Transactional Memory support

POWER kernel support for this feature is currently limited to supporting its use by user programs. It is not currently used by the kernel itself.

This file aims to sum up how it is supported by Linux and what behaviour you can expect from your user programs.

Basic overview

Hardware Transactional Memory is supported on POWER8 processors, and is a feature that enables a different form of atomic memory access. Several new instructions are presented to delimit transactions; transactions are guaranteed to either complete atomically or roll back and undo any partial changes.

A simple transaction looks like this:

```
begin_move_money:
  tbegin
  beq   abort_handler

  ld    r4, SAVINGS_ACCT(r3)
  ld    r5, CURRENT_ACCT(r3)
  subi  r5, r5, 1
  addi  r4, r4, 1
  std   r4, SAVINGS_ACCT(r3)
  std   r5, CURRENT_ACCT(r3)

  tend

  b     continue

abort_handler:
  ... test for odd failures ...

  /* Retry the transaction if it failed because it conflicted with
   * someone else: */
  b     begin_move_money
```

The 'tbegin' instruction denotes the start point, and 'tend' the end point. Between these points the processor is in 'Transactional' state; any memory references will complete in one go if there are no conflicts with other transactional or non-transactional accesses within the system. In this example, the transaction completes as though it were normal straight-line code IF no other processor has touched SAVINGS_ACCT(r3) or CURRENT_ACCT(r3); an atomic move of money from the current account to the savings account has been performed. Even though the normal ld/std instructions are used (note no lwarx/stwcx), either *both* SAVINGS_ACCT(r3) and CURRENT_ACCT(r3) will be updated, or neither will be updated.

If, in the meantime, there is a conflict with the locations accessed by the transaction, the transaction will be aborted by the CPU. Register and memory state will roll back to that at the 'tbegin', and control will continue from 'tbegin+4'. The branch to abort_handler will be taken this second time; the abort handler can check the cause of the failure, and retry.

Checkpointed registers include all GPRs, FPRs, VRs/VSRs, LR, CCR/CR, CTR, FPCSR and a few other status/flag regs; see the ISA for details.

Causes of transaction aborts

- Conflicts with cache lines used by other processors
- Signals
- Context switches
- See the ISA for full documentation of everything that will abort transactions.

Syscalls

Syscalls made from within an active transaction will not be performed and the transaction will be doomed by the kernel with the failure code TM_CAUSE_SYSCALL | TM_CAUSE_PERSISTENT.

Syscalls made from within a suspended transaction are performed as normal and the transaction is not explicitly doomed by the kernel. However, what the kernel does to perform the syscall may result in the transaction being doomed by the hardware. The syscall is performed in suspended mode so any side effects will be persistent, independent of transaction success or failure. No guarantees are provided by the kernel about which syscalls will affect transaction success.

Care must be taken when relying on syscalls to abort during active transactions if the calls are made via a library. Libraries may cache values (which may give the appearance of success) or perform operations that cause transaction failure before entering the kernel (which may produce different failure codes). Examples are glibc's getpid() and lazy symbol resolution.

Signals

Delivery of signals (both sync and async) during transactions provides a second thread state (ucontext/mcontext) to represent the second transactional register state. Signal delivery 'treclaim's to capture both register states, so signals abort transactions. The usual ucontext_t passed to the signal handler represents the checkpointed/original register state; the signal appears to have arisen at 'tbegin+4'.

If the sighandler ucontext has uc_link set, a second ucontext has been delivered. For future compatibility the MSR.TS field should be checked to determine the transactional state -- if so, the second ucontext in uc->uc_link represents the active transactional registers at the point of the signal.

For 64-bit processes, uc->uc_mcontext.regs->msr is a full 64-bit MSR and its TS field shows the transactional mode.

For 32-bit processes, the mcontext's MSR register is only 32 bits; the top 32 bits are stored in the MSR of the second ucontext, i.e. in uc->uc_link->uc_mcontext.regs->msr. The top word contains the transactional state TS.

However, basic signal handlers don't need to be aware of transactions and simply returning from the handler will deal with things correctly:

Transaction-aware signal handlers can read the transactional register state from the second ucontext. This will be necessary for crash handlers to determine, for example, the address of the instruction causing the SIGSEGV.

Example signal handler:

```
    void crash_handler(int sig, siginfo_t *si, void *uc)
    {
      ucontext_t *ucp = uc;
      ucontext_t *transactional_ucp = ucp->uc_link;

      if (ucp_link) {
        u64 msr = ucp->uc_mcontext.regs->msr;
        /* May have transactional ucontext! */
#ifndef __powerpc64__
        msr |= ((u64)transactional_ucp->uc_mcontext.regs->msr) << 32;
#endif
        if (MSR_TM_ACTIVE(msr)) {
           /* Yes, we crashed during a transaction.  Oops. */
 fprintf(stderr, "Transaction to be restarted at 0x%llx, but "
                         "crashy instruction was at 0x%llx\n",
                         ucp->uc_mcontext.regs->nip,
                         transactional_ucp->uc_mcontext.regs->nip);
        }
      }

      fix_the_problem(ucp->dar);
    }
```

When in an active transaction that takes a signal, we need to be careful with the stack. It's possible that the stack has moved back up after the tbegin. The obvious case here is when the tbegin is called inside a function that returns before a tend. In this case, the stack is part of the checkpointed transactional memory state. If we write over this non transactionally or in suspend, we are in trouble because if we get a tm abort, the program counter and stack pointer will be back at the tbegin but our in memory stack won't be valid anymore.

To avoid this, when taking a signal in an active transaction, we need to use the stack pointer from the checkpointed state, rather than the speculated state. This ensures that the signal context (written tm suspended) will be written below the stack required for the rollback. The transaction is aborted because of the treclaim, so any memory written between the tbegin and the signal will be rolled back anyway.

For signals taken in non-TM or suspended mode, we use the normal/non-checkpointed stack pointer.

Any transaction initiated inside a sighandler and suspended on return from the sighandler to the kernel will get reclaimed and discarded.

Failure cause codes used by kernel

These are defined in <asm/reg.h>, and distinguish different reasons why the kernel aborted a transaction:

TM_CAUSE_RESCHED	Thread was rescheduled.
TM_CAUSE_TLBI	Software TLB invalid.
TM_CAUSE_FAC_UNAV	FP/VEC/VSX unavailable trap.
TM_CAUSE_SYSCALL	Syscall from active transaction.
TM_CAUSE_SIGNAL	Signal delivered.
TM_CAUSE_MISC	Currently unused.
TM_CAUSE_ALIGNMENT	Alignment fault.

TM_CAUSE_EMULATE	Emulation that touched memory.

These can be checked by the user program's abort handler as TEXASR[0:7]. If bit 7 is set, it indicates that the error is considered persistent. For example a TM_CAUSE_ALIGNMENT will be persistent while a TM_CAUSE_RESCHED will not.

GDB

GDB and ptrace are not currently TM-aware. If one stops during a transaction, it looks like the transaction has just started (the checkpointed state is presented). The transaction cannot then be continued and will take the failure handler route. Furthermore, the transactional 2nd register state will be inaccessible. GDB can currently be used on programs using TM, but not sensibly in parts within transactions.

POWER9

TM on POWER9 has issues with storing the complete register state. This is described in this commit:

```
commit 4bb3c7a0208fc13ca70598efd109901a7cd45ae7
Author: Paul Mackerras <paulus@ozlabs.org>
Date:   Wed Mar 21 21:32:01 2018 +1100
KVM: PPC: Book3S HV: Work around transactional memory bugs in POWER9
```

To account for this different POWER9 chips have TM enabled in different ways.

On POWER9N DD2.01 and below, TM is disabled. ie HWCAP2[PPC_FEATURE2_HTM] is not set.

On POWER9N DD2.1 TM is configured by firmware to always abort a transaction when tm suspend occurs. So tsuspend will cause a transaction to be aborted and rolled back. Kernel exceptions will also cause the transaction to be aborted and rolled back and the exception will not occur. If userspace constructs a sigcontext that enables TM suspend, the sigcontext will be rejected by the kernel. This mode is advertised to users with HWCAP2[PPC_FEATURE2_HTM_NO_SUSPEND] set. HWCAP2[PPC_FEATURE2_HTM] is not set in this mode.

On POWER9N DD2.2 and above, KVM and POWERVM emulate TM for guests (as described in commit 4bb3c7a0208f), hence TM is enabled for guests ie. HWCAP2[PPC_FEATURE2_HTM] is set for guest userspace. Guests that makes heavy use of TM suspend (tsuspend or kernel suspend) will result in traps into the hypervisor and hence will suffer a performance degradation. Host userspace has TM disabled ie. HWCAP2[PPC_FEATURE2_HTM] is not set. (although we make enable it at some point in the future if we bring the emulation into host userspace context switching).

POWER9C DD1.2 and above are only available with POWERVM and hence Linux only runs as a guest. On these systems TM is emulated like on POWER9N DD2.2.

Guest migration from POWER8 to POWER9 will work with POWER9N DD2.2 and POWER9C DD1.2. Since earlier POWER9 processors don't support TM emulation, migration from POWER8 to POWER9 is not supported there.

Kernel implementation

h/rfid mtmsrd quirk

As defined in the ISA, rfid has a quirk which is useful in early exception handling. When in a userspace transaction and we enter the kernel via some exception, MSR will end up as TM=0 and TS=01 (ie. TM off but TM suspended). Regularly the kernel will want change bits in the MSR and will perform an rfid to do this. In this case rfid can have SRR0 TM = 0 and TS = 00 (ie. TM off and non transaction) and the resulting MSR will retain TM = 0 and TS=01 from before (ie. stay in suspend). This is a quirk in the architecture as this would normally be a transition from TS=01 to TS=00 (ie. suspend -> non transactional) which is an illegal transition.

This quirk is described the architecture in the definition of rfid with these lines:

```
if (MSR 29:31 Â¬ = 0b010 | SRR1 29:31 Â¬ = 0b000) then
        MSR 29:31 <- SRR1 29:31
```

hrfid and mtmsrd have the same quirk.

The Linux kernel uses this quirk in its early exception handling.