384 lines (333 sloc) | 10.8 KB

```
1   /* SPDX-License-Identifier: GPL-2.0 */
2   #ifndef _ASM_X86_MSR_H
3   #define _ASM_X86_MSR_H
4
5   #include "msr-index.h"
6
7   #ifndef __ASSEMBLY__
8
9   #include <asm/asm.h>
10  #include <asm/errno.h>
11  #include <asm/cpumask.h>
12  #include <uapi/asm/msr.h>
13  #include <asm/shared/msr.h>
14
15  struct msr_info {
16          u32 msr_no;
17          struct msr reg;
18          struct msr *msrs;
19          int err;
20  };
21
22  struct msr_regs_info {
23          u32 *regs;
24          int err;
25  };
26
27  struct saved_msr {
28          bool valid;
29          struct msr_info info;
```

```
};

struct saved_msrs {
        unsigned int num;
        struct saved_msr *array;
};

/*
 * both i386 and x86_64 returns 64-bit value in edx:eax, but gcc's "A"
 * constraint has different meanings. For i386, "A" means exactly
 * edx:eax, while for x86_64 it doesn't mean rdx:rax or edx:eax. Instead,
 * it means rax *or* rdx.
 */
#ifdef CONFIG_X86_64
/* Using 64-bit values saves one instruction clearing the high half of low */
#define DECLARE_ARGS(val, low, high)    unsigned long low, high
#define EAX_EDX_VAL(val, low, high)     ((low) | (high) << 32)
#define EAX_EDX_RET(val, low, high)     "=a" (low), "=d" (high)
#else
#define DECLARE_ARGS(val, low, high)    unsigned long long val
#define EAX_EDX_VAL(val, low, high)     (val)
#define EAX_EDX_RET(val, low, high)     "=A" (val)
#endif

/*
 * Be very careful with includes. This header is prone to include loops.
 */
#include <asm/atomic.h>
#include <linux/tracepoint-defs.h>

#ifdef CONFIG_TRACEPOINTS
DECLARE_TRACEPOINT(read_msr);
DECLARE_TRACEPOINT(write_msr);
DECLARE_TRACEPOINT(rdpmc);
extern void do_trace_write_msr(unsigned int msr, u64 val, int failed);
extern void do_trace_read_msr(unsigned int msr, u64 val, int failed);
extern void do_trace_rdpmc(unsigned int msr, u64 val, int failed);
#else
static inline void do_trace_write_msr(unsigned int msr, u64 val, int failed) {}
static inline void do_trace_read_msr(unsigned int msr, u64 val, int failed) {}
static inline void do_trace_rdpmc(unsigned int msr, u64 val, int failed) {}
#endif

/*
 * __rdmsr() and __wrmsr() are the two primitives which are the bare minimum MSR
 * accessors and should not have any tracing or other functionality piggybacking
 * on them - those are *purely* for accessing MSRs and nothing more. So don't even
 * think of extending them - you will be slapped with a stinking trout or a frozen
 * shark will reach you, wherever you are! You've been warned.
```

```
 */
static __always_inline unsigned long long __rdmsr(unsigned int msr)
{
        DECLARE_ARGS(val, low, high);

        asm volatile("1: rdmsr\n"
                     "2:\n"
                     _ASM_EXTABLE_TYPE(1b, 2b, EX_TYPE_RDMSR)
                     : EAX_EDX_RET(val, low, high) : "c" (msr));

        return EAX_EDX_VAL(val, low, high);
}

static __always_inline void __wrmsr(unsigned int msr, u32 low, u32 high)
{
        asm volatile("1: wrmsr\n"
                     "2:\n"
                     _ASM_EXTABLE_TYPE(1b, 2b, EX_TYPE_WRMSR)
                     : : "c" (msr), "a"(low), "d" (high) : "memory");
}

#define native_rdmsr(msr, val1, val2)                   \
do {                                                    \
        u64 __val = __rdmsr((msr));                     \
        (void)((val1) = (u32)__val);                    \
        (void)((val2) = (u32)(__val >> 32));            \
} while (0)

#define native_wrmsr(msr, low, high)                    \
        __wrmsr(msr, low, high)

#define native_wrmsrl(msr, val)                         \
        __wrmsr((msr), (u32)((u64)(val)),               \
                       (u32)((u64)(val) >> 32))

static inline unsigned long long native_read_msr(unsigned int msr)
{
        unsigned long long val;

        val = __rdmsr(msr);

        if (tracepoint_enabled(read_msr))
                do_trace_read_msr(msr, val, 0);

        return val;
}

static inline unsigned long long native_read_msr_safe(unsigned int msr,
                                                       int *err)
```

```
128 {
129         DECLARE_ARGS(val, low, high);
130
131         asm volatile("1: rdmsr ; xor %[err],%[err]\n"
132                      "2:\n\t"
133                      _ASM_EXTABLE_TYPE_REG(1b, 2b, EX_TYPE_RDMSR_SAFE, %[err])
134                      : [err] "=r" (*err), EAX_EDX_RET(val, low, high)
135                      : "c" (msr));
136         if (tracepoint_enabled(read_msr))
137                 do_trace_read_msr(msr, EAX_EDX_VAL(val, low, high), *err);
138         return EAX_EDX_VAL(val, low, high);
139 }
140
141 /* Can be uninlined because referenced by paravirt */
142 static inline void notrace
143 native_write_msr(unsigned int msr, u32 low, u32 high)
144 {
145         __wrmsr(msr, low, high);
146
147         if (tracepoint_enabled(write_msr))
148                 do_trace_write_msr(msr, ((u64)high << 32 | low), 0);
149 }
150
151 /* Can be uninlined because referenced by paravirt */
152 static inline int notrace
153 native_write_msr_safe(unsigned int msr, u32 low, u32 high)
154 {
155         int err;
156
157         asm volatile("1: wrmsr ; xor %[err],%[err]\n"
158                      "2:\n\t"
159                      _ASM_EXTABLE_TYPE_REG(1b, 2b, EX_TYPE_WRMSR_SAFE, %[err])
160                      : [err] "=a" (err)
161                      : "c" (msr), "0" (low), "d" (high)
162                      : "memory");
163         if (tracepoint_enabled(write_msr))
164                 do_trace_write_msr(msr, ((u64)high << 32 | low), err);
165         return err;
166 }
167
168 extern int rdmsr_safe_regs(u32 regs[8]);
169 extern int wrmsr_safe_regs(u32 regs[8]);
170
171 /**
172  * rdtsc() - returns the current TSC without ordering constraints
173  *
174  * rdtsc() returns the result of RDTSC as a 64-bit integer.  The
175  * only ordering constraint it supplies is the ordering implied by
176  * "asm volatile": it will put the RDTSC in the place you expect.  The
```

```
177  * CPU can and will speculatively execute that RDTSC, though, so the
178  * results can be non-monotonic if compared on different CPUs.
179  */
180 static __always_inline unsigned long long rdtsc(void)
181 {
182         DECLARE_ARGS(val, low, high);
183
184         asm volatile("rdtsc" : EAX_EDX_RET(val, low, high));
185
186         return EAX_EDX_VAL(val, low, high);
187 }
188
189 /**
190  * rdtsc_ordered() - read the current TSC in program order
191  *
192  * rdtsc_ordered() returns the result of RDTSC as a 64-bit integer.
193  * It is ordered like a load to a global in-memory counter.  It should
194  * be impossible to observe non-monotonic rdtsc_unordered() behavior
195  * across multiple CPUs as long as the TSC is synced.
196  */
197 static __always_inline unsigned long long rdtsc_ordered(void)
198 {
199         DECLARE_ARGS(val, low, high);
200
201         /*
202          * The RDTSC instruction is not ordered relative to memory
203          * access.  The Intel SDM and the AMD APM are both vague on this
204          * point, but empirically an RDTSC instruction can be
205          * speculatively executed before prior loads.  An RDTSC
206          * immediately after an appropriate barrier appears to be
207          * ordered as a normal load, that is, it provides the same
208          * ordering guarantees as reading from a global memory location
209          * that some other imaginary CPU is updating continuously with a
210          * time stamp.
211          *
212          * Thus, use the preferred barrier on the respective CPU, aiming for
213          * RDTSCP as the default.
214          */
215         asm volatile(ALTERNATIVE_2("rdtsc",
216                                    "lfence; rdtsc", X86_FEATURE_LFENCE_RDTSC,
217                                    "rdtscp", X86_FEATURE_RDTSCP)
218                         : EAX_EDX_RET(val, low, high)
219                         /* RDTSCP clobbers ECX with MSR_TSC_AUX. */
220                         :: "ecx");
221
222         return EAX_EDX_VAL(val, low, high);
223 }
224
225 static inline unsigned long long native_read_pmc(int counter)
```

```
226 {
227         DECLARE_ARGS(val, low, high);
228
229         asm volatile("rdpmc" : EAX_EDX_RET(val, low, high) : "c" (counter));
230         if (tracepoint_enabled(rdpmc))
231                 do_trace_rdpmc(counter, EAX_EDX_VAL(val, low, high), 0);
232         return EAX_EDX_VAL(val, low, high);
233 }
234
235 #ifdef CONFIG_PARAVIRT_XXL
236 #include <asm/paravirt.h>
237 #else
238 #include <linux/errno.h>
239 /*
240  * Access to machine-specific registers (available on 586 and better only)
241  * Note: the rd* operations modify the parameters directly (without using
242  * pointer indirection), this allows gcc to optimize better
243  */
244
245 #define rdmsr(msr, low, high)                                   \
246 do {                                                            \
247         u64 __val = native_read_msr((msr));                     \
248         (void)((low) = (u32)__val);                             \
249         (void)((high) = (u32)(__val >> 32));                    \
250 } while (0)
251
252 static inline void wrmsr(unsigned int msr, u32 low, u32 high)
253 {
254         native_write_msr(msr, low, high);
255 }
256
257 #define rdmsrl(msr, val)                        \
258         ((val) = native_read_msr((msr)))
259
260 static inline void wrmsrl(unsigned int msr, u64 val)
261 {
262         native_write_msr(msr, (u32)(val & 0xffffffffULL), (u32)(val >> 32));
263 }
264
265 /* wrmsr with exception handling */
266 static inline int wrmsr_safe(unsigned int msr, u32 low, u32 high)
267 {
268         return native_write_msr_safe(msr, low, high);
269 }
270
271 /* rdmsr with exception handling */
272 #define rdmsr_safe(msr, low, high)                              \
273 ({                                                              \
274         int __err;                                              \
```

```
275         u64 __val = native_read_msr_safe((msr), &__err);        \
276         (*low) = (u32)__val;                                    \
277         (*high) = (u32)(__val >> 32);                           \
278         __err;                                                  \
279 })
280
281 static inline int rdmsrl_safe(unsigned int msr, unsigned long long *p)
282 {
283         int err;
284
285         *p = native_read_msr_safe(msr, &err);
286         return err;
287 }
288
289 #define rdpmc(counter, low, high)                       \
290 do {                                                    \
291         u64 _l = native_read_pmc((counter));            \
292         (low)  = (u32)_l;                               \
293         (high) = (u32)(_l >> 32);                       \
294 } while (0)
295
296 #define rdpmcl(counter, val) ((val) = native_read_pmc(counter))
297
298 #endif  /* !CONFIG_PARAVIRT_XXL */
299
300 /*
301  * 64-bit version of wrmsr_safe():
302  */
303 static inline int wrmsrl_safe(u32 msr, u64 val)
304 {
305         return wrmsr_safe(msr, (u32)val,  (u32)(val >> 32));
306 }
307
308 struct msr *msrs_alloc(void);
309 void msrs_free(struct msr *msrs);
310 int msr_set_bit(u32 msr, u8 bit);
311 int msr_clear_bit(u32 msr, u8 bit);
312
313 #ifdef CONFIG_SMP
314 int rdmsr_on_cpu(unsigned int cpu, u32 msr_no, u32 *l, u32 *h);
315 int wrmsr_on_cpu(unsigned int cpu, u32 msr_no, u32 l, u32 h);
316 int rdmsrl_on_cpu(unsigned int cpu, u32 msr_no, u64 *q);
317 int wrmsrl_on_cpu(unsigned int cpu, u32 msr_no, u64 q);
318 void rdmsr_on_cpus(const struct cpumask *mask, u32 msr_no, struct msr *msrs);
319 void wrmsr_on_cpus(const struct cpumask *mask, u32 msr_no, struct msr *msrs);
320 int rdmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 *l, u32 *h);
321 int wrmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 l, u32 h);
322 int rdmsrl_safe_on_cpu(unsigned int cpu, u32 msr_no, u64 *q);
323 int wrmsrl_safe_on_cpu(unsigned int cpu, u32 msr_no, u64 q);
```

```
int rdmsr_safe_regs_on_cpu(unsigned int cpu, u32 regs[8]);
int wrmsr_safe_regs_on_cpu(unsigned int cpu, u32 regs[8]);
#else  /*  CONFIG_SMP  */
static inline int rdmsr_on_cpu(unsigned int cpu, u32 msr_no, u32 *l, u32 *h)
{
        rdmsr(msr_no, *l, *h);
        return 0;
}
static inline int wrmsr_on_cpu(unsigned int cpu, u32 msr_no, u32 l, u32 h)
{
        wrmsr(msr_no, l, h);
        return 0;
}
static inline int rdmsrl_on_cpu(unsigned int cpu, u32 msr_no, u64 *q)
{
        rdmsrl(msr_no, *q);
        return 0;
}
static inline int wrmsrl_on_cpu(unsigned int cpu, u32 msr_no, u64 q)
{
        wrmsrl(msr_no, q);
        return 0;
}
static inline void rdmsr_on_cpus(const struct cpumask *m, u32 msr_no,
                                 struct msr *msrs)
{
        rdmsr_on_cpu(0, msr_no, &(msrs[0].l), &(msrs[0].h));
}
static inline void wrmsr_on_cpus(const struct cpumask *m, u32 msr_no,
                                 struct msr *msrs)
{
        wrmsr_on_cpu(0, msr_no, msrs[0].l, msrs[0].h);
}
static inline int rdmsr_safe_on_cpu(unsigned int cpu, u32 msr_no,
                                    u32 *l, u32 *h)
{
        return rdmsr_safe(msr_no, l, h);
}
static inline int wrmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 l, u32 h)
{
        return wrmsr_safe(msr_no, l, h);
}
static inline int rdmsrl_safe_on_cpu(unsigned int cpu, u32 msr_no, u64 *q)
{
        return rdmsrl_safe(msr_no, q);
}
static inline int wrmsrl_safe_on_cpu(unsigned int cpu, u32 msr_no, u64 q)
{
        return wrmsrl_safe(msr_no, q);
```

```
373 }
374 static inline int rdmsr_safe_regs_on_cpu(unsigned int cpu, u32 regs[8])
375 {
376         return rdmsr_safe_regs(regs);
377 }
378 static inline int wrmsr_safe_regs_on_cpu(unsigned int cpu, u32 regs[8])
379 {
380         return wrmsr_safe_regs(regs);
381 }
382 #endif  /* CONFIG_SMP */
383 #endif /* __ASSEMBLY__ */
384 #endif /* _ASM_X86_MSR_H */
```