```
**********************************************************
   18577 Thu Feb 12 20:33:37 2015
new/usr/src/uts/armv6/loader/fakeloader.c
loader: allow 1MB device maps
There's no reason we shouldn't allow 1MB PTEs for use on device memory.
**********************************************************
______unchanged_portion_omitted_

 152 static void
 153 fakeload_map_1mb(uintptr_t pa, uintptr_t va, int prot)
 154 {
 155         int entry;
 156         armpte_t *pte;
 157         arm_l1s_t *l1e;

 159         entry = ARMPT_VADDR_TO_L1E(va);
 160         pte = &pt_addr[entry];
 161         if (ARMPT_L1E_ISVALID(*pte))
 162                 fakeload_panic("armboot_mmu: asked to map a mapped region!\n");
 163         l1e = (arm_l1s_t *)pte;
 164         *pte = 0;
 165         l1e->al_type = ARMPT_L1_TYPE_SECT;

 167         if (prot & PF_DEVICE) {
 168                 l1e->al_bbit = 1;
 169                 l1e->al_cbit = 0;
 170                 l1e->al_tex = 0;
 171                 l1e->al_sbit = 1;
 172         } else {
 166         /* Assume it's not device memory */
 173                 l1e->al_bbit = 1;
 174                 l1e->al_cbit = 1;
 175                 l1e->al_tex = 1;
 176                 l1e->al_sbit = 1;
 177         }
 178 #endif /* ! codereview */

 180         if (!(prot & PF_X))
 181                 l1e->al_xn = 1;
 182         l1e->al_domain = 0;

 184         if (prot & PF_W) {
 185                 l1e->al_ap2 = 1;
 186                 l1e->al_ap = 1;
 187         } else {
 188                 l1e->al_ap2 = 0;
 189                 l1e->al_ap = 1;
 190         }
 191         l1e->al_ngbit = 0;
 192         l1e->al_issuper = 0;
 193         l1e->al_addr = ARMPT_PADDR_TO_L1SECT(pa);
 194 }

 196 /*
 197  * Set freemem to be 1 MB aligned at the end of boot archive. While the L1 Page
 198  * table only needs to be 16 KB aligned, we opt for 1 MB alignment so that way
 199  * we can map it and all the other L2 page tables we might need. If we don't do
 200  * this, it'll become problematic for unix to actually modify this.
 201  */
 202 static void
 203 fakeload_pt_arena_init(const atag_initrd_t *aii)
 204 {
 205         int entry, i;
 206         armpte_t *pte;
 207         arm_l1s_t *l1e;
```



```
 209         pt_arena = aii->ai_start + aii->ai_size;
 210         if (pt_arena & MMU_PAGEOFFSET1M) {
 211                 pt_arena &= MMU_PAGEMASK1M;
 212                 pt_arena += MMU_PAGESIZE1M;
 213         }
 214         pt_arena_max = pt_arena + 4 * MMU_PAGESIZE1M;
 215         freemem = pt_arena_max;

 217         /* Set up the l1 page table by first invalidating it */
 218         pt_addr = (armpte_t *)pt_arena;
 219         pt_arena += ARMPT_L1_SIZE;
 220         bzero(pt_addr, ARMPT_L1_SIZE);
 221         for (i = 0; i < 4; i++)
 222                 fakeload_map_1mb((uintptr_t)pt_addr + i * MMU_PAGESIZE1M,
 223                     (uintptr_t)pt_addr + i * MMU_PAGESIZE1M,
 224                     PF_R | PF_W);
 225 }

 227 /*
 228  * This is our generally entry point. We're passed in the entry point of the
 229  * header.
 230  */
 231 static uintptr_t
 232 fakeload_archive_mappings(atag_header_t *chain, const void *addr,
 233     atag_illumos_status_t *aisp)
 234 {
 235         atag_illumos_mapping_t aim;
 236         fakeloader_hdr_t *hdr;
 237         Elf32_Ehdr *ehdr;
 238         Elf32_Phdr *phdr;
 239         int nhdrs, i;
 240         uintptr_t ret;
 241         uintptr_t text = 0, data = 0;
 242         size_t textln = 0, dataln = 0;

 244         hdr = (fakeloader_hdr_t *)addr;

 246         if (hdr->fh_magic[0] != FH_MAGIC0)
 247                 fakeload_panic("fh_magic[0] is wrong!\n");
 248         if (hdr->fh_magic[1] != FH_MAGIC1)
 249                 fakeload_panic("fh_magic[1] is wrong!\n");
 250         if (hdr->fh_magic[2] != FH_MAGIC2)
 251                 fakeload_panic("fh_magic[2] is wrong!\n");
 252         if (hdr->fh_magic[3] != FH_MAGIC3)
 253                 fakeload_panic("fh_magic[3] is wrong!\n");

 255         if (hdr->fh_unix_size == 0)
 256                 fakeload_panic("hdr unix size is zero\n");
 257         if (hdr->fh_unix_offset == 0)
 258                 fakeload_panic("hdr unix offset is zero\n");
 259         if (hdr->fh_archive_size == 0)
 260                 fakeload_panic("hdr archive size is zero\n");
 261         if (hdr->fh_archive_offset == 0)
 262                 fakeload_panic("hdr archive_offset is zero\n");

 264         ehdr = (Elf32_Ehdr *)((uintptr_t)addr + hdr->fh_unix_offset);

 266         if (ehdr->e_ident[EI_MAG0] != ELFMAG0)
 267                 fakeload_panic("magic[0] wrong");
 268         if (ehdr->e_ident[EI_MAG1] != ELFMAG1)
 269                 fakeload_panic("magic[1] wrong");
 270         if (ehdr->e_ident[EI_MAG2] != ELFMAG2)
 271                 fakeload_panic("magic[2] wrong");
 272         if (ehdr->e_ident[EI_MAG3] != ELFMAG3)
 273                 fakeload_panic("magic[3] wrong");
 274         if (ehdr->e_ident[EI_CLASS] != ELFCLASS32)
```

```
275                 fakeload_panic("wrong elfclass");
276         if (ehdr->e_ident[EI_DATA] != ELFDATA2LSB)
277                 fakeload_panic("wrong encoding");
278         if (ehdr->e_ident[EI_OSABI] != ELFOSABI_SOLARIS)
279                 fakeload_panic("wrong os abi");
280         if (ehdr->e_ident[EI_ABIVERSION] != EAV_SUNW_CURRENT)
281                 fakeload_panic("wrong abi version");
282         if (ehdr->e_type != ET_EXEC)
283                 fakeload_panic("unix is not an executable");
284         if (ehdr->e_machine != EM_ARM)
285                 fakeload_panic("unix is not an ARM Executible");
286         if (ehdr->e_version != EV_CURRENT)
287                 fakeload_panic("wrong version");
288         if (ehdr->e_phnum == 0)
289                 fakeload_panic("no program headers");
290         ret = ehdr->e_entry;

292         FAKELOAD_DPRINTF("validated unix's headers\n");

294         nhdrs = ehdr->e_phnum;
295         phdr = (Elf32_Phdr *)((uintptr_t)addr + hdr->fh_unix_offset +
296             ehdr->e_phoff);
297         for (i = 0; i < nhdrs; i++, phdr++) {
298                 if (phdr->p_type != PT_LOAD) {
299                         fakeload_puts("skipping non-PT_LOAD header\n");
300                         continue;
301                 }

303                 if (phdr->p_filesz == 0 || phdr->p_memsz == 0) {
304                         fakeload_puts("skipping PT_LOAD with 0 file/mem\n");
305                         continue;
306                 }

308                 /*
309                  * Create a mapping record for this in the atags.
310                  */
311                 aim.aim_header.ah_size = ATAG_ILLUMOS_MAPPING_SIZE;
312                 aim.aim_header.ah_tag = ATAG_ILLUMOS_MAPPING;
313                 aim.aim_paddr = (uintptr_t)addr + hdr->fh_unix_offset +
314                     phdr->p_offset;
315                 aim.aim_plen = phdr->p_filesz;
316                 aim.aim_vaddr = phdr->p_vaddr;
317                 aim.aim_vlen = phdr->p_memsz;
318                 /* Round up vlen to be a multiple of 4k */
319                 if (aim.aim_vlen & 0xfff) {
320                         aim.aim_vlen &= ~0xfff;
321                         aim.aim_vlen += 0x1000;
322                 }
323                 aim.aim_mapflags = phdr->p_flags;
324                 atag_append(chain, &aim.aim_header);

326                 /*
327                  * When built with highvecs we need to account for the fact that
328                  * _edata, _etext and _end are built assuming that the highvecs
329                  * are normally part of our segments. ld is not doing anything
330                  * wrong, but this breaks the assumptions that krtld currently
331                  * has. As such, unix will use this information to overwrite the
332                  * normal entry points that krtld uses in a similar style to
333                  * SPARC.
334                  */
335                 if (aim.aim_vaddr != 0xffff0000) {
336                         if ((phdr->p_flags & PF_W) != 0) {
337                                 data = aim.aim_vaddr;
338                                 dataln = aim.aim_vlen;
339                         } else {
340                                 text = aim.aim_vaddr;
```



```
341                                 textln = aim.aim_vlen;
342                         }
343                 }
344         }

346         aisp->ais_stext = text;
347         aisp->ais_etext = text + textln;
348         aisp->ais_sdata = data;
349         aisp->ais_edata = data + dataln;

351         /* 1:1 map the boot archive */
352         aim.aim_header.ah_size = ATAG_ILLUMOS_MAPPING_SIZE;
353         aim.aim_header.ah_tag = ATAG_ILLUMOS_MAPPING;
354         aim.aim_paddr = (uintptr_t)addr + hdr->fh_archive_offset;
355         aim.aim_plen = hdr->fh_archive_size;
356         aim.aim_vaddr = aim.aim_paddr;
357         aim.aim_vlen = aim.aim_plen;
358         aim.aim_mapflags = PF_R | PF_W | PF_X;
359         atag_append(chain, &aim.aim_header);
360         aisp->ais_archive = aim.aim_paddr;
361         aisp->ais_archivelen = aim.aim_plen;

363         return (ret);
364 }

366 static void
367 fakeload_mkatags(atag_header_t *chain)
368 {
369         atag_illumos_status_t ais;
370         atag_illumos_mapping_t aim;

372         bzero(&ais, sizeof (ais));
373         bzero(&aim, sizeof (aim));

375         ais.ais_header.ah_size = ATAG_ILLUMOS_STATUS_SIZE;
376         ais.ais_header.ah_tag = ATAG_ILLUMOS_STATUS;
377         atag_append(chain, &ais.ais_header);
378         aim.aim_header.ah_size = ATAG_ILLUMOS_MAPPING_SIZE;
379         aim.aim_header.ah_tag = ATAG_ILLUMOS_MAPPING;
380         atag_append(chain, &aim.aim_header);
381 }

383 static uintptr_t
384 fakeload_alloc_l2pt(void)
385 {
386         uintptr_t ret;

388         if (pt_arena & ARMPT_L2_MASK) {
389                 ret = pt_arena;
390                 ret &= ~ARMPT_L2_MASK;
391                 ret += ARMPT_L2_SIZE;
392                 pt_arena = ret + ARMPT_L2_SIZE;
393         } else {
394                 ret = pt_arena;
395                 pt_arena = ret + ARMPT_L2_SIZE;
396         }
397         if (pt_arena >= pt_arena_max) {
398                 fakeload_puts("pt_arena, max\n");
399                 fakeload_ultostr(pt_arena);
400                 fakeload_puts("\n");
401                 fakeload_ultostr(pt_arena_max);
402                 fakeload_puts("\n");
403                 fakeload_puts("l2pts alloced\n");
404                 fakeload_ultostr(nl2pages);
405                 fakeload_puts("\n");
406                 fakeload_panic("ran out of page tables!");
```

```
407         }
409         bzero((void *)ret, ARMPT_L2_SIZE);
410         nl2pages++;
411         return (ret);
412 }
414 /*
415  * Finally, do all the dirty work. Let's create some page tables. The L1 page
416  * table is full of 1 MB mappings by default. The L2 Page table is 1k in size
417  * and covers that 1 MB. We're going to always create L2 page tables for now
418  * which will use 4k and 64k pages.
419  */
420 static void
421 fakeload_map(armpte_t *pt, uintptr_t pstart, uintptr_t vstart, size_t len,
422     uint32_t prot)
423 {
424         int entry, chunksize;
425         armpte_t *pte, *l2pt;
426         arm_l1pt_t *l1pt;
428         /*
429          * Make sure both pstart + vstart are 4k aligned, along with len.
430          */
431         if (pstart & MMU_PAGEOFFSET)
432                 fakeload_panic("pstart is not 4k aligned");
433         if (vstart & MMU_PAGEOFFSET)
434                 fakeload_panic("vstart is not 4k aligned");
435         if (len & MMU_PAGEOFFSET)
436                 fakeload_panic("len is not 4k aligned");
438         /*
439          * We're going to logically deal with each 1 MB chunk at a time.
440          */
441         while (len > 0) {
442                 if (vstart & MMU_PAGEOFFSET1M) {
443                         chunksize = MIN(len, MMU_PAGESIZE1M -
444                             (vstart & MMU_PAGEOFFSET1M));
445                 } else {
446                         chunksize = MIN(len, MMU_PAGESIZE1M);
447                 }
449                 entry = ARMPT_VADDR_TO_L1E(vstart);
450                 pte = &pt[entry];
452                 if (!ARMPT_L1E_ISVALID(*pte)) {
453                         uintptr_t l2table;
455                         if (!(vstart & MMU_PAGEOFFSET1M) &&
456                             !(pstart & MMU_PAGEOFFSET1M) &&
457                             len >= MMU_PAGESIZE1M) {
458                                 fakeload_map_1mb(pstart, vstart, prot);
459                                 vstart += MMU_PAGESIZE1M;
460                                 pstart += MMU_PAGESIZE1M;
461                                 len -= MMU_PAGESIZE1M;
462                                 continue;
463                         }
465                         l2table = fakeload_alloc_l2pt();
466                         *pte = 0;
467                         l1pt = (arm_l1pt_t *)pte;
468                         l1pt->al_type = ARMPT_L1_TYPE_L2PT;
469                         l1pt->al_ptaddr = ARMPT_ADDR_TO_L1PTADDR(l2table);
470                 } else if ((*pte & ARMPT_L1_TYPE_MASK) != ARMPT_L1_TYPE_L2PT) {
471                         fakeload_panic("encountered l1 entry that's not a "
472                             "pointer to a level 2 table\n");
```



```
473                 } else {
474                         l1pt = (arm_l1pt_t *)pte;
475                 }
477                 /* Now that we have the l1pt fill in l2 entries */
478                 l2pt = (void *)(l1pt->al_ptaddr << ARMPT_L1PT_TO_L2_SHIFT);
479                 len -= chunksize;
480                 while (chunksize > 0) {
481                         arm_l2e_t *l2pte;
483                         entry = ARMPT_VADDR_TO_L2E(vstart);
484                         pte = &l2pt[entry];
486 #ifdef  MAP_DEBUG
487                         fakeload_puts("4k page pa->va, l2root, entry\n");
488                         fakeload_ultostr(pstart);
489                         fakeload_puts("->");
490                         fakeload_ultostr(vstart);
491                         fakeload_puts(", ");
492                         fakeload_ultostr((uintptr_t)l2pt);
493                         fakeload_puts(", ");
494                         fakeload_ultostr(entry);
495                         fakeload_puts("\n");
496 #endif
498                         if ((*pte & ARMPT_L2_TYPE_MASK) !=
499                             ARMPT_L2_TYPE_INVALID)
500                                 fakeload_panic("found existing l2 page table, "
501                                     "overlap in requested mappings detected!");
502                         /* Map vaddr to our paddr! */
503                         l2pte = ((arm_l2e_t *)pte);
504                         *pte = 0;
505                         if (!(prot & PF_X))
506                                 l2pte->ale_xn = 1;
507                         l2pte->ale_ident = 1;
508                         if (prot & PF_DEVICE) {
509                                 l2pte->ale_bbit = 1;
510                                 l2pte->ale_cbit = 0;
511                                 l2pte->ale_tex = 0;
512                                 l2pte->ale_sbit = 1;
513                         } else {
514                                 l2pte->ale_bbit = 1;
515                                 l2pte->ale_cbit = 1;
516                                 l2pte->ale_tex = 1;
517                                 l2pte->ale_sbit = 1;
518                         }
519                         if (prot & PF_W) {
520                                 l2pte->ale_ap2 = 1;
521                                 l2pte->ale_ap = 1;
522                         } else {
523                                 l2pte->ale_ap2 = 0;
524                                 l2pte->ale_ap = 1;
525                         }
526                         l2pte->ale_ngbit = 0;
527                         l2pte->ale_addr = ARMPT_PADDR_TO_L2ADDR(pstart);
529                         chunksize -= MMU_PAGESIZE;
530                         vstart += MMU_PAGESIZE;
531                         pstart += MMU_PAGESIZE;
532                 }
533         }
534 }
536 static void
537 fakeload_create_map(armpte_t *pt, atag_illumos_mapping_t *aimp)
538 {
```

```
#ifdef MAP_DEBUG
        fakeload_puts("paddr->vaddr\n");
        fakeload_ultostr(aimp->aim_paddr);
        fakeload_puts("->");
        fakeload_ultostr(aimp->aim_vaddr);
        fakeload_puts("\n");
        fakeload_puts("plen-vlen\n");
        fakeload_ultostr(aimp->aim_plen);
        fakeload_puts("-");
        fakeload_ultostr(aimp->aim_vlen);
        fakeload_puts("\n");
#endif /* MAP_DEBUG */

        /*
         * Can we map this in place or do we need to basically allocate a new
         * region and bcopy everything into place for proper alignment?
         *
         * Criteria for this: we have a vlen > plen. plen is not page aligned.
         */
        if (aimp->aim_vlen > aimp->aim_plen ||
            (aimp->aim_paddr & MMU_PAGEOFFSET) != 0) {
                uintptr_t start;

                if (aimp->aim_mapflags & PF_NORELOC)
                        fakeload_panic("tried to reloc unrelocatable mapping");
#ifdef  MAP_DEBUG
                FAKELOAD_DPRINTF("reloacting paddr\n");
#endif
                start = freemem;
                if (start & MMU_PAGEOFFSET) {
                        start &= MMU_PAGEMASK;
                        start += MMU_PAGESIZE;
                }
                bcopy((void *)aimp->aim_paddr, (void *)start,
                    aimp->aim_plen);
                if (aimp->aim_vlen > aimp->aim_plen) {
                        bzero((void *)(start + aimp->aim_plen),
                            aimp->aim_vlen - aimp->aim_plen);
                }
                aimp->aim_paddr = start;
                freemem = start + aimp->aim_vlen;
#ifdef MAP_DEBUG
                fakeload_puts("new paddr: ");
                fakeload_ultostr(start);
                fakeload_puts("\n");
#endif /* MAP_DEBUG */
        }

        /*
         * Now that everything has been set up, go ahead and map the new region.
         */
        fakeload_map(pt, aimp->aim_paddr, aimp->aim_vaddr, aimp->aim_vlen,
            aimp->aim_mapflags);
#ifdef MAP_DEBUG
        FAKELOAD_DPRINTF("\n");
#endif /* MAP_DEBUG */
}

void
fakeload_init(void *ident, void *ident2, void *atag)
{
        atag_header_t *hdr;
        atag_header_t *chain = (atag_header_t *)atag;
        const atag_initrd_t *initrd;
        atag_illumos_status_t *aisp;
        atag_illumos_mapping_t *aimp;
```



```
        uintptr_t unix_start;

        fakeload_backend_init();
        fakeload_puts("Hello from the loader\n");
        initrd = (atag_initrd_t *)atag_find(chain, ATAG_INITRD2);
        if (initrd == NULL)
                fakeload_panic("missing the initial ramdisk\n");

        /*
         * Create the status atag header and the initial mapping record for the
         * atags. We'll hold onto both of these.
         */
        fakeload_mkatags(chain);
        aisp = (atag_illumos_status_t *)atag_find(chain, ATAG_ILLUMOS_STATUS);
        if (aisp == NULL)
                fakeload_panic("can't find ATAG_ILLUMOS_STATUS");
        aimp = (atag_illumos_mapping_t *)atag_find(chain, ATAG_ILLUMOS_MAPPING);
        if (aimp == NULL)
                fakeload_panic("can't find ATAG_ILLUMOS_MAPPING");
        FAKELOAD_DPRINTF("created proto atags\n");

        fakeload_pt_arena_init(initrd);

        fakeload_selfmap(chain);

        /*
         * Map the boot archive and all of unix
         */
        unix_start = fakeload_archive_mappings(chain,
            (const void *)(uintptr_t)initrd->ai_start, aisp);
        FAKELOAD_DPRINTF("filled out unix and the archive's mappings\n");

        /*
         * Fill in the atag mapping header for the atags themselves. 1:1 map it.
         */
        aimp->aim_paddr = (uintptr_t)chain & ~0xfff;
        aimp->aim_plen = atag_length(chain) & ~0xfff;
        aimp->aim_plen += 0x1000;
        aimp->aim_vaddr = aimp->aim_paddr;
        aimp->aim_vlen = aimp->aim_plen;
        aimp->aim_mapflags = PF_R | PF_W | PF_NORELOC;

        /*
         * Let the backend add mappings
         */
        fakeload_backend_addmaps(chain);

        /*
         * Turn on unaligned access
         */
        FAKELOAD_DPRINTF("turning on unaligned access\n");
        fakeload_unaligned_enable();
        FAKELOAD_DPRINTF("successfully enabled unaligned access\n");

        /*
         * To turn on the MMU we need to do the following:
         *  o Program all relevant CP15 registers
         *  o Program 1st and 2nd level page tables
         *  o Invalidate and Disable the I/D-cache
         *  o Fill in the last bits of the ATAG_ILLUMOS_STATUS atag
         *  o Turn on the MMU in SCTLR
         *  o Jump to unix
         */

        /* Last bits of the atag */
        aisp->ais_freemem = freemem;
```

```
671         aisp->ais_version = 1;
672         aisp->ais_ptbase = (uintptr_t)pt_addr;

674         /*
675          * Our initial page table is a series of 1 MB sections. While we really
676          * should map 4k pages, for the moment we're just going to map 1 MB
677          * regions, yay team!
678          */
679         hdr = chain;
680         FAKELOAD_DPRINTF("creating mappings\n");
681         while (hdr != NULL) {
682                 if (hdr->ah_tag == ATAG_ILLUMOS_MAPPING)
683                         fakeload_create_map(pt_addr,
684                             (atag_illumos_mapping_t *)hdr);
685                 hdr = atag_next(hdr);
686         }

688         /*
689          * Now that we've mapped everything, update the status atag.
690          */
691         aisp->ais_freeused = freemem - aisp->ais_freemem;
692         aisp->ais_pt_arena = pt_arena;
693         aisp->ais_pt_arena_max = pt_arena_max;

695         /* Cache disable */
696         FAKELOAD_DPRINTF("Flushing and disabling caches\n");
697         armv6_dcache_flush();
698         armv6_dcache_disable();
699         armv6_dcache_inval();
700         armv6_icache_disable();
701         armv6_icache_inval();

703         /* Program the page tables */
704         FAKELOAD_DPRINTF("programming cp15 regs\n");
705         fakeload_pt_setup((uintptr_t)pt_addr);


708         /* MMU Enable */
709         FAKELOAD_DPRINTF("see you on the other side\n");
710         fakeload_mmu_enable();

712         FAKELOAD_DPRINTF("why helo thar\n");

714         /* we should never come back */
715         fakeload_exec(ident, ident2, chain, unix_start);
716         fakeload_panic("hit the end of the world\n");
717 }
```