MCR10144
Workaround CAM Wait Issue for Heavily-loaded Host Systems
Revision 1.3

Charles Anthony

February 6, 2025

1 Problem Description

When Multics is running under the DPS8M simulator on a heavily-loaded host system, it sometimes crashes with a PRDS stack overflow. This issue was observed when running the CI-Kit (continuous integration test kit) many times in succession on a host system where Multics CPUs run slowly due to host starvation.

Ocassionally, Multics crashes with `sys_trouble` "Fault in idle process" or "Fault/interrupt while on prds". This was determined to be a race condition in Multics triggered by the inability of the simulator to fairly allocate host cycles to the different simulated CPUs.

See the detailed analysis in a subsequent section. We observed the following simulator messages leading up to the PRDS overflow fault:

```
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 4 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 3 FAULT: Fault 8(010), sub 0(00), 'Connect'
```

The above output shows various CPUs handing a Connect fault. The second CPU 1 fault, on the third line suggests that the code to clear the Content Accessible Memory (CAM) was timing out. This results in it reissuing connects to the other CPUs. The CAM code that does this is in `cam_cache`. It is entered when one CPU wishes all CPUs to clear their CAMs. The first CPU, after clearing its CAM, spins waiting on the other CPUs (which it issues connects to in order to get them into the `cam_cache` code), to clear their CAMs and signal that they have done so.

The lack of CPUs 2 and 3 messages suggests that they did not get a chance to respond in time.

The source of the connects could only have been `cam_cache` since the history debugger log showed that all of the processors were involved in `cam_cacher` or the `wired_fim` connect fault handler.

Sometimes the system was observed to crash with "sys_trouble: Fault while in masked environment."

The original bug report for this issue is located here: https://multics-trac.swenson.org/ticket/344.

2 Detailed Analysis

Release Engineer Jeffrey Johnson reported CI-Kit run failures on Eric Swenson's ThinkPad that were not observed on other build/test platforms; specifically a Multics Fault in idle process.

Eric's laptop is used as a job runner for the DPS8M projects Continuous Integration (CI) and can be under very heavy load at times; thus it was suspected that the root cause of the issue was a race condition in Multics or in the simulator (or both). (Since the clock synchronization code to solve a race condition had recently been added to the code, race conditions were on everyones mind.)

Many attempts were made on various machines with simulated load conditions to try to reproduce the bug elsewhere, and to find a way to maximize the failure rate to make fix testing easier.

I was able to reproduce the crash moderately reliably on his laptop (1 out of 4 or so runs would crash). To accomplish this, I had to run a simulated load with:

```
stress-ng --cpu 6 --syncload 32 --syncload-msbusy 2000 --syncload-mssleep 500
```

Next, I tried to shorten the test. As presented, the fault occurred during a Multics install process, usually at the start of the `reload` of the `12.8EXEC` or `12.8LDD_STANDARD` tapes. I reasoned that everything after the BCE boot command should have the same execution environment, so I booted a Multics instance, entered BCE, and performed the `reload` command. After multiple tries, I was unable to reproduce the crash, so something nuanced was going on.

I reworked the CI-Kit scripts to isolate the section of the script that booted into ring 1 on a new disk and executed the `reload` commands; the script would crash often. We were able to do a BCE dump and run `analyze_multics` on it. It was determined that the crash condition was an access violation on the PRDS segment of a CPU in the idle process. Examination of the PRDS stack showed that the stack had overflowed. I then added code to the simulator to try to track the sequence of events leading up to the crash. This proved quite problematic – often the added code would cause CI-Kit to not crash. Eventually, I added code to track the depth of the PRDS stack that still crashed CI-Kit. (The code resided in the Append Unit Operand Store Logic. If the operand address segment matches the PRDS segment and the offset matches the offset of `stack_end` pointer the CPU ID and the operand value is printed).

The printed values were recorded in a file, and graphs of the PRDS stack depth of each CPU over time were produced. The graphs showed that a small number of frames would get stacked and freed until the onset of the crash where the stack frames of four of the five CPUs shot upward suddenly; one of those CPUs quickly hitting the end of the segment and causing the "Fault on PRDS" crash.

There had been debate over whether the PRDS stack overflow was a gradual leak or a cascade failure; the graph data supported the cascade failure model. For some reason, Multics was either unable to drop frames, or frames were being created faster than they could be dropped. (The monotonic growth of the stack shown by the graph suggested the first model – unable to drop frames.)

3 Further Analysis

Eventually I was able to get a subset of the history debugger running. It collects events into a circular buffer and saves the buffer to a file when commanded. The code collected the following events:

Instruction execution: Every time a CPU executes an instruction, the CPU ID, PPR.PSR, PPR.IC and operation code are stored in the buffer.

Interrupt: Every time a CPU transfers through an interrupt vector, the CPU ID and interrupt pair address are stored in the buffer.

Interrupt Cell Set: Every time a SCU interrupt cell is set, the cell number is stored in the buffer.

Fault: Every time a CPU transfers through a fault vector, the CPU IS and fault number are stored in the buffer.

The data collection would start when a PRDS stack end pointer offset was observed to exceed 20000o; when an offset exceeded 24000o, the buffer contents would be saved to a file and the simulation terminated. The circular buffer was configured to hold the last 100000 events. (The logic behind the selection of those offset thresholds lie in the observation that enabling the history debugger data collection caused the code to not crash. The offset of 20000o was chosen as a value larger than seen during normal operation and indicated that the overflow cascade had started; the 240008 value was selected as sufficiently large that several cascade cycles would have been captured by the history debugger.)

The code was run and a history debugger file was generated. A portion of the file is shown here:

```
DBG(81170336)> CPU 4 TRACE: 00034:002401 0 002405600000 (TZE 002405)
DBG(774406647)> CPU 0 TRACE: 00042:036622 0 036613601000 (TNZ 036613)
DBG(54052955)> CPU 1 TRACE: 00034:002403 0 000110777000 (LLR 000110)
DBG(774406649)> CPU 0 TRACE: 00042:036613 0 000001175007 (SBA 000001,DL)
DBG(81170338)> CPU 4 TRACE: 00034:002402 0 000110777000 (LLR 000110)
DBG(54052957)> CPU 1 TRACE: 00034:002404 0 002400710000 (TRA 002400)
DBG(774406651)> CPU 0 TRACE: 00042:036614 0 036573604000 (TMI 036573)
DBG(81170340)> CPU 4 TRACE: 00034:002403 0 000110777000 (LLR 000110)
DBG(54052959)> CPU 1 TRACE: 00034:002400 0 400162315120 (CANA PR4|162,N*)
DBG(774406653)> CPU 0 TRACE: 00042:036615 0 000140106400 (CMPC 000140)
DBG(81170342)> CPU 4 TRACE: 00034:002404 0 002400710000 (TRA 002400)
DBG(54052961)> CPU 1 TRACE: 00034:002401 0 002405600000 (TZE 002405)
DBG(81170344)> CPU 4 TRACE: 00034:002400 0 400162315120 (CANA PR4|162,N*)
```

The number in parentheses is the cycle number that the CPU is executing; CPU 0 indicates CPU A, CPU 1 indicates CPU B, and so on. The `TRACE` keyword indicates that the line contains an instruction execution event. Following `TRACE:` is the executed instruction's segment number and offset (`PPR.PSR`, `PPR.IC`), ring number (`PPR.PRR`), the instruction, and in parentheses, the instruction in ALM. (The cycle numbers are elided from the following listings.)

That file was then post-processed to show the lines from the assembler and compiler listings associated with the instruction. The above sample became:

```
CPU 4 TRACE: 00034:002401 0 002405600000 (TZE 002405)
CPU 4 TRACE: 00034:002401 bound_interceptors:wired_fim+0171
```

```
CPU 4 TRACE:   000171 0a   000175 6000 00  358     tze    *+4           if not, exit loop

CPU 0 TRACE: 00042:036622 0 036613601000 (TNZ 036613)
CPU 0 TRACE: 00042:036622 bound_page_control:cam_cache+0132
CPU 0 TRACE:   000132 0a   000123 6010 00  245    tnz    wait          all cells haven't
    cleared

CPU 1 TRACE: 00034:002403 0 000110777000 (LLR 000110)
CPU 1 TRACE: 00034:002403 bound_interceptors:wired_fim+0173
CPU 1 TRACE:   000173 aa   000110 7770 00  360     llr    72

CPU 0 TRACE: 00042:036613 0 000001175007 (SBA 000001,DL)
CPU 0 TRACE: 00042:036613 bound_page_control:cam_cache+0123
CPU 0 TRACE:   000123 aa   000001 1750 07  238 wait:     sba    1,dl          one more
    loop

CPU 4 TRACE: 00034:002402 0 000110777000 (LLR 000110)
CPU 4 TRACE: 00034:002402 bound_interceptors:wired_fim+0172
CPU 4 TRACE:   000172 aa   000110 7770 00  359     llr    72

CPU 1 TRACE: 00034:002404 0 002400710000 (TRA 002400)
CPU 1 TRACE: 00034:002404 bound_interceptors:wired_fim+0174
CPU 1 TRACE:   000174 0a   000170 7100 00  361     tra    *-4           if so, wait more
```

A second line showing the mapping of the instruction address into segment, component, and offset into the component, and a third line reproducing the matching line from the listing file of the component.

By using text processing tools, the overlapping CPU activities can be teased out. For example, to look at CPU A exclusively:

```
$ grep "CPU 0" hdbg0000.list.annotated
CPU 0 TRACE: 00042:036622 0 036613601000 (TNZ 036613)
CPU 0 TRACE: 00042:036622 bound_page_control:cam_cache+0132
CPU 0 TRACE:   000132 0a   000123 6010 00  245    tnz    wait          all cells haven't
    cleared
CPU 0 TRACE: 00042:036613 0 000001175007 (SBA 000001,DL)
CPU 0 TRACE: 00042:036613 bound_page_control:cam_cache+0123
CPU 0 TRACE:   000123 aa   000001 1750 07  238 wait:     sba    1,dl          one more
    loop
CPU 0 TRACE: 00042:036614 0 036573604000 (TMI 036573)
CPU 0 TRACE: 00042:036614 bound_page_control:cam_cache+0124
CPU 0 TRACE:   000124 0a   000103 6040 00  239    tmi    repeat        try entire cycle
    again
CPU 0 TRACE: 00042:036615 0 000140106400 (CMPC 000140)
CPU 0 TRACE: 00042:036615 bound_page_control:cam_cache+0125
CPU 0 TRACE:   000125 aa   000140 1064 00  240    cmpc   (),(pr,rl),fill(0)   check
    entire array clear
CPU 0 TRACE: 00042:036620 0 000000011000 (NOP 000000)
CPU 0 TRACE: 00042:036620 bound_page_control:cam_cache+0130
CPU 0 TRACE:   000130 aa   000000 0110 00  243    nop
CPU 0 TRACE: 00042:036621 0 000000011000 (NOP 000000)
```

```
CPU 0 TRACE: 00042:036621 bound_page_control:cam_cache+0131
CPU 0 TRACE:   000131 aa   000000 0110 00  244    nop
CPU 0 TRACE: 00042:036622 0 036613601000 (TNZ 036613)
CPU 0 TRACE: 00042:036622 bound_page_control:cam_cache+0132
CPU 0 TRACE:   000132 0a   000123 6010 00  245    tnz    wait          all cells haven't
    cleared
```

We see that CPU A is looping in `cam_cache`, waiting for `scs$fast_cam_pending` to become 0.

For CPU B, we see:

```
$ grep "CPU 1" hdbg0000.list.annotated | less
CPU 1 TRACE: 00034:002403 0 000110777000 (LLR 000110)
CPU 1 TRACE: 00034:002403 bound_interceptors:wired_fim+0173
CPU 1 TRACE:   000173 aa   000110 7770 00  360     llr    72
CPU 1 TRACE: 00034:002404 0 002400710000 (TRA 002400)
CPU 1 TRACE: 00034:002404 bound_interceptors:wired_fim+0174
CPU 1 TRACE:   000174 0a   000170 7100 00  361     tra    *-4           if so, wait more
CPU 1 TRACE: 00034:002400 0 400162315120 (CANA PR4|162,N*)
CPU 1 TRACE: 00034:002400 bound_interceptors:wired_fim+0170
CPU 1 TRACE:   000170                      355     inhibit off
    <->-><->-><->-><->-><->-><->-><->-><->
CPU 1 TRACE:   000170 4a  4 00026 3151 20  357     cana   scs$cam_wait  still waiting?
CPU 1 TRACE: 00034:002401 0 002405600000 (TZE 002405)
CPU 1 TRACE: 00034:002401 bound_interceptors:wired_fim+0171
CPU 1 TRACE:   000171 0a   000175 6000 00  358     tze    *+4           if not, exit loop
CPU 1 TRACE: 00034:002402 0 000110777000 (LLR 000110)
CPU 1 TRACE: 00034:002402 bound_interceptors:wired_fim+0172
CPU 1 TRACE:   000172 aa   000110 7770 00  359     llr    72
CPU 1 TRACE: 00034:002403 0 000110777000 (LLR 000110)
CPU 1 TRACE: 00034:002403 bound_interceptors:wired_fim+0173
CPU 1 TRACE:   000173 aa   000110 7770 00  360     llr    72
```

CPU B is in `wired_fim`, looping waiting for `scs$cam_wait` to clear. CPUs C, D, and E are also in `wired_fim` looping on `scs$cam_wait`. Additional filtering of the data shows that for the duration of the captured events, CPU A was only executing `cam_clear`, and CPUs B, C, D and E were in `wired_fim`, responding to connect faults.

Reviewing the source code of `cam_cache` and `wired_fim`, we see that the CAM cache clear is described (`cam_cache.alm`, lines 59-99) as:

```
"      The protocol for multi-processor clearing is as follows:
"
"              This processor obtains the connect lock.
"
"              Under the connect lock, the processor
"                 1. sets scs$cam_pair to the instructions
"                    which do the clearing
"                 2. sets the scs$fast_cam_pending cell
"                    non-zero for all other processors
"                 3. if this is a cam-with-wait call, sets
```

```
"                   the appropriate bit in scs$cam_wait for
"                   all other processors
"                4. sends a connect to all other processors
"                5. XED's the code in scs$cam_pair
"                6. waits for all scs$fast_cam_pending cells
"                   to clear (indicating clearing done by
"                   all other processors).
"                7. releases the connect lock and returns
"             Note - if only one processor is active, most
"                of this is skipped.
"
"             Upon receipt of a connect, all other processors
"                1. if its scs$fast_cam_pending cell is set,
"                   XED the code in scs$cam_pending and
"                   clear it scs$fast_cam_pending cell
"                2. if its bit in scs$cam_wait is set, wait
"                   for that bit to clear (this clearing is
"                   done by the caller of cam_cache on the
"                   originating processor).
"
"
"      There are only two ways a conect fired to another
"      processor can be lost. One is hardware failure, and the
"      other is a processor put into step mode before the connect
"      and taken out of step mode after the connect. There is
"      a hedge against the latter here. If all processors have
"      not responded within an unreasonable amount of time, the
"      connects are re-issued, and the waiting begins anew.
"      This hedge should not be construed as implicitly condoning
"      putting a cpu on a multi-processor in step mode. It may
"      help in truly strange circumstances.
```

Armed with this, we can try to determine what seems to be happening. First: Is `cam_cache` executing the repeat loop, impling that the connects are timing out?

```
$ grep  CPU  0   hdbg0000.list.annotated | grep repeat:
CPU 0 TRACE:   000103                     219  repeat:
CPU 0 TRACE:   000103                     219  repeat:
CPU 0 TRACE:   000103                     219  repeat:
CPU 0 TRACE:   000103                     219  repeat:
CPU 0 TRACE:   000103                     219  repeat:
CPU 0 TRACE:   000103                     219  repeat:
```

In the 100,000 events logged, the CPU A retried the connect loop six times. Next question: Are the other CPUs receiving the connects?

```
$ grep FAULT hdbg0000.list.annotated
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 4 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
```

```
CPU 3 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 3 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 3 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 2 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 1 FAULT: Fault 8(010), sub 0(00), 'Connect'
CPU 2 FAULT: Fault 8(010), sub 0(00), 'Connect'
```

We see that not all CPUs are responding to the connects in a timely manner. We know that all of the connects were issued by the `cam_clear` code, as we have observed that only `cam_clear` and `connect_handler` code was executed during the logging period. So the supposition is that the vagaries of the host scheduler in the heavily loaded host is not allowing the other CPUs enough host CPU cycles to respond to the connect before the timeout loop in `cam_clear` expires and reissues the connects. The `cam_clear` code reissues connects to all CPUs, not just the non-responsive CPUs.

Examination of the connect handler in `wired_fim` shows that a PRDS stack frame is created every time a connect fault is received. The handler acknowledges the connect by clearing the appropriate flag in `scs$fast_cam_pending`, and waits for the cam clear code to clear `scs$cam_wait`, indicating that all CPUs have completed the CAM cache clear. Since some CPUs are not responding in time, `scs$cam_wait` is not getting cleared; instead a new set of connects are issued, and `wired_fim` is reentered on a new stack frame. Without `scs$cam_wait` getting cleared, the connect handler never completes and the process pre-emption never occurs and the stack is never reset. After a sufficient number of timeouts, a PRDS stack overflow occurs.

4 Mechanism That Leads to Crash

To understand how the existing Multics code can be affected by the simulator host's CPU starvation, leading to an the PRDS overflow requires an understanding of the CAM clearing logic.

The CAM clearing logic is as follows. A CPU determines that a clearing of the CAM cache is required. It then sends a connect signal, by way of a `CIOC` instruction to each of the other CPUs, after setting flags that tell the CPUs receiving the connect faults, that a CAM clear operation is required. It then goes into a spin loop (using a loop counter of 1000) waiting for the other CPUs to respond that they have cleared their caches. Once all other CPUs have updated state indicating that they have cleared their CAM, the originating CPU, as well as all the other CPUs can resume executing normally.

The root of the issue is that each simulated Multics CPU runs in a separately scheduled thread on the host system, so if the host is heavily loaded, it is possible that Multics CPUs will not run sufficiently long to get through the CAM clearing logic before the loop counter in the originating CPU runs out. If this loop counter does run out, as it has been seen to do in the scenario that prompted this MCR, it reissues connect signals to all the other CPUs, and the whole process starts again. This may occur many times. Each time it does occur a stack frame is added to the PRDS stack for a given CPU. If this occurs enough times, the PRDS stack will overflow, and the system will crash in the manner descrbed above.

A simple fix that decreases the chance of a slow CPU (or set of slow CPUs) leading to the PRDS

stack overflow is to increase the loop counter from its current value of 1000 to some larger value. In fact, that is what the proposed fix (described later) actually does.

5 Proof of Concept Fix

For the simulator, the SD and PT caches are not normally used. (The cache hardware uses content-addressable memory, allowing cache lookup in a single memory cycle; the simulator has no access to such hardware, so must do a sequential search of the cache memory, which takes about two orders of magnitude longer than just fetching the table entry from memory. Multics is not aware of this; the simulator just reports the entry was not in the cache and the page lookup algorithm is perfectly happy with this.) Because the cache is not used, the CAM cache clearing operation is effectively a NOP, and the Multics requirement to clear the cache to keep the PT and SD tables synchronized across the CPUs is also a no-op.

Eric Swenson created a boot tape that had all calls to the `cam_clear` code commented out; testing of that code showed that the PRDS overflow could not be recreated. Disabling the CAM cache clearing code did introduce some other issues due to the host `ucache` having an implicit dependency on the clearing. Those issues will be addressed in a future MCR that proposes to allow disabling the Multics CAM clearing logic.

6 Proposed Temporary Workaround

With high confidence, I believe that the origin of the PRDS stack overflow is due to the simulators inability to guarantee fair thread scheduling. Since the Multics `cam_clear` algorithm is expecting the exact behavior of the hardware (in terms of timing), and because the `cam_clear` is not needed for Multics to run on the simulator, and not doing the `cam_clear` can significantly improve Multics throughput, adding a production quality implementation of the "just dont call `cam_clear`" proof-of-concept change is the best fix.

However, updating Multics to "just don't call `cam_clear`" is an involved one. Ideally, in order to allow Multics to be run on the original (or future) hardware with CAM support, the change should not simply remove all support for the CAM clearing logic. Instead, a reasonable fix would be to conditionalize this CAM clearing logic on a global variable. Then, a config deck `PARM` card parameter could be added to control the value of the global switch.

Finally, the ability to turn off the `cam_clear` logic should be gated by Multics' being run on the DPS8M simulator. To do that, we'd need the ability to have Multics check to see whether it is running on the simulator and adjust its behavior according. The natural way to do this would be to leverage CPU switches, with the corresponding simulator support.

This work will be the subject of another MCR and proposed at a later date. One of the reasons for deferring this fix is the simulator's current `ucache` implementation, which provides a considerable performance gain, also has an implicit dependency on the CAM cache clearing behavior. This complicates the wholesale removal of the Multics CAM clearing code until the simulator has been updated to account for it.

To see an initial proposal of these changes, please see https://dps8m.gitlab.io/blog/posts/20241204_NOCA/.

For the time being, this MCR proposes to simply increase the loop count (many-fold) in the `cam_wait` logic. Currently, the logic times out (and forces another connect sequence, as described above), when a loop iteration counter exeeds 1000. That is, the code iterates up to 1000 times waiting on acknowledgement from the other CPUs that they have cleared their CAMs. Experimentation has shown that increasing this timeout to 100 times the original number, or 100000 allows the system to run normally (perhaps with a bit of a delay in exceptionally heavily-loaded host CPU cases) even on the slowest of host hardware. For good measure, we propose to double that loop counter, to 200000.

In summary, raising the timeout limit from 1000 to 200000 gives much more time for all CPUs to respond before any re-sending of connects, and thereby reduce likelihood of a flood of connects causing the PRDS stack overrun.

Experimentation shows, that with the increased counter, we no longer see the PRDS overflow crashes. This will give us time to prepare a proper fix, as described above.

7 Proposed Fix

The proposed fix simply updates a single source file, `cam_cache.alm` and there, a single value – the loop counter.

```
cpa [lpn cam_cache.alm] ==

A236              lda      1000,dl           bail-out of loop limit
Changed by B to:
B236              lda      200000,dl         bail-out of loop limit

Comparison finished: 1 difference, 2 lines.
```

8 Motivation for Workaround Rather Than Full Solution

The reason we are proposing to move forward with the above workaround is because the PRDS stack overflow crash can occur on any platform/host system where multiprocessing Multics is being used and the host system is not dedicated to running Multics or otherwise specially configured. The most common environment where the crash can occur is when Multics is being run under the simulator on a virtual host, such as provided by many virtual host providers, including Linode, AWS, and Google Cloud Platform.

The bug has been seen a few times outside the CI-Kit test environment. It may not have been seen much by casual Multics users, because they typically run the Multics QuickStart, which doesn't configure Multics with more than 1 CPU. The bug only manifests itself when there are more than one Multics CPU configured.

There are certain types of hardware that are much more likely to cause the problem to manifest. These include multi-tenant VPS systems (where the user has no control over the load of process scheduling of the host system because it is shared with hundreds or possibly thousands of other users), hosts with CPUs that run at different speeds (increasingly common), hosts that use aggressive dynamic frequency scaling or power saving features (laptops, embedded boards), etc.

The systems least likely to exhibit the problem are server hardware running at fixed frequencies with many (uniform) high performance processing cores dedicated to running the simulator. This would include high-end desktop systems and dedicated server equipment.

It's also worth noting that desktop computers (and dedicated server hardware) are becoming much less common in the field now, while laptops, small embedded computer systems, cell phones, tablets, and mobile devices are becoming much more common.

The problem is mostly independent of the overall speed of the host (as in throughput or instruction execution speed) but triggered by scheduling latency. Even in on a system in a configuration where the problem manifesting is exceedingly unlikely, it is not 100% avoidable without very specific special configuration of the system, which would mean dedicating that computer system to running only the simulator.

Indeed, the problem can happen on the slowest of slow systems (capable of running Multics at less than one tenth the speed it ran on the real hardware) or on hardware which can execute simulated Multics 30-50 times faster than the real hardware ever could.

For the reasons above, and because a fix that involves disabling CAM clearing in Multics, and updating the DPS8M simulator to let Multics know that it should do so based on communicating to Multics that it is running on the simulator will take time to design, coordinate, and develop, we believe that the interim fix is appropriate to install into the next Multics release now.

9 Suggested Future Multics Fix

The CAM clear code reissues conencts to all CPUs, not just the non-responsive CPUs. In the classical environment, with predictable CPU behavior, this sufficed as the probability of multiple misssed connects was vanishingly small and the PRDS stack would not overflow.

I would argue that reissuing connects to only the non-responsive CPUs is better code and appears to be faily easy to accomplish. I propose that we explore this as a possible future fix.

10 Documentation

No documentation changes are needed since this is an internal hardcore change.

11 Test

The change will be tested by running with the updated hardcore on a variety of simulator-based host configurations. In fact, this fix has already proven itself during initial testing.

12 Version History

2025-02-03	1.0	Initial version of the MCR.
2025-02-03	1.1	Incorporate suggestions from a couple early reviewers.
2025-02-04	1.2	Incorporate more review comments.
2025-02-06	1.3	Added section on motivation for workaround as opposed to full solution.