReached = true;
    ReleaseLock(&lock);
}

// Print a memory write record
VOID RecordMemWrite(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                        ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; wrts++;
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_wrts++;
    UINT64 c;
```

```
    if (KnobRelThres) c = icount;
    else c = rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobWrite && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            char line[256] = "\0";
            if (KnobIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                  (unsigned long long)icount);
            if (KnobMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                                 (unsigned long long)rds_wrts);
            if (KnobThIcount) sprintf(strchr(line,0), countFormat.c_str(),
                                       (unsigned long long)tdata->_icount);
            if (KnobThMcount) sprintf(strchr(line,0), countFormat.c_str(),
                                    (unsigned long long)tdata->_rds_wrts);
            if (KnobTimestamp) sprintf(strchr(line,0), tsFormat.c_str(),
                                                                 get_ts());
            if (KnobThreadid) sprintf(strchr(line,0), tidFormat.c_str(), tid);
            if (KnobIaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                    (unsigned long long)ip);
            if (KnobTypeop) sprintf(strchr(line,0), "W ");
            if (KnobNrops) sprintf(strchr(line,0), opsFormat.c_str(), ops);
            if (KnobOpsize) sprintf(strchr(line,0), opsFormat.c_str(), sz);
            if (KnobVaddr) sprintf(strchr(line,0), addrFormat.c_str(),
                                                     (unsigned long long)addr);
            if (KnobNcacheline) sprintf(strchr(line,0), addrFormat.c_str(),
                                   (unsigned long long)get_cacheline(addr));
            if (KnobNset) sprintf(strchr(line,0), nsetFormat.c_str(),
                                                           get_nset(addr));
            if (KnobNpage) sprintf(strchr(line,0), pageFormat.c_str(),
                                       (unsigned long long)get_page(addr));
            if (KnobPaddr) sprintf(strchr(line,0), "%s",
                                      get_physical_address(addr).c_str());
            char* last = line + strlen(line) - 1;
            if (*last == ' ') *last = '\0';
            if (KnobGzip)
            {
                gzprintf(trace, "%s\n", line);
                gzflush(trace, Z_SYNC_FLUSH);
            }
            else
            {
                fprintf(trace, "%s\n", line);
                fflush(trace);
            }
        }
    if (c >= uthreshold) topReached = true;
    ReleaseLock(&lock);
}

// Count a memory write when thread separated files are required
VOID CountMemWriteTh(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                       ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; wrts++;
```

```
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++;
 tdata->_wrts++;
    UINT64 c;
    if (KnobRelThres) c = tdata->_icount;
    else c = tdata->_rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobWrite && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            paddr_t pa = get_physical_address(addr);
            if (pa.page_present) (*tdata->_am_map)[pa.pfn]++;
        }
    if ((c >= uthreshold) && thPrint(tid)) topReached = true;
    ReleaseLock(&lock);
}

// Count a memory write
VOID CountMemWrite(ADDRINT ip, UINT32 sz, UINT32 ops, THREADID tid,
                                                       ADDRINT addr)
{
    GetLock(&lock, tid+1);
    rds_wrts++; wrts++;
    thread_data_t* tdata = get_tls(tid);
    tdata->_rds_wrts++; tdata->_wrts++;
    UINT64 c;
    if (KnobRelThres) c = icount;
    else c = rds_wrts;
    if ((c >= lthreshold) && (c <= uthreshold) && KnobWrite && thPrint(tid))
        if (((c - lthreshold) % sampling) == 0)
        {
            paddr_t pa = get_physical_address(addr);
            if (pa.page_present) (*am_map)[pa.pfn]++;
        }
    if (c >= uthreshold) topReached = true;
    ReleaseLock(&lock);
}

// This routine is executed every time a thread is created.
VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    GetLock(&lock, threadid+1);
    numThreads++;
    ReleaseLock(&lock);

    thread_data_t* tdata = new thread_data_t;

    PIN_SetThreadData(tls_key, tdata, threadid);

      /******   Code added by Jose Antonio Martin    ******/
      if ( KnobUseModuleAddressTranslator || KnobCompareMethods )
    {
            int pid = PIN_GetPid();
          int ret = ioctl(module, IOCTL_ENVIAR_PID, pid);
            if ( ret < 0 )
            {
```

```
                printf("Access Error to module 'Module Address Translator' \n");
                  printf("Error code: %d \n" , ret);
                  exit(-1);
            }
            else
            {
                printf("PID process:  %d \n" , pid);
            }
      }
      /**************************************************/

    if (KnobThFiles && thPrint(threadid))
    {
        // Open output file
        string fname = KnobOutputFile.Value();
        if (KnobUniqueOutputFile) fname = fname + "." + decstr(PIN_GetPid());
        if (KnobGzip)
        {
            fname = fname + "." + decstr(threadid) + ".gz";
            tdata->_trace = (FILE*)gzopen(fname.c_str(), "wb9");
            if (KnobStd) printf("Output file: %s\n", fname.c_str());
            LOG(string("Output file: ") + fname.c_str() + "\n");
        }
        else
        {
            fname = fname + "." + decstr(threadid);
            tdata->_trace = fopen(fname.c_str(), "w");
            if (KnobStd) printf("Output file: %s\n", fname.c_str());
            LOG(string("Output file: ") + fname.c_str() + "\n");
        }
        if (tdata->_trace == NULL)
        {
            if (KnobStd) printf("Unable to open output file for writing.\n");
            LOG("Unable to open output file for writing.\n");
            exit(1);
        }
        // Print headers
        print_header(tdata->_trace);
        // Create thread map in analytic mode
        if (KnobAnalyticMode)
        {
            tdata->_am_map = new map<UINT64, UINT64>;
        }
    }
}

// This routine is executed every time a thread is destroyed.
VOID ThreadFini(THREADID threadid, const CONTEXT *ctxt, INT32 code, VOID *v)
{
    if (KnobThFiles && thPrint(threadid))
    {
        thread_data_t* tdata = get_tls(threadid);
        if (KnobGzip)
        {
            if (KnobAnalyticMode) print_am_map_th(tdata);
```

```
            gzprintf(tdata->_trace, "#eof\n");
            gzclose(tdata->_trace);
        }
        else
        {
            if (KnobAnalyticMode) print_am_map_th(tdata);
            fprintf(tdata->_trace, "#eof\n");
            fclose(tdata->_trace);
        }
    }
}

// Called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    if (!KnobThFiles)
    {
        if (KnobAnalyticMode) print_am_map();
        // Print last line and close output file
        if (KnobGzip)
        {
                gzprintf(trace, "#eof\n");
                gzclose(trace);
        }
        else
        {
                fprintf(trace, "#eof\n");
                fclose(trace);
        }
    }
    // Print counters to stdout and logfile
    print_counters();
    /* Close pagemap file if open */
    if (pagemap != -1)
    {
        close(pagemap);
    }
}

// Function called for every instruction and instruments reads and writes
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction.
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_THREAD_ID,
                                                         IARG_END);

    // Instruments memory accesses using a predicated call, instrumentation
    // is called if the instruction will actually be executed.
    //
    // The IA-64 architecture has explicitly predicated instructions.
    // On the IA-32 and Intel(R) 64 architectures conditional moves and REP
    // prefixed instructions appear as predicated instructions in Pin.
    UINT32 memOperands = INS_MemoryOperandCount(ins);

    // Iterate over each memory operand of the instruction.
```

```
for (UINT32 memOp = 0; memOp < memOperands; memOp++)
{
    if (INS_MemoryOperandIsRead(ins, memOp))
    {
        if (!KnobAnalyticMode)
        {
            if (KnobThFiles)
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemReadTh,
                    IARG_INST_PTR,
                    IARG_MEMORYREAD_SIZE,
                    IARG_UINT32, memOperands,
                    IARG_THREAD_ID,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_END);
            }
            else
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                    IARG_INST_PTR,
                    IARG_MEMORYREAD_SIZE,
                    IARG_UINT32, memOperands,
                    IARG_THREAD_ID,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_END);
            }
        }
        else
        {
            if (KnobThFiles)
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)CountMemReadTh,
                    IARG_INST_PTR,
                    IARG_MEMORYREAD_SIZE,
                    IARG_UINT32, memOperands,
                    IARG_THREAD_ID,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_END);
            }
            else
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)CountMemRead,
                    IARG_INST_PTR,
                    IARG_MEMORYREAD_SIZE,
                    IARG_UINT32, memOperands,
                    IARG_THREAD_ID,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_END);
            }
        }
    }
```

```
// Note that in some architectures a single memory operand can be
// both read and written (for instance incl (%eax) on IA-32)
// In that case we instrument it once for read and once for write.
if (INS_MemoryOperandIsWritten(ins, memOp))
{
    if (!KnobAnalyticMode)
    {
        if (KnobThFiles)
        {
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWriteTh,
                IARG_INST_PTR,
                IARG_MEMORYWRITE_SIZE,
                IARG_UINT32, memOperands,
                IARG_THREAD_ID,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
        }
        else
        {
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
                IARG_INST_PTR,
                IARG_MEMORYWRITE_SIZE,
                IARG_UINT32, memOperands,
                IARG_THREAD_ID,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
        }
    }
    else
    {
        if (KnobThFiles)
        {
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)CountMemWriteTh,
                IARG_INST_PTR,
                IARG_MEMORYWRITE_SIZE,
                IARG_UINT32, memOperands,
                IARG_THREAD_ID,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
        }
        else
        {
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)CountMemWrite,
                IARG_INST_PTR,
                IARG_MEMORYWRITE_SIZE,
                IARG_UINT32, memOperands,
                IARG_THREAD_ID,
                IARG_MEMORYOP_EA, memOp,
                IARG_END);
        }
    }
```

```
        }
        // Exit pintool if reached upper thresholds
        if (topReached && KnobForceExit)
        {
            char logFx[64];
            sprintf(logFx, "Exiting due to %llu upper threshold reached.\n",
                                              (unsigned long long)uthreshold);
            if (KnobStd) printf("%s", logFx);
            LOG(logFx);
            for (int t = (numThreads - 1); t >= 0; t--)
                ThreadFini(t,0,0,0);
            Fini(0,0);
            exit(0);
        }
    }
}

/* ===================================================================== */
/* Print Help Message
                */
/* ===================================================================== */

INT32 Usage()
{
    cerr << "This Pintool prints a file with a memory access trace of" << endl;
    cerr << "a multi-threaded application." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */

int main(int argc, char *argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

      /******   Code added by Jose Antonio Martin    ******/
      // Open the module access
      if ( KnobUseModuleAddressTranslator || KnobCompareMethods )
      {
          module = open(DEVICE_FILE_NAME, 0);
            if ( module < 0 )
            {
                printf("Error opening module 'Module Address Translator' \n");
                  printf("Error code: %d \n" , module);
                  exit(-1);
            }
    }
      /************************************************/

    // Print and log Pintool name
    if (KnobStd) printf("Mtaptrace Pintool starting...\n");
```

```
    LOG("Mtaptrace Pintool starting...\n");

    // Set print formats for printf
    set_print_formats();

    // Check thresholds and sampling
    if (check_threshold_sampling() == 0) return 0;

    // Check threads to print
    if (check_thlist() == 0) return 0;

    // Check page size
    if (KnobNpage || KnobPaddr || KnobAnalyticMode)
        if (get_page_size() == 0) return 0;

    // Check cache info
    if (KnobNcacheline || KnobNset)
        if (get_cache_info() == 0) return 0;

    // Open pagemap file for read if required
    if (KnobPaddr || KnobAnalyticMode)
    {
        char n_pagemap[256];
        //sprintf(n_pagemap, "/proc/self/pagemap");
        sprintf(n_pagemap, "/proc/%d/pagemap", PIN_GetPid());
        pagemap = open(n_pagemap, O_RDONLY);
        if (KnobStd) printf("Pagemap file: %s\n", n_pagemap);
        LOG(string("Pagemap file: ") + n_pagemap + "\n");
        if (pagemap == -1)
        {
            if (KnobStd) printf("Unable to open pagemap file for reading.\n");
            LOG("Unable to open pagemap file for reading.\n");
            return 0;
        }
    }

    if (!KnobThFiles)
    {
        // Open output file
        string fname = KnobOutputFile.Value();
        if (KnobUniqueOutputFile) fname = fname + "." + decstr(PIN_GetPid());
        if (KnobGzip)
        {
            fname = fname + ".gz";
            trace = (FILE*)gzopen(fname.c_str(), "wb9");
            if (KnobStd) printf("Output file: %s\n", fname.c_str());
            LOG(string("Output file: ") + fname.c_str() + "\n");
        }
        else
        {
            trace = fopen(fname.c_str(), "w");
            if (KnobStd) printf("Output file: %s\n", fname.c_str());
            LOG(string("Output file: ") + fname.c_str() + "\n");
        }
        if (trace == NULL)
```

```
        {
            if (KnobStd) printf("Unable to open output file for writing.\n");
            LOG("Unable to open output file for writing.\n");
            return 0;
        }
        // Print header to output file
        print_header(trace);
        // Create global array in analytic mode
        if (KnobAnalyticMode)
        {
            am_map = new map<UINT64, UINT64>;
        }
    }
    // Initialize the pin lock
    InitLock(&lock);

    // Obtain  a key for TLS storage.
    tls_key = PIN_CreateThreadDataKey(0);

    // Register ThreadStart to be called when a thread starts.
    PIN_AddThreadStartFunction(ThreadStart, 0);

    // Register ThreadFini to be called when a thread exits.
    PIN_AddThreadFiniFunction(ThreadFini, 0);

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Print and log program start
    if (KnobStd) printf("Program starting...\n");
    LOG("Program starting...\n");

    // Start program, never returns
    PIN_StartProgram();

      /******   Code added by Jose Antonio Martin    ******/
      // Close access "Module Address Translator"
      if ( KnobUseModuleAddressTranslator || KnobCompareMethods )
      {
          close(module);
      }
    /*************************************************/

    return 0;
}
```