# IBM Education Assistance for z/OS V2R1

Item: Transactional Execution
Element/Component: BCP Supervisor

Material is current as of March 2013

# Agenda

- Trademarks
- Presentation Objectives
- Overview
- Usage & Invocation
- Interactions & Dependencies
- Installation
- Appendix

# Trademarks

- See url http://www.ibm.com/legal/copytrade.shtml for a list of trademarks.

# Presentation Objectives

Describe z/OS changes with respect to the Transactional Execution Facility of the zEC12 and zBC12.

# Overview

- Problems:
  - zEC12 and zBC12 provides the Transactional Execution Facility (think of it as TX but you may see TX or HTM – Hardware Transactional Memory)
- Solution:
  - z/OS provides the infrastructure so that z/OS itself, middleware, and applications can exploit it
- Benefit / Value:
  - Hardware exploitation

# Usage & Invocation

- APAR OA38829 was shipped for z/OS 1.13 to coincide with zEC12 GA. It provided minimal infrastructure for Java and for testing new usage in anticipation of the z/OS 2.1 support. z/OS 1.13 is considered a “test environment” for transactional execution exploitation and should not be used for production code.
- Primarily what was missing was diagnostic support that is truly needed in order to debug either development or production code problems when using TX

# What is TX and a Transaction?

- Hardware-based facility, new with zEnterprise EC 12 and zBC12
- Supports a “transaction” which may be thought of as a sequence of instructions that act atomically
- New instructions TBEGIN, TBEGINC, TABORT, TEND, PPA, NTSTG
- A transaction completes successfully or aborts
- Upon abort,
  - Storage (aside from NTSTG) is unchanged to the program compared to time-of-begin, except for a transaction diagnostic block (TDB)
  - GRs may be unchanged to the program
  - ARs are not unchanged
  - FPRs are not unchanged
- Full availability when bits CVTTX and CVTTXC are on (z/OS 2.1), available for testing when CVTTXTE is on (z/OS 1.13)

# TX (cont)

- Special rules are applied to instructions executed within TX mode.
- PoOp has a section “Restricted Instruction” in chapter 5 that lists those instructions not allowed within a transaction. Loosely, chapter 7 (general) instructions are OK; chapter 10 (control) instructions are not
- The instructions are treated as block-concurrent (as observed by other CPUs and the channel subsystem), with the TX facility providing the serialization that you might otherwise implement yourself
- Two transactions “conflict” in many ways. One is that both need to access the same cache line and at least one needs to write to that line. Not all conflicts can be controlled by the application
- Upon a conflict, the transaction cannot complete successfully but might succeed if retried.
- When no conflict exists, one processor might be able to complete in a simple way, without having to obtain software-managed serialization or utilizing serializing instructions

# TX (cont)

- You can think of stores within a transaction as being "rolled back" upon transaction abort.
- Similarly, you can think of the registers as being saved at the transaction begin and then (optionally) rolled back to their pre-transaction begin values.

# Simple transaction

```
          LA    2,Source_Data_Word
          LA    3,Target_Data_Word
          TBEGIN theTDB,X'8000'  the "80" indicates to restore
*                                GRs 0-1
*                                upon abort, each bit in those 2
*                                hex digits corresponds to a
*                                double register pair
          BRC   7,Transaction_aborted
          L     1,0(,2)
          ST    1,0(,3)
          TEND
```

<<when you get here, register 1 will have been changed by the "L", and the target word will have been set>>

```
          …
Transaction_aborted DS 0H
```

<<when you get here, all the registers will have the value they had before the TBEGIN instruction (0-1 restored, 2-F not used), the target word will be unchanged, and the TDB, identified on the TBEGIN instruction, will contain information about the transaction abort>>

# Types of Transactions

- There are two types of transactions
  - Non-Constrained
  - Constrained

# Non-constrained Transactions

- Begins with TBEGIN
- Ends normally with TEND
- Can be aborted with TABORT
- May be aborted for many system-defined reasons
- TBEGIN may identify a “transaction diagnostic block” (TDB, mapped by IHATDB)
- Upon abort, flow proceeds to the instruction after TBEGIN

# Non-constrained Transactions (Cont)

The instruction after TBEGIN is usually a conditional relative branch to handle the CC's from TBEGIN completion

- CC=0 (transaction initiation successful) should "fall through".
- CC=1 (abort due to an indeterminate condition) should branch somewhere to deal with this situation.
- CC=2 (abort due to a transient condition) should branch somewhere to deal with this situation (dealing with this might retry, but should eventually "time out" and go to the fall-back path). While there is no "right number" for the question "how many times should I retry", “6” is the recommended number as a default.
- CC=3 (abort due to a persistent condition) should branch somewhere to deal with this situation, eventually winding up in a "fall-back" path because, for some reason, the system believes that the transaction is unlikely ever to succeed. (Note that this applies to the current circumstances. For example, a list search on a hash table might not succeed for the current hash, but for most other hashes might succeed. ).

# Transaction Diagnostic Block (TDB)

- 256 byte area, mapped by IHATDB
- Bytes 8-F: Transaction abort code
  - External Interrupt (2), Program Interrupt (4), I/O Interrupt (6), Lots of codes for overflows and cache conflicts
- Bytes 18-1F: Aborted transaction instruction address
- Byte 20: Exception access ID
- Byte 21: Data exception code
- Bytes 24-27: Program Interruption ID
- Bytes 28 – 2F: Translation exception ID
- Bytes 40 – 47: Breaking Event Address
- Bytes 80 – FF: 64-bit GRs 0-F

# Constrained Transactions

- Begins with TBEGINC
- Ends normally with TEND
- May be aborted for many system-defined reasons
- Upon abort, flow proceeds to the TBEGINC

**In the absence of repeated constraint violations, a constrained transaction is assured of eventual completion. Thus it needs no fallback path.**

# Constrained Transaction Constraints

The transaction executes no more than 32 instructions.

- All instructions within the transaction must be within 256 contiguous bytes of storage.
- The only branches you may use are relative branches that branch forward (so there can be no loops).
- No SS- or SSE-format instructions may be used.
- Additional general instructions may not be used.
- The transaction's storage operands may not access more than four octowords.
- The transaction may not access storage operands in any 4 K-byte blocks that contain the 256 bytes of storage beginning with the TBEGINC instruction.
- Operand references must be within a single doubleword, except for some of the "multiple" instructions for which the limitation is a single octoword.

# Simple constrained transaction

```
LA     2,Source_Data_Word
LA     3,Target_Data_Word
TBEGINC 0,X'8000'
L      1,0(,2)
ST     1,0(,3)
TEND
```

<<when you get here, register 1 will have been changed by the "L", and the target word will have been set>>

The transaction may have aborted one or more times but at some point it succeeded.

The TBEGINC asked that register pair 0/1 be restored, but there was no need to do so since the transaction did not depend on the initial values of those registers, so the operand could have been X'0000'.

# User controls over a transaction

The user can control

- The general register pairs that are to be saved at the initiation of a transaction (TBEGIN or TBEGINC instruction), and restored upon a transaction abort.
- Whether access register modification is allowed within the transaction. Note: upon transaction abort, access register values are never restored.
- Whether floating point operations are allowed within the non-constrained transaction. Note: upon transaction abort, floating point register values are never restored.
- Program interrupt filtering for a non-constrained transaction. For example, the application may ask that certain classes of program interrupts be presented to the application as an abort, rather than processed by the system as a program interrupt.
- Whether data is returned to provide information about the abort (Transaction Diagnostic Block – TDB, mapped by IHATDB)

# Program Interruption Info

- New code x'18' – transaction constraint exception
  - –Machine may or may not issue; the machine is allowed to ignore these conditions. Assume that it is not predictable if you will get a transaction constraint exception.
  - –You must not rely on its doing so
  - –Results in z/OS System Code 0E0-18
- X'0200' bit in the program interrupt code indicates that it occurred within TX
  - –e.g., x'0211' – page fault while within a transaction
- Program Interrupt TDB (PITDB) saved at location x'1800' upon a program interruption while within a transaction. Contains information about register and instruction address at the time of the program interruption. This is not saved by z/OS 1.13.

# Non-transactional store

- NTSTG instruction
  - A store that is not rolled back upon abort and thus can be examined after the transaction ends
  - Perhaps counts number of tries (this could also be in a register)

# HLASM

- HLASM support for the instructions will be provided via PM49761
- HLASM provides a print exit named ASMAXTXP, which you can use in your assembly. It will be available via PM66334
  - It flags as errors things that violate a transactional execution restriction, to the extent it can determine those violations.
  - It is primarily detected constrained-transaction constraints.

# Transaction / Fallback considerations

- When using non-constrained transactions, the transaction must be serialized against the fall-back path.
- For example, if one processor is within a transaction and another within the fall-back path, each needs to know enough to protect itself against the other.
- Also, typically, a fall-back path needs to be serialized against other concurrent execution of that fall-back path.
- You can think of the fall-back path as needing "real" serialization (hardware or software-provided) and the transaction needing to be able to tell that the fall-back path is running.
- One way of accomplishing this is for the fall-back path to set a footprint when it has obtained "real" serialization and for the transaction to query that footprint.

# Considerations (Cont)

## Transaction

```
TBEGIN
BRC   7,Transaction_Aborted
If I_Have_ENQ is on then TABORT
the group of updates
TEND
```

## Fallback_Path

```
ENQ
Set bit I_Have_ENQ
the group of updates
Reset bit I_Have_ENQ
DEQ
```

# Considerations (Cont)

- Even this simple example is incomplete.
- To avoid cases where a spin would result (when the ENQ holder does not get a chance to reset the bit), the transaction path needs to limit the number of times that the transaction is started over.
- This can be done using a counter in a register that is not restored upon the abort, or a non-transactional store.
- Thus, at "Transaction_Aborted", there might be a test to see if flow should proceed to the fall-back path or back to the TBEGIN.
- IBM suggests limiting the number of times to retry on CC=2 to 6.
- The architecture also provides a Perform Processor Assist instruction which should be used prior to retrying the transaction. Applying these concepts yields the transaction on the next slide.

# Considerations (Cont)

```
          LHI   15,0   Zero count of transaction aborts
Transaction_Again DS 0H
          TBEGIN
          BRC 7,Transaction_Aborted
          If I_Have_ENQ is on then TABORT
          the group of updates
          TEND
          …
Transaction_Aborted DS 0H
          JC    5,Fallback_Path Not worth retrying for
*             CC=1,CC=3
          AHI   15,1    One more transaction abort
          CIJNL 15,6,Fallback_Path Give up after 6 tries
          PPA   15,0,1  Perform Processor Assist option
*                       1, count in GR15
          J     Transaction_Again
          ...
Fallback_Path DS 0H
```

# Roll-out

- z/OS 1.13 via OA38829: TX Lite
  - Bit CVTTXTE “TX Test Environment”
  - No particular diagnostics (but TDB may be used, mapped by IHATDB). Program interrupt does write TDB into PSA as architected at location x'1800' but this is not captured
  - Transactions work (both non-constrained and constrained)
  - Recommend not using in production
- z/OS 2.1
  - Bits CVTTX and CVTTXC “Full support”
  - PI TDB is captured into SDWA
  - SDWA has both “time of program interrupt regs/psw” and “transaction abort regs/PSW”
  - EPIE for ESPIE service has both sets too

# TX Diagnostics (z/OS 2.1 only)

- If a program interrupt in a transaction is not filtered, normal z/OS recovery processing takes place
- Additional information in an SDWA (all FRR SDWAs, SDWALOC31=YES ESTAE-type SDWAs)
- Bits SDWAPTX1 (within byte SDWAIC1H in field SDWAAEC1) and SDWAPTX2 (within byte SDWAIC2H in field SDWAAEC2): The program interrupt occurred while within transactional execution and therefore bit SDWAPTX1 is valid only when bit SDWAPCHK is on.
- Existing fields SDWAG64, SDWAG64H, and SDWAGRSV contain the time of error register information. These are the registers current when the program interrupt occurred .

# TX Diagnostics (z/OS 2.1 only) (Cont)

- Existing field SDWAPSW16 contains the time of error register information. This is the PSW current when the program interrupt occurred.
- When bits SDWAPCHK and SDWAPTX2 are on, new field SDWATXG64 contains the registers that resulted from the transaction abort due to the program interrupt.
- When bits SDWAPCHK and SDWAPTX2 are on, new field SDWATXPSW16 contains the PSW that resulted from the transaction abort due to the program interrupt.

# Random Aborts (z/OS 2.1 only)

- To help test your code (both transaction and fallback) random aborts can be requested
- With IEATXDC, you can request random aborts of transactions for your task so that, upon repeated runnings, you are likely to exercise both the non-abort and abort paths.
- IEATXDC SCOPE={PROBLEM | ALL},

  OPERATION={NO_ABORT, SET_EVERY, SET_RANDOM}
- Implemented by bits in the DUCT
- DIAGxx option available for having aborts cover every work unit in the system (SRBs too)
  - IEATXABEVERY
  - IEATXABRANDOM

  Probably don't want to use these unless running in a sandbox

# Other interesting parts of the TX architecture

- PER event suppression
- PER Transaction End event

- Consider a SLIP trap in effect with a PER range covering a constrained transaction.
- Every time through the transaction, the SLIP trap hits, the PER event aborts the transaction, and the abort process resumes at the TBEGINC

- System recognizes “PER event within constrained transaction”
- System sets “PER Event Suppression” so that subsequent PER events within a transaction are suppressed
- System sets “PER Transaction End event” so that the TEND will be noticed and can turn off Event Suppression

# SLIP TXIGD (z/OS 2.1 only)

- Suppose you have a SLIP trap with a DATA keyword looking to see if storage has changed to a bad value
- If the bad value change occurs within a transaction, by the time SLIP sees things, storage no longer looks bad (it has been rolled back)
- Best that can be done: capture diagnostic data in the hope that this is the cause
  - TXIGD keyword (Transactional Execution Ignore Data) on SLIP: if the only thing keeping the trap from matching is the data keyword, ignore the data keyword
- NOTXIGD may be specified to indicate the other case

# More SLIP (z/OS 2.1 only)

- A new ERRTYP option, TXPROG, is supported. When specified, the SLIP trap will match only if the event was a program interrupt error event that occurred within transactional execution mode
- The SLIP DISPLAY of an active trap will display, when non-0, the number of times that the SLIP trap was examined for an event in TX, but did not match because of the DATA keyword (in a transactional execution case, this could be normal, because the stores were rolled back when the error or PER program interrupt occurred). This is referred to as the transactional execution DATA filter mismatch count (Note: if the value is 255, the count of events could have exceeded 255).
- The SLIP GTF record, at offset (decimal) 135, has a 1-byte count that is the transactional execution DATA filter mismatch count ,

# Interactions & Dependencies

- Software Dependencies
  - Use TX only if bits CVTTX and CVTTXC are on
  - You cannot use TX for z/OS running under VM
- Hardware Dependencies
  - None
- Exploiters
  - Java exploits TX using the z/OS 1.13 support
  - z/OS itself exploits TX in JBB778H (Flash support)

# Installation

- Install HLASM PM49761 and PM66334 to have access to the instructions and the print exit.

# Appendix

- Publications:
  - MVS System Commands (SLIP) SA38-0666-00
  - MVS Assembler Services References (IEATXC) SA23-1370-00
  - z/Architecture Principles of Operation SA22-7832-09