Reference Manual of Impulse System Calls

Lixin Zhang, Leigh Stoller

UUCS-99-018

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

January 20, 1999

Abstract

This document describes the Impulse system calls. The Impulse system calls allow user applications to use remapping functionality provided by the Impulse Adaptive Memory System to reorganize their data structures without actually moving data around the physical memory. Impulse supports several remapping algorithms. User applications choose the desired remapping algorithms by calling the right Impulse system calls. This note uses detailed examples to illustrate each Impulse system call.

This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

Contents

1 Introduction

This document describes the Impulse system calls. The Impulse system calls and their related data structures are defined in the following header file.

```
/nfs/flux/impsrc/dist/simpulse/app-lib/include/amssup.h
```

User applications should include this header file and link with the following object file to pick up the Impulse system calls.

```
/nfs/flux/impsrc/dist/simpulse/app-lib/libkernel.o
```

Currently, the Impulse memory system supports the following remappings: **no-copy superpage formation**, **strided remapping**, **transpose remapping**, **no-copy page coloring**, and **scatter/gather through an indirection vector**. Depending on what kind of values are stored in the indirection vector and how the indirection vector is created, **scatter/gather through an indirection vector** is further split into three sub-types: **scatter/gather through an index vector**, if the indirection vector stores indices to an array; **scatter/gather through an offset vector**, if the indirection vector stores byte offsets; and **dynamic cacheline assembly**, if the indirection vector is dynamically created by the OS.

2 Impulse System Calls

The section describes four Impulse system calls:

- `ams_mapshadow()` sets up remappings and is the primary interface for applications to access Impulse features;
- `ams_remapshadow()` allows users to adjust existing remappings previously set up by `ams_mapshadow()`;
- `ams_allocvirt()` allocates virtual memory, used in conjunction with `ams_mapvtov()`;
- `ams_mapvtov()` maps a set of virtual addresses to a specified shadow region, used in conjunction with `ams_allocvirt()` to optimize the layout of shadow data in virtually indexed caches.

2.1 ams_mapshadow()

```
int
```

```
ams_mapshadow(int        type,
              void      *datablock);
```

`ams_mapshadow()` is the main Impulse system call. Application programs use this system call to set up remappings. The argument `type` specifies which type of remapping to set up. The argument `datablock` is the address of a type-specific data structure. The `type` can be one of the followings:

- `AMS_TYPE_SUPERPAGE`: no-copy superpage formation;
- `AMS_TYPE_BASESTRIDE`: strided scatter/gather remapping;
- `AMS_TYPE_TRANSPOSE`: transpose remapping;
- `AMS_TYPE_PAGECOLOR`: no-copy page coloring;
- `AMS_TYPE_VINDIRECT`: scatter/gather through an index vector;
- `AMS_TYPE_VINDIRECT2`: scatter/gather through an offset vector;
- `AMS_TYPE_VINDIRECT3`: dynamic cacheline assembly.

2.1.1 ams_mapshadow(AMS_TYPE_SUPERPAGE, superpage_args_t *datablock)

`ams_mapshadow(AMS_TYPE_SUPERPAGE, datablock)` sets up a remapping of **no-copy superpage formation**. The argument `datablock` points to a `superpage_args_t` structure defined as the following:

```
typedef struct {
        vaddr_t   vaddr;
        int       size;
        int       prefcount;
        int       prefinfo;
} superpage_args_t;
```

This call maps the virtual memory region, starting at address `vaddr` with `size` bytes in length, to an equally-sized contiguous shadow memory region. The Impulse MMC is responsible for translating the shadow memory region back to the physical pages to which the original virtual memory region maps. Both `vaddr` and `size` must be page-aligned.

`prefetch` and `prefinfo` contain information about how to perform prefetching inside the shadow region: `prefcount` is the number of blocks to prefetch each time; and `prefinfo` is the prefetch distance.

For example, assuming that `prefcount` equals 2 and `prefinfo` equals -3 (a negative value means prefetching backwards), when the memory controller receives a load request for block **A**, it will prefetch two blocks — block ($\mathbf{A} - 3$) and block ($\mathbf{A} - 6$).

Since the remapped region is contiguous in both virtual memory and shadow memory after remapping, it can use superpages to reduce the number of TLB entries required to map it. This call converts the virtual memory region to TLB superpages, with the largest possible superpages allocated, based on the alignment of `vaddr`. Because superpages must be aligned to their sizes, superpages are allocated by walking the virtual address region and assigning the largest possible superpage restricted by the current alignment. For example, if the current address is 16Kbyte aligned, a 16Kbyte superpage can be assigned. The address is then incremented; and the new address alignment is checked. This procedure proceeds until the end of the region is reached.

Superpage sizes are powers of 2 multiple of base pages ranging from 8K to 4M bytes in size. The base page size is 4K bytes. In practice, the first few and last few pages are often base pages, with the pages in the middle being larger. For example, a virtual memory region at 0x00039000 with 0x100000 bytes will be converted to 9 pages in size of 4K, 8K, 16K, 256K, 512K, 128K, 64K, 32K, and 4K bytes respectively. This conversion reduces the number of TLB entries required for this region from 256 to 9.

`ams_mapshadow()` returns 0 on success and -1 otherwise.

ERRORS:

`EINVAL` Either `vaddr` or `size` is not page-aligned.

Figure 1 shows a simple example of using `ams_mapshadow(AMS_TYPE_SUPERPAGE, ...)`. The example contains both the Impulse version and the non-Impulse version. The Impulse version has `IMPULSE` defined while the non-Impulse version has not.

2.1.2 ams_mapshadow(AMS_TYPE_BASESTRIDE, basestride_args_t *datablock)

`ams_mapshadow(AMS_TYPE_BASESTRIDE, datablock)` sets up a **strided scatter/gather remapping**. The argument `datablock` points to a `basestride_args_t` structure defined as the following:

```
typedef struct {
        vaddr_t  *newaddr;
        vaddr_t   vaddr;
        int       count;
        int       objsize;
        int       stride;
        int       offset;
```

```
/*
 * Creates superpages for array A[count]
 */

#define PAGESIZE        0x1000

double foo(double *A, int count)
{
   int     i;
   double  sum = 0;
   superpage_args_t sp_args;

#ifdef IMPULSE
   sp_args.vaddr     = (vaddr_t) A;
   sp_args.size      = count * sizeof(double);
   sp_args.prefcount = 0;
   sp_args.prefinfo  = 0;

   if (ams_mapshadow(AMS_TYPE_SUPERPAGE, &sp_args) < 0) {
      printf("ams_mapshadow(AMS_TYPE_SUPERPAGE, ...), failed\n");
      exit(1);
   }
#endif

   for (i = 0; i < count; i += PAGESIZE/sizeof(double))
      sum += A[i];

   return sum;
}
```

Figure 1: A code fragment using `ams_mapshadow(AMS_TYPE_SUPERPAGE, ...)`.

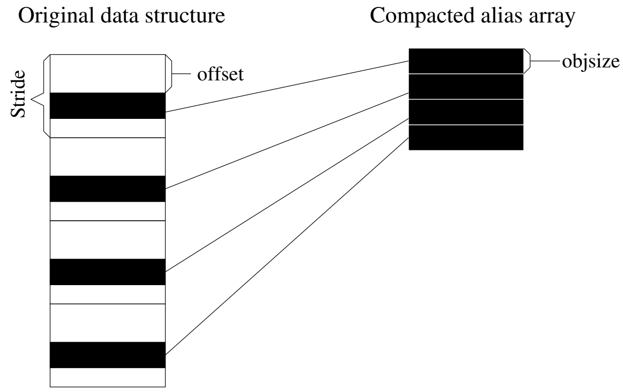

Figure 2: Visualize strided remapping.

```
        int        prefcount;
        int        prefinfo;
        int        salign;
        int        valign;
} basestride_args_t;
```

This call compacts data items in strided memory locations into a dense alias array, as shown by Figure 2.

`vaddr` is the starting address of the data structure being remapped and must be page-aligned. The data structure has `count` strides, each of which is `stride` bytes in length. Each required data element is `objsize` bytes in length; and its byte offset from the base of the stride is given by `offset`. `prefcount` is the number of blocks to prefetch each time and `prefinfo` is the prefetch stride.

The OS first allocates a shadow region for the alias array storing compacted data items. The shadow region has (`count` $\times$ `objsize`) bytes. The OS then allocates a new virtual region and map it to the shadow region. The starting address of the new virtual region is stored at a location in the application address space pointed to by `newaddr`. `salign` specifies the expected alignment of the new shadow region and determines where the alias array will be mapped to in physically index caches (like most L2 caches in modern microarchitectures); and `valign` specifies the expected alignment of the new virtual region and determines where the alias array will be mapped to in virtually index caches (like most L1 caches in modern microarchitectures). Both `salign` and `valign` must be page-aligned.

`ams_mapshadow()` returns 0 on success, and -1 otherwise. If successful, the virtual address of the alias array is placed into the memory location pointed to by `newaddr`.

ERRORS:

`EINVAL`	Either `vaddr` or `salign` or `valign` is not page-aligned.
`EFAULT`	`newaddr` specifies an invalid address.

Figure 3 shows a simple example of using `ams_mapshadow(AMS_TYPE_BASESTRIDE, ...)`. The example contains both the Impulse version and the non-Impulse version. The Impulse version has `IMPULSE` defined while the non-Impulse version has not.

2.1.3 ams_mapshadow(AMS_TYPE_TRANSPOSE, transpose_args_t *datablock)

`ams_mapshadow(AMS_TYPE_TRANSPOSE, datablock)` sets up a **transpose remapping**. The argument `datablock` points to a `transpose_args_t` structure defined as the following:

```
typedef struct {
        vaddr_t  *newaddr;
        vaddr_t   vaddr;
        int       elemsize;
        int       rownum;
        int       rowsize;
        int       prefcount;
        int       prefinfo;
        int       salign;
        int       valign;
} transpose_args_t;
```

This system call maps a two-dimensional matrix to its transpose without copying.

`newaddr` is a pointer to a location in the application address space where the kernel can store the return value. `vaddr` is the virtual address of a two-dimensional matrix and must be page-aligned. `elemsize` gives the size of matrix element in bytes. `rownum` gives the number of rows that the two-dimensional matrix has. `rowsize` gives the size of each row in bytes. (Thus, the matrix has `rowsize`/`elemsize` columns.) `prefcount` is the number of blocks to prefetch each time and `prefinfo` is the prefetch distance, as described in Section 2.1.1.

The return value of this function is the virtual address of a new matrix – the transpose of the original matrix. `salign` specifies the expected alignment of the new matrix's shadow region; and `valign` specifies the expected alignment of the new matrix's virtual region. They determine where the new matrix will be mapped to in the caches. Both of them must be page-aligned.

`ams_mapshadow()` returns 0 on success, and -1 otherwise. If successful, the virtual address of the new matrix is placed into the memory location pointed to by `newaddr`.

ERRORS:

`EINVAL` Either `vaddr` or `salign` or `valign` is not page-aligned.

```
/*
 * Compute the sum of A[8*i+2], where i is from 0 to (size/8 - 1).
 */
float foo(float *A, int size)
{
   basestride_args_t  bs_args;
   float              *Anew;
   float              sum  = 0;
   int                step = 8;

#ifdef IMPULSE
   bs_args.newaddr   = (vaddr_t *) &Anew;
   bs_args.vaddr     = (vaddr_t) A;
   bs_args.count     = size / step;
   bs_args.objsize   = sizeof(float);
   bs_args.stride    = sizeof(float) * step;
   bs_args.offset    = sizeof(float) * 2;
   bs_args.prefcount = 1; /* prefetch one block each time */
   bs_args.prefinfo  = 1; /* prefetch forward */
   bs_args.salign    = 0x4000;  /* Ai[0] to offset 0x4000 in L2C */
   bs_args.valign    = 0x2000;  /* Ai[0] to offset 0x2000 in L1C */

   if (ams_mapshadow(AMS_TYPE_BASESTRIDE, &bs_args) == -1) {
      perror("ams_mapshadow(AMS_TYPE_BASESTRIDE, ...) failed.");
      exit(1);
   }
#endif

   for (i = 0; i < size / step; i++) {
#ifdef IMPULSE
      sum += Anew[i];
#else
      sum += A[i * 8 + 2];
#endif
   }

   return sum;
}
```

Figure 3: A code fragment using `ams_mapshadow(AMS_TYPE_BASESTRIDE, ...)`.

`EFAULT` `newaddr` specifies an invalid address.

Figure 4 shows a simple example of using `ams_mapshadow(AMS_TYPE_TRANSPOSE, ...)`. The example contains both the Impulse version and the non-Impulse version. The Impulse version has `IMPULSE` defined while the non-Impulse version has not.

2.1.4 ams_mapshadow(AMS_TYPE_PAGECOLOR, pagecolor_args_t *datablock)

`ams_mapshadow(AMS_TYPE_PAGECOLOR, datablock)` sets up a remapping of **no-copy page coloring**. The argument `datablock` points to a `pagecolor_args_t` structure defined as the following:

```
typedef struct {
        vaddr_t   vaddr;
        int       size;
        int       waysize;
        int       colorfactor;
        int       colorid;
        int       prefcount;
        int       prefinfo;
} pagecolor_args_t;
```

This call sets up a remapping for a specified virtual region so that the whole region will be mapped to only a designated portion of a physically indexed cache. Figure 5 shows how to use page coloring to map a data structure to the third quadrant of a physically indexed L2 cache.

`vaddr` points to the virtual region being remapped and must be page-aligned. `size` gives the size of the virtual region in bytes. `waysize` gives the way size of targeted physically indexed cache, which equals cache size divided by its associativity. `colorfactor` is number of colors that the cache is split into. `colorid` is the index of the color to which the virtual region will solely map. In figure 5, `colorfactor` is 4 and `colorid` is 2. `prefcount` is the number of blocks to prefetch each time and `prefinfo` is the prefetch distance.

`ams_mapshadow()` returns 0 on success, and -1 otherwise.

ERRORS:

`EINVAL` `vaddr` is not page-aligned.

Figure 6 shows a simple example of using `ams_mapshadow(AMS_TYPE_PAGECOLOR, ...)`. The example contains both the Impulse version and the non-Impulse version. The Impulse version has `IMPULSE` defined while the non-Impulse version has not.

```
/*
 * Dense matrix-matrix multiplication: C = A * B.
 * A, B, and C are (size x size) matrices.
 */
foo(double *A, double *B, double *C, int size)
{
   transpose_args_t  tr_args;
   double            *Bnew, sum;
   int               i, j, k;

#ifdef IMPULSE
   tr_args.newaddr   = (vaddr_t *) &Bnew;
   tr_args.vaddr     = (vaddr_t) x;
   tr_args.elemsize  = sizeof(double);
   tr_args.rownum    = size;
   tr_args.rowsize   = sizeof(double) * size;
   tr_args.prefcount = 1;
   tr_args.prefinfo  = 1;
   tr_args.salign    = 0; /* don't care */
   tr_args.valign    = 0; /* don't care */

   if (ams_mapshadow(AMS_TYPE_TRANSPOSE, &tr_args) == -1) {
      perror("ams_mapshadow(AMS_TYPE_TRANSPOSE, ...) failed.");
      exit(1);
   }
#endif

   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++) {
         for (sum = 0, k = 0; k < size; k++)
#ifdef IMPULSE
            sum += A[i][k] * Bnew[j][k];
#else
            sum += A[i][k] * B[k][j];
#endif
         C[i][j] = sum;
      }
}
```

Figure 4: A code fragment using `ams_mapshadow(AMS_TYPE_TRANSPOSE, ...)`.

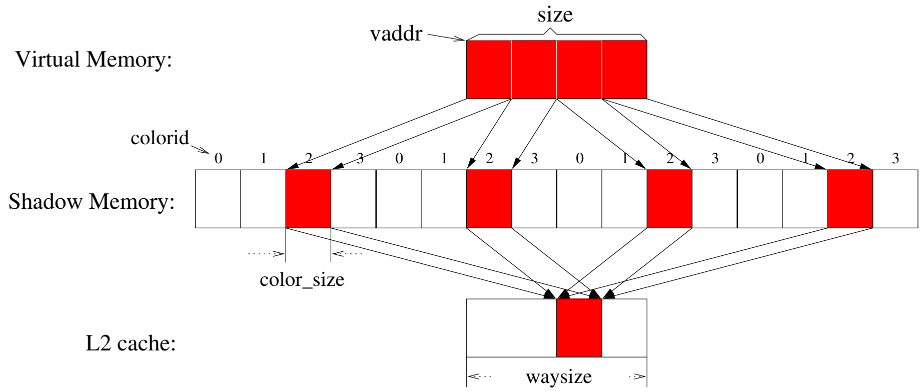

Figure 5: Mapping a data structure to the third quadrant of a physically indexed L2 cache. In this example, `colorfactor` is 4; `colorid` is 2. Note *color_size* is `waysize` divided by `colorfactor`.

2.1.5 ams_mapshadow(AMS_TYPE_VINDIRECT, vindirect_args_t *datablock)

`ams_mapshadow(AMS_TYPE_VINDIRECT, datablock)` sets up a remapping of **scatter/gather through an index vector**, a special case of scatter/gather through an indirection vector when the indirection vector stores array indices. The argument `datablock` points to a `vindirect_args_t` structure defined as the following:

```
typedef struct {
        vaddr_t  *newaddr;
        vaddr_t   vaddr;
        int       count;
        int       objsize;
        vaddr_t   iv_vaddr;
        int       iv_objcount;
        int       iv_objsize;
        int       isfortran;
        int       iv_subtype;
        int       maxcount;
        int       prefcount;
        int       prefinfo;
        int       salign;
        int       valign;
} vindirect_args_t;
```

This call sets up a region of shadow addresses mapped to a one-dimensional array through an indirection vector. That is, a shadow address at offset *soffset* in a shadow region is mapped to data item *vector[soffset]* in physical memory.

```
/*
 * Map A to the second half of L2 cache and B to the first half
 * to avoid A[i] and B[i] mapping to the same line of
 * a two-way set-associative, 256Kbytes L2 cache.
 */
foo(double *A, double *B, int size)
{
   int      i;
   double   sum = 0;

#ifdef IMPULSE
   color_array(A, size * sizeof(double), 2, 1);
   color_array(B, size * sizeof(double), 2, 0);
#endif

   for (i = 0; i < SIZE; i++)
      sum += A[i] + B[i];
}

#ifdef IMPULSE
color_array(void *x, int size, int colorfactor, int colorid)
{
   pagecolor_args_t args;

   args.vaddr       = (vaddr_t) x;
   args.size        = size;
   args.waysize     = 128 * 1024;   /* 256Kbytes/2-way */
   args.colorfactor = colorfactor;
   args.colorid     = colorid;
   args.prefcount   = 1;
   args.prefinfo    = 1;

   if (ams_mapshadow(AMS_TYPE_PAGECOLOR, &args) == -1) {
      printf("ams_mapshadow_pagecolor failed\n");
      exit(1);
   }
}
#endif
```

Figure 6: A code fragment using `ams_mapshadow(AMS_TYPE_PAGECOLOR, ...)`.

newaddr is be a pointer to a location in the application address space where the kernel can store the return value. vaddr is the starting virtual address of the original one-dimensional array and must be page-aligned. The array contains count elements and each element is objsize bytes in length. iv_vaddr is the starting virtual address of the indirection vector. The indirection vector contains iv_count elements and each element is iv_objsize bytes in length. isfortran indicates whether or not the indirection vector stores Fortran-style array subscripts, i.e., subscripts starting from 1, not 0 as C/C++ does. iv_subtype represents subtype of this remapping. In current implementation, iv_subtype should be 0 if iv_count equals maxcount or be 1 if iv_count is less than maxcount. prefcount is the number of blocks to prefetch each time and prefinfo is the prefetch distance.

The return value of this function is the virtual address of a new vector. The new vector has maxcount elements. When maxcount is larger than iv_count, the indirection vector will be reused, in the sense that the 0th and iv_countth element of the new vector both use the 0th element of the indirection vector. salign specifies the expected alignment of the new vector's shadow region; and valign specifies the expected alignment of the new vector's virtual region. They determine where the new vector will be mapped to in the caches. Both of them must be page-aligned.

ams_mapshadow() returns 0 on success, and -1 otherwise. If successful, the virtual address of the new vector is placed into the memory location pointed to by newaddr.

ERRORS:

EINVAL	Either vaddr or iv_vaddr or salign or valign is not page-aligned.
EFAULT	newaddr specifies an invalid address.

Figure 7 shows a simple example of using ams_mapshadow(AMS_TYPE_VINDIRECT, ...). The example contains both the Impulse version and the non-Impulse version. The Impulse version has IMPULSE defined while the non-Impulse version has not.

2.1.6 ams_mapshadow(AMS_TYPE_VINDIRECT2, vindirect2_args_t *datablock)

ams_mapshadow(AMS_TYPE_VINDIRECT2, datablock) sets up a remapping of **scatter/gather through an offset vector**, a special case of scatter/gather through an indirection vector when the indirection vector stores byte offsets. The argument datablock points to a vindirect2_args_t structure defined as the following:

```
typedef struct {
        vaddr_t         offset;
        vaddr_t         size;
} ATTRIB;
#define MAX_ATTR        8
```

```
/*
 * sum = A x P, where A is a sparse matrix and P is a dense vector.
 * The original code is extracted from CG of NPB2.3.
 */
foo(double *A, double *P, int *colidx, int *rows,
    int acount, int pcount, int rowcount, double *sum)
{
   vindirect_args_t  vi_args;
   double           *pnew;
   int               i, k;

#ifdef IMPULSE
   vi_args.newaddr    = (vaddr_t *) &pnew;
   vi_args.vaddr      = (vaddr_t) P;
   vi_args.count      = pcount;
   vi_args.objsize    = sizeof(double);
   vi_args.iv_vaddr   = colidx;
   vi_args.iv_count   = acount;
   vi_args.iv_objsize = sizeof(int);
   vi_args.isfortran  = 1; /* CG is Fortran code */
   vi_args.maxcount   = acount;
   vi_args.prefcount  = 1;
   vi_args.prefinfo   = 1;
   vi_args.salign     = 0; /* don't care */
   vi_args.valign     = 0; /* don't care */

   if (ams_mapshadow(AMS_TYPE_VINDIRECT, &vi_args) == -1) {
      perror("ams_mapshadow(AMS_TYPE_VINDIRECT, ...) failed.");
      exit(1);
   }
#endif

   for (i = 0; i < rowcount; i++) {
      for (k = rows[i]; k < rows[i+1]; k++)
#ifdef IMPULSE
         sum[k] = A[k] * pnew[k];
#else /* Non-Impulse version */
         sum[k] = A[k] * p[colidx[k]];
#endif
   }
}
```

Figure 7: A code fragment using `ams_mapshadow(AMS_TYPE_VINDIRECT, ...)`.

```
typedef struct {
        vaddr_t  *newaddr;
        vaddr_t   vaddr;
        int       count;
        int       size;
        vaddr_t   iv_vaddr;
        int       iv_objsize;
        int       prefcount;
        int       prefinfo;
        int       attribs_num;
        ATTRIB    attribs[MAX_ATTR];
} vindirect2_args_t;
```

This system call was specifically designed for the PostSQL database management program. The main data structures of PostSQL are *active pages*. Each active page has the same format: a small header, followed by an offset vector, followed by database records. The last attribute of a database record varies in length, which make the database records vary in size too. The offset vector stores each record's byte offset from the base of the active page. This call maps the required attributes of database records in an active page into a dense shadow region.

`newaddr` is a pointer to a location in the application address space where the kernel can store the return value. `vaddr` is the virtual address of an active page and must be page-aligned. `size` is the number of bytes in the active page. `count` is the number of database records in the active page. `iv_vaddr` and `iv_objsize` are the virtual address and element size of the offset vector in the active page. `attribs_num` is the number of attributes to be gathered in each database record, with eight as its maximum value. The offset and size of each required attribute are stored in array `attribs[]`. `prefcount` is the number of blocks to prefetch each time and `prefinfo` is the prefetch distance.

The return value of this function is the virtual address of a new vector. Each element of this vector contains all gathered attributes of a database record.

`ams_mapshadow()` returns 0 on success, and -1 otherwise. If successful, the virtual address of the new vector is placed into the memory location pointed to by `newaddr`.

ERRORS:

`EINVAL`	`vaddr` is not page-aligned.
`EFAULT`	`newaddr` specifies an invalid address.

Figure 8 shows a simple example of using `ams_mapshadow(AMS_TYPE_VINDIRECT2, ...)`. The example contains both the Impulse version and the non-Impulse version. The Impulse version has `IMPULSE` defined while the non-Impulse version has not.

```
typedef struct {
   int    a;
   float  b;
   int    c;
   float  d;
} RECORD;
typedef struct {
   int    a;
   float  d;
} OBJECT;
/*
 * Sum up attributes ``a'' and ``d'' of RECORD
 */
float foo(RECORD *A, int *offsets, int asize, int record_count,
          int *suma, float *sumd)
{
   vindirect2_args_t  vi2_args;
   OBJECT             *objects;
   RECORD             *record;
#ifdef IMPULSE
   vi2_args.newaddr           = (vaddr_t *) &objects;
   vi2_args.vaddr             = (vaddr_t) A;
   vi2_args.count             = record_count;
   vi2_args.size              = asize;
   vi2_args.attribs_num       = 2;
   vi2_args.attribs[0].offset = 0;
   vi2_args.attribs[0].size   = sizeof(int);
   vi2_args.attribs[1].offset = sizeof(int) * 2 + sizeof(float);
   vi2_args.attribs[1].size   = sizeof(float);
   vi2_args.offset_vaddr      = (vaddr_t) offsets;
   vi2_args.offset_objsize    = sizeof(int);
   vi2_args.prefcount         = 1;
   vi2_args.prefinfo          = 1;

   if (ams_mapshadow(AMS_TYPE_VINDIRECT2, &vi2_args) == -1) {
      perror("ams_mapshadow(AMS_TYPE_VINDIRECT2, ...) failed.");
      exit(1);
   }
   for (int i = 0; i < record_count; i++) {
       *suma += objects[i].a; *sumd += objects[i].d;
   }
#else
   for (int i = 0; i < record_count; i++) {
      record = (RECORD *) ((vaddr_t) A + (vaddr_t) offsets[i]);
      *suma += record->a; *sumd += record->d;
   }
#endif
}
```


Figure 8: A code fragment using `ams_mapshadow(AMS_TYPE_VINDIRECT2, ...)`.

2.1.7 ams_mapshadow(**AMS_TYPE_VINDIRECT3, vindirect3_args_t *datablock**)

`ams_mapshadow(AMS_TYPE_VINDIRECT3, datablock)` sets up **dynamic cacheline assembly**, a variation of **scatter/gather through an indirection vector** when the indirection vector is dynamically created by this system call. The argument `datablock` points to a `vindirect3_args_t` structure defined as the following:

```
typedef struct {
        vaddr_t  *newaddr;
        vaddr_t  *iv_newaddr;
        vaddr_t   vaddr;
        vaddr_t   size;
        int       objsize;
        int       objcount;
        int       iv_objsize;
        int       iv_subtype;
        int       prefcount;
        int       prefinfo;
        int       salign;
        int       valign;
} vindirect3_args_t;
```

This system call creates an indirection vector and an alias array. To access data using the alias array, the application program must first fill in the indirection vector, then flush it back to the memory.

`newaddr` and `iv_newaddr` are two pointers to locations in the application address space where the kernel can store the address of the alias array and indirection vector. `vaddr` points to the starting place of a virtual region inside which data will be gathered from. `size` is the number of bytes in this virtual region. `objsize` is the size of data item being gathered. `objcount` is the number of elements in the alias array or indirection vector. `iv_object` is the size of each element in the indirection vector. `iv_subtype` indicates the type of values stored in the indirection vector: 0 means array indices, 1 means byte offsets in virtual memory, 2 means virtual addresses, 3 means shadow addresses, and 4 means real physical addresses[1]. `prefcount` is the number of blocks to prefetch each time and `prefinfo` is the prefetch distance.

The return values of this function are the virtual addresses of a new alias array and a new indirection vector. `salign` specifies the expected alignment of the new alias array's shadow region; and `valign` specifies the expected alignment of the new alias array's virtual region. They determine where the alias array will be mapped to in the caches. Both of them must be page-aligned.

[1]Note that only subtype 0 is currently fully supported by the simulator. Other types will be supported if found necessary later.

`ams_mapshadow()` returns 0 on success, and -1 otherwise. If successful, the virtual address of the new alias array is placed into the memory location pointed to by `newaddr`; the virtual address of the new indirection vector is placed into the memory location pointed to by `iv_newaddr`;

ERRORS:

`EINVAL`	Either `vaddr` or `salign` or `valign` is not page-aligned.
`EFAULT`	`newaddr` or `iv_newaddr` specifies an invalid address.

Figure 9 shows a simple example of using `ams_mapshadow(AMS_TYPE_VINDIRECT3, ...)`. The example contains both the Impulse version and the non-Impulse version. The Impulse version has `IMPULSE` defined while the non-Impulse version has not.

2.2 ams_remapshadow()

```
int
ams_remapshadow(int       type,
                void     *datablock,
                int       flags);
```

`ams_remapshadow()` allows user applications to adjust the parameters of existing remappings. The argument `type` specifies remapping type. The argument `datablock` should point to a structure associated with `type`. The `newaddr` of `datablock` must be a virtual address returned by a previous call to `ams_mapshadow()`. It allows the kernel to find the previous setting of a specific remapping. The argument `flags` specifies what kinds of change to make.

Currently, there are only a very limited set of parameters allowed to be reset. More parameters would be allowed, should the needs arise. By now, `type` can be one of the followings: `AMS_TYPE_BASESTRIDE` or `AMS_TYPE_VINDIRECT3`. For `AMS_TYPE_BASESTRIDE`, `flags` can be one of the followings:

- `AMS_REMAP_PURGE` — purges the associated shadow region data out of CPU caches;
- `AMS_REMAP_FLUSH` — flushes the associated shadow region data back to main memory;
- `AMS_REMAP_STRIDE` — resets the associated remapping with new `stride` value in `datablock`;
- `AMS_REMAP_OFFSET` — resets the associated remapping with new `offset` value in `datablock`.

For `AMS_TYPE_VINDIRECT3`, `flags` can be one of the followings:

- `AMS_REMAP_VADDR` — resets the starting address of the original virtual region.

```
/*
 * Optimize a random access loop using AMS_TYPE_VINDIRECT3.
 * Basic idea: in each iteration, precompute 32 addresses then
 * access 32 data items.
 *
 */
#define OBJCOUNT        32

float foo(float *array, int size, int itcount)
{
   vindirect3_args_t  args;
   float *alias_array, sum;
   int   *idx_vector, i, j;

#ifdef IMPULSE
   args.newaddr    = (vaddr_t *) &alias_array;
   args.iv_newaddr = (vaddr_t *) &idx_vector;
   args.vaddr      = (vaddr_t) array;
   args.size       = sizeof(float) * size;
   args.objsize    = sizeof(float);
   args.objcount   = OBJCOUNT;
   args.iv_objsize = sizeof(int);
   args.prefcount  = 0; args.prefinfo = 0;
   args.salign     = 0; args.valign   = 0;

   if (ams_mapshadow(AMS_TYPE_VINDIRECT3, &args) == -1) {
      printf("ams_mapshadow(AMS_TYPE_VINDIRECT3, ...) failed.\n");
      exit(1);
   }

   for (sum = 0, i = 0; i < itcount / OBJCOUNT; i++) {
      /* Precompute addresses */
      for (j = 0; j < OBJCOUNT; j++)
         idx_vector[j] = random() % size;
      flush_cacheline(0, (vaddr_t) &(idx_vector[0]));

      /* Access data */
      for (j = 0; j < OBJCOUNT; j++)
         sum += alias_array[j];
      purge_cacheline(0, (vaddr_t) &(alias_array[0]));
   }
#else /* Non-Impulse version */
   for (sum = 0, i = 0; i < itcount; i++)
      sum += array[random() % size];
#endif
   return sum;
}
```

Figure 9: A code fragment using `ams_mapshadow(AMS_TYPE_VINDIRECT3, ...)`.

```
/*
 * Change the "offset" or "stride" of a strided remapping identified by
 * "vsaddr" which was returned by a previous call to ams_mapshadow().
 */
foo(double *vsaddr, int value, int flags)
{
   basestride_args_t bs_args;

   bs_args.newaddr = (unsigned *) &vsaddr;

   if (flags & AMS_REMAP_OFFSET)
      bs_args.offset = value;
   if (flags & AMS_REMAP_STRIDE)
      bs_args.stride = value;

   if (ams_remapshadow(AMS_TYPE_BASESTRIDE, &bs_args, flags) != 0) {
      perror("ams_remapshadow failed.");
      exit(1);
   }
}
```

Figure 10: Code fragment illustrating `ams_remapshadow()` usage.

`ams_remapshadow` returns 0 on success, and -1 otherwise.

ERRORS:

`EINVAL` `newaddr` does not specify a valid shadow range.

Figure 10 shows a simple code fragment using `ams_remapshadow()`.

2.3 ams_allocvirt()

```
unsigned long
ams_allocvirt(int size,
              int alignment);
```

User applications can use this system call and `ams_mapvtov()` (described in Section 2.4) to optimize their shadow data layout in virtually indexed caches. `ams_allocvirt()` allocates a new virtual region which will be used by `ams_mapvtov()`. The new virtual region is not mapped to any physical addresses, so it cannot be used until it has be mapped to a region of shadow addresses by `ams_mapvtov()`.

`ams_allocvirt()` returns the base of allocated virtual region on success, and -1 otherwise.

ERRORS:

`EINVAL`	Either `size` or `alignment` is not page-aligned.

2.4 ams_mapvtov()

```
int
ams_mapvtov(unsigned srcvaddr,
            unsigned dstvaddr
            int      size);
```

User applications uses `ams_allocvirt()` (described in Section 2.3) and this system call to optimize shadow data layout in virtually indexed caches (such as most L1 caches in modern microarchitectures). `ams_mapvtov()` remaps a region of virtual addresses starting at `srcvaddr` to a region of physical addresses originally mapped through the virtual region at `dstvaddr`. The size (in bytes) of remapped region is given by `size`. All of `srcvaddr`, `dstvaddr`, and `size` must be page-aligned values. The range of virtual address space specified by `dstvaddr` and `size` must map to a shadow address region previously allocated through a call to `ams_mapshadow()`.

It is worth noting that the application can really screw itself with this call, since the kernel will allow any region of the process' virtual address space to be remapped to any region of shadow address space previously allocated by the process.

`ams_mapvtov()` returns 0 on success, and -1 otherwise.

ERRORS:

`EINVAL`	Either `srcvaddr` or `dstvaddr` is not page-aligned.
`EINVAL`	`size` is not page-aligned.
`EINVAL`	`dstvaddr` and `size` do not specify a valid shadow range.

Figure 11 shows a simple example of using `ams_allocvirt()` and `ams_mapvtov()`.

```
/*
 * This function maps A, B, and C to the first, second, and third
 * quadrant of L1 cache respectively.
 * Assume A, B, and C have the same size as one-fourth of L1 cache,
 * and each of them maps to a shadow region.
 */
foo(void *A, void *B, void *C, int size)
{
   void *Av;

   if ((Av = ams_allocvirt(3 * size, L1_CACHE_SIZE) == -1) {
      perror("ams_allocvirt() failed.");
      exit(1);
   }

   if ((ams_mapvtov(Av,        A, size) == -1) ||
       (ams_mapvtov(Av+size,   B, size) == -1) ||
       (ams_mapvtov(Av+2*size, C, size) == -1)) {
      perror("ams_mapvtov() failed.");
      exit(1);
   }
}
```

Figure 11: Using `ams_allocvirt()` and `ams_mapvtov()` to optimize data layout in virtually indexed L1 cache.