IBM Systems & Technology Group

Transactional Execution Facility z/OS 1.13, 2.1

Session 14255

Peter Relson
IBM Poughkeepsie
relson@us.ibm.com
14 August 2013

Trademarks

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.

AIX*	FlashCopy*	Parallel Sysplex*	System Storage	z10
CICS*	HiperSockets	ProductPac*	System z	z10 BC
DB2*	IBM*	RACF*	System z9	z10 EC
DFSMSdss	IBM eServer	Redbooks*	System z10	z/OS*
DFSMShsm	IBM logo*	REXX	System z10 Business Class	zSeries*
DFSMSrmm	IMS	RMF	Tivoli*	
DS6000	Infiniband*	ServerPac*	WebSphere*	
DS8000*	Language Environment*	SystemPac*	z9*	
FICON*				

* Registered trademarks of IBM Corporation

The following are trademarks or registered trademarks of other companies.

InfiniBand is a registered trademark of the InfiniBand Trade Association (IBTA).
Intel is a trademark of the Intel Corporation in the United States and other countries.
Linux is a trademark of Linux Torvalds in the United States, other countries, or both.
Java and all Java-related trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.
Microsoft, Windows and Windows NT are registered trademarks of Microsoft Corporation.
UNIX is a registered trademark of The Open Group in the United States and other countries.
All other products may be trademarks or registered trademarks of their respective companies.
The Open Group is a registered trademark of The Open Group in the US and other countries.

Notes:

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

This presentation and the claims outlined in it were reviewed for compliance with US law. Adaptations of these claims for use in other geographies must be reviewed by the local country counsel for compliance with local laws.

Abstract

- This presentation will cover the transactional execution facility of the zEC12 machine, its introduction in z/OS 1.13 and its full availability in z/OS 2.1

Transactional eXecution

- I think of it as “TX”. You may see “TE” or “HTM” (Hardware Transactional Memory)
- What is it?
- Why is it a good thing?
- How do you use it?
- Diagnostics

What is TX and a transaction?

- **The Transactional Execution facility is a hardware-based facility, new with zEnterprise EC 12**
  - **It supports a “transaction” which may be thought of as a sequence of instructions that act atomically**
  - **Transactions “begin” and “end”**
  - **A transaction may “abort” (does not reach the “end”), often due to an environmental consideration. Some sort of “conflict” occurred (might be with another transaction, might not)**
  - **When within a transaction, you are in transactional execution mode**

Why is transactional execution a good thing?

- Most programs need serialization at one time or other
- Getting and releasing serialization takes cycles
- If there is no one contending, you've just wasted time
- Wouldn't it be great if you could just “try it” and if you find there's some one contending, then (and only then) do you get the serialization
- And wouldn't it be great if whatever you tried “didn't count” if you found contention?
- Our normal locks and ENQs are often not fine-grained enough to prevent “false contention”. TX may let you accomplish fine-grained serialization
- Unlike PLO, all statements within a transaction can serialize against non-transactional statements

TX (Cont)

- 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
- New instructions TBEGIN, TBEGINC, TABORT, TEND, PPA, NTSTG, ETND
- Full availability when bits CVTTX and CVTTXC are on (z/OS 2.1), available for testing when CVTTXTE is on (z/OS 1.13)
- Not available if z/OS is a guest of z/VM (the CVT bit(s) will not be on)

TX (cont)

- Special rules are applied to instructions executed within TX mode.
- PoOp has a section “Restricted Instructions” 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

TX (cont)

- 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
- 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.

Types of Transactions

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

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.**
- **For non-constrained transaction, whether data is returned to provide information about the abort (Transaction Diagnostic Block – TDB, mapped by IHATDB)**

More user controls over a transaction

The user can control

- **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 nonconstrained 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.**

Non-constrained Transactions

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

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>>

Non-constrained Transactions (Cont)

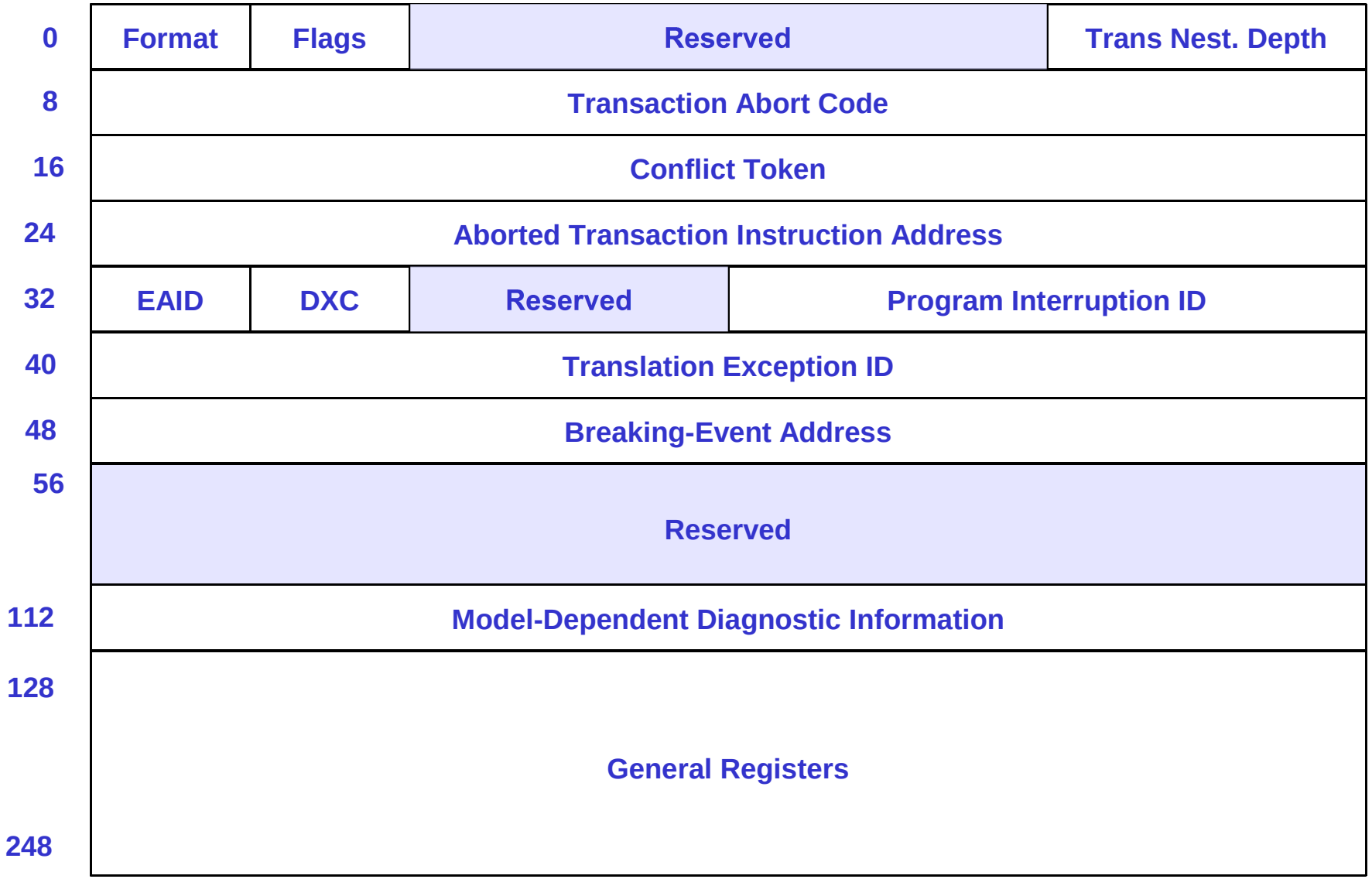
The instruction after TBEGIN is usually a conditional relative branch to handle the CC's from TBEGIN completion

- **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

TDB Contents:



Constrained Transactions

- Begin with TBEGINC
- End 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.

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 (should) have been X'0000'.

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.

Constrained Transaction Constraints (cont)

- 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.
- Neither instructions nor operands may use different logical addresses that are mapped to the same absolute address

Constrained Transactions (cont)

- Even with all of these constraints, with care, you can do useful things.
- It is possible to implement a double-headed double-threaded enqueue/dequeue protocol (particularly one that always adds either to the front or back of the queue)
- This is a protocol that ordinarily requires more formal serialization

Program Interruption Info

- New program interrupt code x'18' – transaction constraint exception
  - Machine may or may not issue; the machine is allowed to ignore these conditions and not issue the program interrupt.
  - You must not rely on the machine ignoring
- 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 detecting constrained-transaction constraints.
  - To your assembly parameters, you would add EX(PRX(ASMAXTXP))'

HLASM (cont)

Some sample warning messages via ASMAXTXP:

- **** ASMA701W LISTING: ASMAXTXP: Transaction exceeds total byte limit.**
- **** ASMA701W LISTING: ASMAXTXP: Relative branch target is zero or negative.**
- **** ASMA701W LISTING: ASMAXTXP: Instruction is restricted.**

Extending existing code to use TX

Consider a section of code that is serialized by an ENQ

```
ENQ
the group of updates
DEQ
```

Extending existing code to use TX (cont)

```
         TM    CVTFLAG4,CVTTX
         JZ    FALLBACK_PATH
         TBEGIN
         JNZ Deal_With_Abort
         the group of updates
         TEND
...
FALLBACK_PATH DS  0H (also the no-TX path)
         ENQ
         the group of updates
         DEQ
```

Not so fast, this isn't right

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 as 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.

Extending existing code to use TX (cont)

```
         TM    CVTFLAG4,CVTTX
         JZ    FALLBACK_PATH
         TBEGIN
         JNZ Deal_With_Abort
         If I_Have_ENQ is on then TABORT
         the group of updates
         TEND
...
FALLBACK_PATH DS  0H  (Also no-TX 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 by using 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

Considerations (Cont)

- Transaction

```
          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 attempts
          PPA   15,0,1  Perform Processor Assist option 1,
*                       count in GR15
          J     Transaction_Again
          ...
Fallback_Path DS 0H
```

Considerations (Cont)

- **The technique of using a footprint for the transaction to detect the fallback path is better than doing something like checking “is the ENQ held” or “is the lock held” because the footprint might be very granular for your particular case and the lock might not be.**

Fine-grained

- Consider the previous example, but with SETLOCK to obtain/release the LOCAL lock instead of ENQ/DEQ.
- The LOCAL lock serializes many things, including the allocation of private storage, ECBs via WAIT/POST, etc. Some of these you might care about, others you might not.
- If you need serialization at an address space level similar to that provided by the LOCAL lock but can accomplish your operation with a transaction, then in that case you do not need to be competing for the LOCAL lock with processes you do not care about. Only your fallback path would compete.
- Thus you might not have to be as concerned about the overall LOCAL lock contention for the space.

Are transactions for me?

The good news

- **If you have “existing code” you have a built-in fallback path.**
- **Your code could check if TX is available (via CVTTX) and if not, go directly to the fallback path**
- **Otherwise it might try doing the operation with a transaction, serializing against the fallback back that another asynchronous process might be in the midst of**
  - **If the transaction aborts too often, go to the fallback path**

Thus your “addition” is only the transactional equivalent of your old code

Are transactions for me (cont)?

The bad news

- **If you have too many aborts you have wasted time**
- **It's next to impossible (in my opinion) to accurately calculate the abort rate because some aborts are not dependent only on some instances of your code interfering with others**
  - **It might be other work units**
  - **It might even be other LPARs**
- **Experimentation / prototyping is very important**

Are transactions for me (cont)?

The good news

- **If you cannot afford to get serialization for whatever reason, you may be able to use constrained transactions and accomplish things that are not possible without them**
- **You may be able to attain higher granularity of your serialization technique**

Are transactions for me (cont)?

The bad news

- **To attain best granularity, even beyond avoiding contending with unrelated work, but to avoid contending with yourself if possible, you may have to adjust your data structures to allow for finer checking.**
  - Instead of a single linked list, you might have an ordered hash table

Are transactions for me (cont)?

The good news

- **It may not be trivial or even easy, but “possible” is a far better answer than “not possible”.**

A tidbit: instructions within TX vs non-TX

Suppose you need to “add 1 to X” with serialization

CS approach

```
          L     old,X
 Again    DS    0H
          LR    new,old
          AHI   new,1
          CS    old,new,X
          JNZ   Again
```

TX approach

```
          TBEGINC
          L     temp,X
          AHI   temp,1
          ST    temp,X
          TEND
```

The transactional approach can be a bit more straightforward. You are allowed to use CS but need not.

My TX usage recommendations

- **Keep it as simple as you can**
  - **You are allowed to do program interrupt filtering. Java may do this. I recommend against using this function. z/OS does not prevent its use. The complexity involved with using it is probably not worth it.**
  - **You can nest transactions within transactions (in that case, an abort goes to the outermost TBEGIN+4). Don't do it. It would be foolish to make a call within a transaction if you do not know exactly what that target may do. And if you do know what that target will do, you should probably put its code inline within your transaction.**
  - **If you can meet the constraints, use constrained transactions**

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 – very hard to diagnose
- z/OS 2.1
  - Bits CVTTX and CVTTXC “Full support”
  - PI TDB is captured and placed into SDWA
  - SDWA has both “time of program interrupt regs/psw” and “transaction abort regs/PSW”
  - For ESPIE, macro IHAEPIE's EPIE has both sets too

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 Dispatchable Unit Control Table (DUCT)

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; therefore bit SDWAPTX1 is valid only when bit SDWAPCHK is on.

TX Diagnostics (z/OS 2.1 only)

- Existing fields SDWAG64, SDWAG64H, and SDWAGRSV contain the time of error register information. These are the registers current when the program interrupt occurred .
- 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.

SLIP (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)

More SLIP (z/OS 2.1 only)

- 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,

SLIP TXIGD (z/OS 2.1 only)

- The best that can be done: capture diagnostic data in the hope that this is the cause
  - TXIGD keyword (Transactional eXecution IGnore Data) on a SLIP trap: if the only thing keeping the trap from matching is the data keyword, ignore the data keyword
- NOTXIGD may be specified to indicate the normal case
- You can modify an existing SLIP trap to set (TXIGD) or reset (NOTXIGD)
  - SLIP MOD,ID=MYTX,TXIGD
  - SLIP MOD,ID=MYTX,NOTXIGD

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

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
- This could loop forever if nothing is done about it.

- 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

Summary

- Transactional Execution is here
- It can be a powerful tool for your applications
- Coding a transaction is easy; providing for fallback less so (constrained transactions are constrained, but need no fallback)
- The benefit depends on the amount of contention that occurs

References

Related Session(s)

- **Session s14087 – zEC12 CPU Facilities**

Publications

- **z/OS V2R1 MVS System Commands**
- **z/OS V2R1 MVS Programming Assembler Services Reference**

Questions?