```
**********************************************************
  142100 Thu Sep  7 15:20:59 2017
new/usr/src/uts/i86pc/io/rootnex.c
8624 xhci and nvme can't bind DMA memory with IOMMU enabled
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
**********************************************************
______unchanged_portion_omitted_

1963 /*ARGSUSED*/
1964 static int
1965 rootnex_coredma_bindhdl(dev_info_t *dip, dev_info_t *rdip,
1966     ddi_dma_handle_t handle, struct ddi_dma_req *dmareq,
1967     ddi_dma_cookie_t *cookiep, uint_t *ccountp)
1968 {
1969         rootnex_sglinfo_t *sinfo;
1970         ddi_dma_obj_t *dmao;
1971 #if defined(__amd64) && !defined(__xpv)
1972         struct dvmaseg *dvs;
1973         ddi_dma_cookie_t *cookie;
1974 #endif
1975         ddi_dma_attr_t *attr;
1976         ddi_dma_impl_t *hp;
1977         rootnex_dma_t *dma;
1978         int kmflag;
1979         int e;
1980         uint_t ncookies;

1982         hp = (ddi_dma_impl_t *)handle;
1983         dma = (rootnex_dma_t *)hp->dmai_private;
1984         dmao = &dma->dp_dma;
1985         sinfo = &dma->dp_sglinfo;
1986         attr = &hp->dmai_attr;

1988         /* convert the sleep flags */
1989         if (dmareq->dmar_fp == DDI_DMA_SLEEP) {
1990                 dma->dp_sleep_flags = kmflag = KM_SLEEP;
1991         } else {
1992                 dma->dp_sleep_flags = kmflag = KM_NOSLEEP;
1993         }

1995         hp->dmai_rflags = dmareq->dmar_flags & DMP_DDIFLAGS;

1997         /*
1998          * This is useful for debugging a driver. Not as useful in a production
1999          * system. The only time this will fail is if you have a driver bug.
2000          */
2001         if (rootnex_bind_check_inuse) {
2002                 /*
2003                  * No one else should ever have this lock unless someone else
2004                  * is trying to use this handle. So contention on the lock
2005                  * is the same as inuse being set.
2006                  */
2007                 e = mutex_tryenter(&dma->dp_mutex);
2008                 if (e == 0) {
2009                         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_BIND_FAIL]);
2010                         return (DDI_DMA_INUSE);
2011                 }
2012                 if (dma->dp_inuse) {
2013                         mutex_exit(&dma->dp_mutex);
2014                         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_BIND_FAIL]);
2015                         return (DDI_DMA_INUSE);
2016                 }
2017                 dma->dp_inuse = B_TRUE;
2018                 mutex_exit(&dma->dp_mutex);
2019         }
```



```
2021         /* check the ddi_dma_attr arg to make sure it makes a little sense */
2022         if (rootnex_bind_check_parms) {
2023                 e = rootnex_valid_bind_parms(dmareq, attr);
2024                 if (e != DDI_SUCCESS) {
2025                         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_BIND_FAIL]);
2026                         rootnex_clean_dmahdl(hp);
2027                         return (e);
2028                 }
2029         }

2031         /* save away the original bind info */
2032         dma->dp_dma = dmareq->dmar_object;

2034 #if defined(__amd64) && !defined(__xpv)
2035         if (IOMMU_USED(rdip)) {
2036                 dmao = &dma->dp_dvma;
2037                 e = iommulib_nexdma_mapobject(dip, rdip, handle, dmareq, dmao);
2038                 switch (e) {
2039                 case DDI_SUCCESS:
2040                         if (sinfo->si_cancross ||
2041                             dmao->dmao_obj.dvma_obj.dv_nseg != 1 ||
2042                             dmao->dmao_size > sinfo->si_max_cookie_size) {
2043                                 dma->dp_dvma_used = B_TRUE;
2044                                 break;
2045                         }
2046                         sinfo->si_sgl_size = 1;
2047                         hp->dmai_rflags |= DMP_NOSYNC;

2049                         dma->dp_dvma_used = B_TRUE;
2050                         dma->dp_need_to_free_cookie = B_FALSE;

2052                         dvs = &dmao->dmao_obj.dvma_obj.dv_seg[0];
2053                         cookie = hp->dmai_cookie = dma->dp_cookies =
2054                             (ddi_dma_cookie_t *)dma->dp_prealloc_buffer;
2055                         cookie->dmac_laddress = dvs->dvs_start +
2056                             dmao->dmao_obj.dvma_obj.dv_off;
2057                         cookie->dmac_size = dvs->dvs_len;
2058                         cookie->dmac_type = 0;

2060                         ROOTNEX_DPROBE1(rootnex__bind__dvmafast, dev_info_t *,
2061                             rdip);
2062                         goto fast;
2063                 case DDI_ENOTSUP:
2064                         break;
2065                 default:
2066                         rootnex_clean_dmahdl(hp);
2067                         return (e);
2068                 }
2069         }
2070 #endif

2072         /*
2073          * Figure out a rough estimate of what maximum number of pages
2074          * this buffer could use (a high estimate of course).
2075          */
2076         sinfo->si_max_pages = mmu_btopr(dma->dp_dma.dmao_size) + 1;

2078         if (dma->dp_dvma_used) {
2079                 /*
2080                  * The number of physical pages is the worst case.
2081                  *
2082                  * For DVMA, the worst case is the length divided
2083                  * by the maximum cookie length, plus 1. Add to that
2084                  * the number of segment boundaries potentially crossed, and
2085                  * the additional number of DVMA segments that was returned.
```

```
2086                  *
2087                  * In the normal case, for modern devices, si_cancross will
2088                  * be false, and dv_nseg will be 1, and the fast path will
2089                  * have been taken above.
2090                  */
2091                 ncookies = (dma->dp_dma.dmao_size / sinfo->si_max_cookie_size)
2092                     + 1;
2093                 if (sinfo->si_cancross)
2094                         ncookies +=
2095                             (dma->dp_dma.dmao_size / attr->dma_attr_seg) + 1;
2096                 ncookies += (dmao->dmao_obj.dvma_obj.dv_nseg - 1);

2098                 sinfo->si_max_pages = MIN(sinfo->si_max_pages, ncookies);
2099         }

2101         /*
2102          * We'll use the pre-allocated cookies for any bind that will *always*
2103          * fit (more important to be consistent, we don't want to create
2104          * additional degenerate cases).
2105          */
2106         if (sinfo->si_max_pages <= rootnex_state->r_prealloc_cookies) {
2107                 dma->dp_cookies = (ddi_dma_cookie_t *)dma->dp_prealloc_buffer;
2108                 dma->dp_need_to_free_cookie = B_FALSE;
2109                 ROOTNEX_DPROBE2(rootnex__bind__prealloc, dev_info_t *, rdip,
2110                     uint_t, sinfo->si_max_pages);

2112         /*
2113          * For anything larger than that, we'll go ahead and allocate the
2114          * maximum number of pages we expect to see. Hopefuly, we won't be
2115          * seeing this path in the fast path for high performance devices very
2116          * frequently.
2117          *
2118          * a ddi bind interface that allowed the driver to provide storage to
2119          * the bind interface would speed this case up.
2120          */
2121         } else {
2122                 /*
2123                  * Save away how much memory we allocated. If we're doing a
2124                  * nosleep, the alloc could fail...
2125                  */
2126                 dma->dp_cookie_size = sinfo->si_max_pages *
2127                     sizeof (ddi_dma_cookie_t);
2128                 dma->dp_cookies = kmem_alloc(dma->dp_cookie_size, kmflag);
2129                 if (dma->dp_cookies == NULL) {
2130                         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_BIND_FAIL]);
2131                         rootnex_clean_dmahdl(hp);
2132                         return (DDI_DMA_NORESOURCES);
2133                 }
2134                 dma->dp_need_to_free_cookie = B_TRUE;
2135                 ROOTNEX_DPROBE2(rootnex__bind__alloc, dev_info_t *, rdip,
2136                     uint_t, sinfo->si_max_pages);
2137         }
2138         hp->dmai_cookie = dma->dp_cookies;

2140         /*
2141          * Get the real sgl. rootnex_get_sgl will fill in cookie array while
2142          * looking at the constraints in the dma structure. It will then put
2143          * some additional state about the sgl in the dma struct (i.e. is
2144          * the sgl clean, or do we need to do some munging; how many pages
2145          * need to be copied, etc.)
2146          */
2147         if (dma->dp_dvma_used)
2148                 rootnex_dvma_get_sgl(dmao, dma->dp_cookies, &dma->dp_sglinfo);
2149         else
2150                 rootnex_get_sgl(dmao, dma->dp_cookies, &dma->dp_sglinfo);
```



```
2152 out:
2153         ASSERT(sinfo->si_sgl_size <= sinfo->si_max_pages);
2154         /* if we don't need a copy buffer, we don't need to sync */
2155         if (sinfo->si_copybuf_req == 0) {
2156                 hp->dmai_rflags |= DMP_NOSYNC;
2157         }

2159         /*
2160          * if we don't need the copybuf and we don't need to do a partial,  we
2161          * hit the fast path. All the high performance devices should be trying
2162          * to hit this path. To hit this path, a device should be able to reach
2163          * all of memory, shouldn't try to bind more than it can transfer, and
2164          * the buffer shouldn't require more cookies than the driver/device can
2165          * handle [sgllen]).
2166          *
2167          * Note that negative values of dma_attr_sgllen are supposed
2168          * to mean unlimited, but we just cast them to mean a
2169          * "ridiculous large limit".  This saves some extra checks on
2170          * hot paths.
2171          */
2172         if ((sinfo->si_copybuf_req == 0) &&
2173             (sinfo->si_sgl_size <= (unsigned)attr->dma_attr_sgllen) &&
2174             (dmao->dmao_size <= dma->dp_maxxfer)) {
2174             (dmao->dmao_size < dma->dp_maxxfer)) {
2175 fast:
2176                 /*
2177                  * If the driver supports FMA, insert the handle in the FMA DMA
2178                  * handle cache.
2179                  */
2180                 if (attr->dma_attr_flags & DDI_DMA_FLAGERR)
2181                         hp->dmai_error.err_cf = rootnex_dma_check;

2183                 /*
2184                  * copy out the first cookie and ccountp, set the cookie
2185                  * pointer to the second cookie. The first cookie is passed
2186                  * back on the stack. Additional cookies are accessed via
2187                  * ddi_dma_nextcookie()
2188                  */
2189                 *cookiep = dma->dp_cookies[0];
2190                 *ccountp = sinfo->si_sgl_size;
2191                 hp->dmai_cookie++;
2192                 hp->dmai_rflags &= ~DDI_DMA_PARTIAL;
2193                 ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_ACTIVE_BINDS]);
2194                 ROOTNEX_DPROBE4(rootnex__bind__fast, dev_info_t *, rdip,
2195                     uint64_t, rootnex_cnt[ROOTNEX_CNT_ACTIVE_BINDS],
2196                     uint_t, dmao->dmao_size, uint_t, *ccountp);


2199                 return (DDI_DMA_MAPPED);
2200         }

2202         /*
2203          * go to the slow path, we may need to alloc more memory, create
2204          * multiple windows, and munge up a sgl to make the device happy.
2205          */

2207         /*
2208          * With the IOMMU mapobject method used, we should never hit
2209          * the slow path. If we do, something is seriously wrong.
2210          * Clean up and return an error.
2211          */

2213 #if defined(__amd64) && !defined(__xpv)

2215         if (dma->dp_dvma_used) {
2216                 (void) iommulib_nexdma_unmapobject(dip, rdip, handle,
```

```
2217                     &dma->dp_dvma);
2218                 e = DDI_DMA_NOMAPPING;
2219         } else {
2220 #endif
2221                 e = rootnex_bind_slowpath(hp, dmareq, dma, attr, &dma->dp_dma,
2222                     kmflag);
2223 #if defined(__amd64) && !defined(__xpv)
2224         }
2225 #endif
2226         if ((e != DDI_DMA_MAPPED) && (e != DDI_DMA_PARTIAL_MAP)) {
2227                 if (dma->dp_need_to_free_cookie) {
2228                         kmem_free(dma->dp_cookies, dma->dp_cookie_size);
2229                 }
2230                 ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_BIND_FAIL]);
2231                 rootnex_clean_dmahdl(hp); /* must be after free cookie */
2232                 return (e);
2233         }

2235         /*
2236          * If the driver supports FMA, insert the handle in the FMA DMA handle
2237          * cache.
2238          */
2239         if (attr->dma_attr_flags & DDI_DMA_FLAGERR)
2240                 hp->dmai_error.err_cf = rootnex_dma_check;

2242         /* if the first window uses the copy buffer, sync it for the device */
2243         if ((dma->dp_window[dma->dp_current_win].wd_dosync) &&
2244             (hp->dmai_rflags & DDI_DMA_WRITE)) {
2245                 (void) rootnex_coredma_sync(dip, rdip, handle, 0, 0,
2246                     DDI_DMA_SYNC_FORDEV);
2247         }

2249         /*
2250          * copy out the first cookie and ccountp, set the cookie pointer to the
2251          * second cookie. Make sure the partial flag is set/cleared correctly.
2252          * If we have a partial map (i.e. multiple windows), the number of
2253          * cookies we return is the number of cookies in the first window.
2254          */
2255         if (e == DDI_DMA_MAPPED) {
2256                 hp->dmai_rflags &= ~DDI_DMA_PARTIAL;
2257                 *ccountp = sinfo->si_sgl_size;
2258                 hp->dmai_nwin = 1;
2259         } else {
2260                 hp->dmai_rflags |= DDI_DMA_PARTIAL;
2261                 *ccountp = dma->dp_window[dma->dp_current_win].wd_cookie_cnt;
2262                 ASSERT(hp->dmai_nwin <= dma->dp_max_win);
2263         }
2264         *cookiep = dma->dp_cookies[0];
2265         hp->dmai_cookie++;

2267         ROOTNEX_DPROF_INC(&rootnex_cnt[ROOTNEX_CNT_ACTIVE_BINDS]);
2268         ROOTNEX_DPROBE4(rootnex__bind__slow, dev_info_t *, rdip, uint64_t,
2269             rootnex_cnt[ROOTNEX_CNT_ACTIVE_BINDS], uint_t,
2270             dmao->dmao_size, uint_t, *ccountp);
2271         return (e);
2272 }
```

_____***unchanged_portion_omitted_***