Tracking the Virtual World

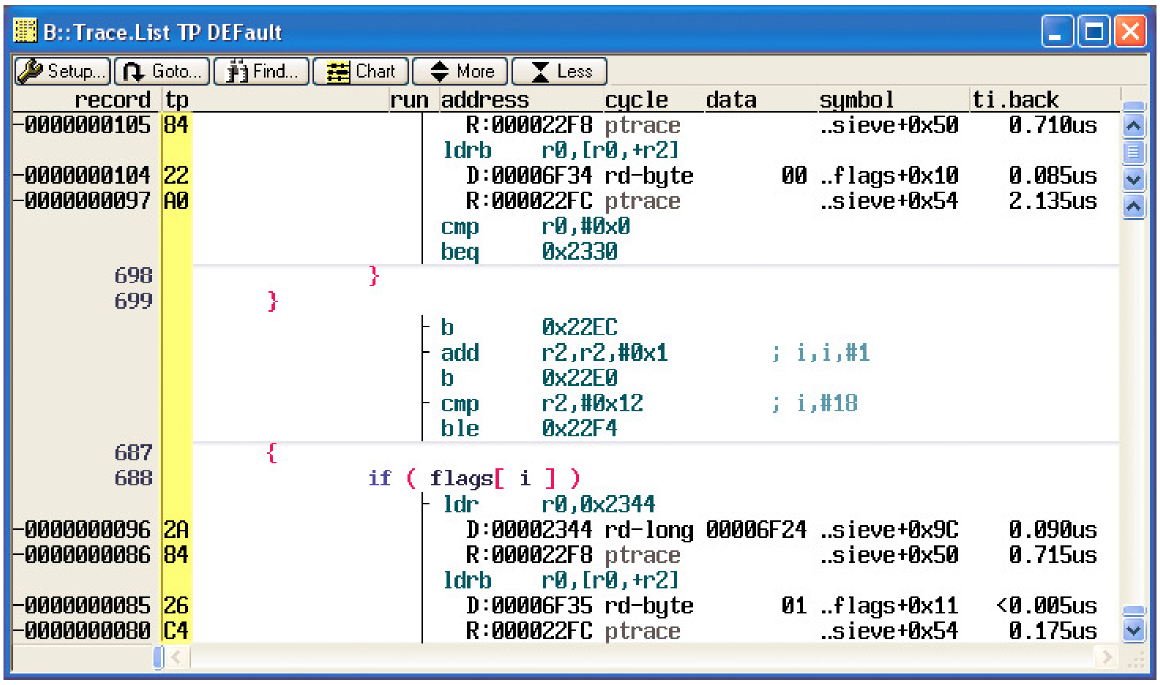
Synopsys:

For many years the JTAG interface has been used for ARM-based SoC debugging. With this JTAG style debugging, the developer has been granted the ability to debug software at the high-level. Breakpoints could be set, registers and memory could be accessed, and the processor could be started and stopped at will. Thus the static state of the system was made visible to the engineer. It seemed the only thing missing was visibility to a history of executed cycles. With the introduction of the Embedded Trace Macrocell, commonly known as ETM, this requirement had been fulfilled. Offering a program flow trace by way of a compressed trace port, the developer could now see a history of executed cycles. Ah, was this freedom from static mode debugging?

ETM enabled the capture of virtual addresses where the processor had executed, enabling the program flow trace. This process works great for systems using a statically mapped MMU(memory management unit), where each virtual address uniquely maps to a physical address. But with many of today's popular operating systems, such as Linux or Windows CE, a dynamic MMU is used. In this case the mapping of the virtual address depends on the active process, requiring additional smarts to be built into the debugger.

This article outlines the method which enables the Lauterbach debugger to analyze a program trace, despite these process-dependent virtual addresses. But before tackling the topic at hand, let's take a deeper look at this ETM technology.

The ETM is a core that sits next to the CPU. Its main purpose is to monitor the cycles executed by the CPU and to present, via a dedicated trace port, the program flow information to the used debugger. Thus the ETM only outputs trace information relating to discontinuities in program flow, such as branches or interrupts, which directs the CPU away from its sequential execution path. With this information, and the knowledge of what code existed in memory at the time of execution, a complete trace history can be reconstructed. To display the complete instruction flow, the debugger reads the corresponding instructions from the target memory in order to fill in the gaps between the execution discontinuities that have been recorded (See Fig. 1).



```
record      tp          run address          cycle     data      symbol         ti.back
-0000000105 84                 R:000022F8 ptrace              ..sieve+0x50    0.710us
                            ldrb     r0,[r0,+r2]
-0000000104 22                 D:00006F34 rd-byte        00  ..flags+0x10    0.085us
-0000000097 A0                 R:000022FC ptrace              ..sieve+0x54    2.135us
                            cmp      r0,#0x0
                            beq      0x2330
        698                }
        699        }
                            b        0x22EC
                            add      r2,r2,#0x1         ; i,i,#1
                            b        0x22E0
                            cmp      r2,#0x12           ; i,#18
                            ble      0x22F4
        687        {
        688                if ( flags[ i ] )
                            ldr      r0,0x2344
-0000000096 2A                 D:00002344 rd-long 00006F24 ..sieve+0x9C    0.090us
-0000000086 84                 R:000022F8 ptrace              ..sieve+0x50    0.715us
                            ldrb     r0,[r0,+r2]
-0000000085 26                 D:00006F35 rd-byte        01  ..flags+0x11   <0.005us
-0000000080 C4                 R:000022FC ptrace              ..sieve+0x54    0.175us
```

Fig. 1: The yellow area to the left of the screenshot displays the raw trace data as it was recorded by the debugger's trace module. The actual program flow is displayed on the right, as it has been reconstructed by the debugger using the code from the target memory.

In addition to the program flow, the ETM also has the ability to supply information relating to data accesses. However, this can cause problems with the program flow information do to the large amount of information that is required (i.e. a long access would require a 32-bit address and 32-bit data). Too many data accesses within a short period of time can cause an overloading of the ETM trace port, resulting in gaps in the recorded program flow.

Since the ETM is assigned directly to the CPU, it always delivers virtual addresses. Which is appropriate since the code is executing in this address space, as well as all of the variables, pointers, stack, etc… In this case we do not care which physical address a given process runs. As the trace and memory are presented in the same way the processor sees them, which is most desirable.

But what happens if MMU is reprogrammed at run time causing the processor's view to change during execution? In this scenario, some virtual addresses can become invalid, others become valid, and yet others could be mapped to different code than when execution began. Since the ETM only displays the virtual branch destination addresses, and not the code that has actually been executed, it now becomes difficult to reconstruct the program flow.

Operating systems utilizing a dynamically managed MMU are typical examples of this scenario. These include, Windows CE, Linux, QNX, LynxOS and Symbian OS. With these embedded operating systems, each process receives a separate virtual address space. Physically, these obviously reside in different memory areas. But virtually, every process uses the same address space, which typically begins with address zero. Therefore the processor only works in the address space of the currently active process. During a process change, the processor's MMU is reprogrammed such that it then sees and handles a completely different process (code and data) within the same virtual address space.

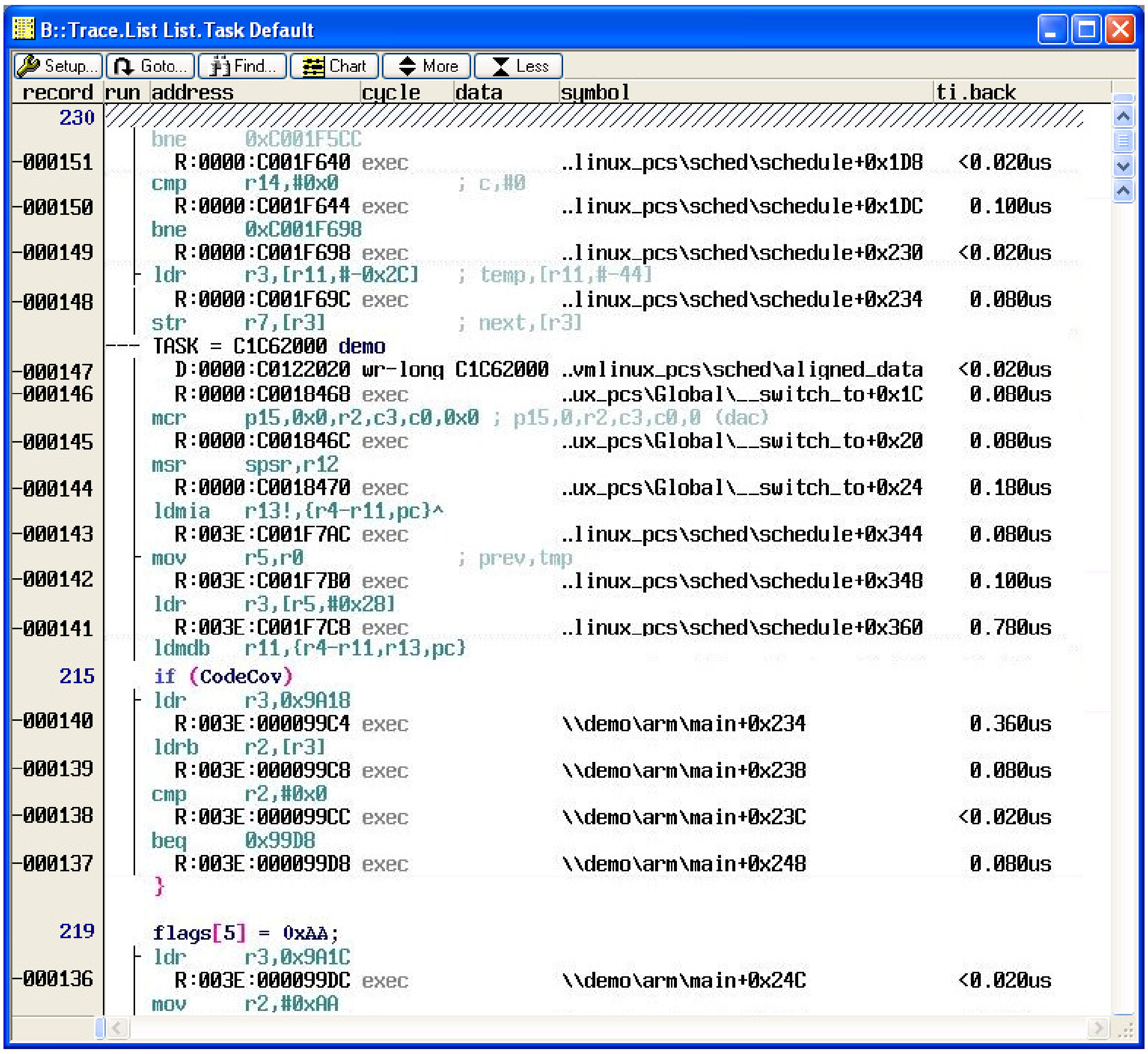
So what happens if a trace recording includes one or more process switches?

Let's assume the system first runs in process "A." In this case the ETM records the program execution using the virtual addresses of process "A." Now assume a process switch occurs causing the MMU to be reprogrammed so that process "B" is now visible to the CPU, and process "B" is now being executed. The ETM now records the program execution with the virtual addresses of process "B." If we now stop the processor along with the trace recording and attempt to reconstruct the program flow, two problems arise:

Problem 1: Since the two processes use the same virtual address space, both processes will appear in the trace buffer to be executing from the same address space. Although, process "A" and process "B" are really two different processes, each with its own unique code. It is now the task of the debugger to distinguish between which parts of the trace belong to which process.

Problem 2: As mentioned previously, the ETM only records branch destination addresses, and not the code that was actually executed. Thus the debugger depends on its ability to access the executed program from target memory in order to reconstruct the program flow. However, at the time of the reconstruction, the processor is in the context of process "B" and cannot directly access the code in process "A." Therefore reconstruction of process "A" would not be possible. Thus the debugger must be allowed to access the code from process "A," even though the current context is in process "B."

Let us first address problem 1. To correctly correlate the collected branch trace history back to the correct piece of executed code, it is important the debugger knows exactly where in the trace history the process switches had occurred. This requires the trace system to include a mechanism by which the process switches can be collected during real-time execution. Typically, each operating system has a variable that shows which process is currently active at a given time. This process variable is written with a new value (relating to the new process) whenever a process switch takes place. By recording the write accesses to this process variable, the debugger has the ability to decipher which sections of the trace history correlate to which process. From the point of one of these process variable writes, until the next time it is written, everything runs within the context of the described process (See Fig. 2).



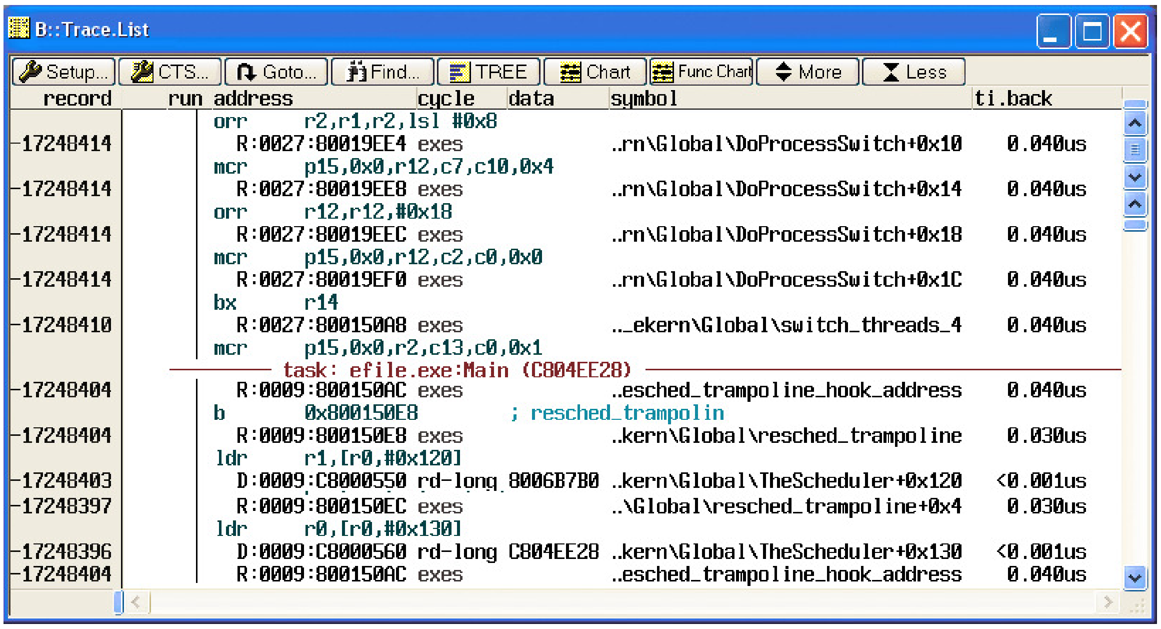
```
record run address                cycle   data       symbol                            ti.back
   230   ///////////////////////////////////////////////////////////////////////////////////
          | bne      0xC001F5CC
-000151   |   R:0000:C001F640 exec              ..linux_pcs\sched\schedule+0x1D8   <0.020us
          | cmp      r14,#0x0          ; c,#0
-000150   |   R:0000:C001F644 exec              ..linux_pcs\sched\schedule+0x1DC    0.100us
          | bne      0xC001F698
-000149   |   R:0000:C001F698 exec              ..linux_pcs\sched\schedule+0x230   <0.020us
          |- ldr     r3,[r11,#-0x2C]   ; temp,[r11,#-44]
-000148   |   R:0000:C001F69C exec              ..linux_pcs\sched\schedule+0x234    0.080us
          | str      r7,[r3]           ; next,[r3]
        --- TASK = C1C62000 demo
-000147   |   D:0000:C0122020 wr-long C1C62000 ..vmlinux_pcs\sched\aligned_data   <0.020us
-000146   |   R:0000:C0018468 exec              ..ux_pcs\Global\__switch_to+0x1C    0.080us
          | mcr      p15,0x0,r2,c3,c0,0x0 ; p15,0,r2,c3,c0,0 (dac)
-000145   |   R:0000:C001846C exec              ..ux_pcs\Global\__switch_to+0x20    0.080us
          | msr      spsr,r12
-000144   |   R:0000:C0018470 exec              ..ux_pcs\Global\__switch_to+0x24    0.180us
          | ldmia    r13!,{r4-r11,pc}^
-000143   |   R:003E:C001F7AC exec              ..linux_pcs\sched\schedule+0x344    0.080us
          |- mov     r5,r0             ; prev,tmp
-000142   |   R:003E:C001F7B0 exec              ..linux_pcs\sched\schedule+0x348    0.100us
          | ldr      r3,[r5,#0x28]
-000141   |   R:003E:C001F7C8 exec              ..linux_pcs\sched\schedule+0x360    0.780us
          | ldmdb    r11,{r4-r11,r13,pc}
   215      if (CodeCov)
          |- ldr     r3,0x9A18
-000140   |   R:003E:000099C4 exec              \\demo\arm\main+0x234               0.360us
          | ldrb     r2,[r3]
-000139   |   R:003E:000099C8 exec              \\demo\arm\main+0x238               0.080us
          | cmp      r2,#0x0
-000138   |   R:003E:000099CC exec              \\demo\arm\main+0x23C              <0.020us
          | beq      0x99D8
-000137   |   R:003E:000099D8 exec              \\demo\arm\main+0x248               0.080us
            }

   219      flags[5] = 0xAA;
          |- ldr     r3,0x9A1C
-000136   |   R:003E:000099DC exec              \\demo\arm\main+0x24C              <0.020us
          | mov      r2,#0xAA
```

Fig. 2: This screenshot shows a trace recording in which a process switch has occurred.

So in order to gain visibility to these context switches, the ETM must be programmed in such a way that the write accesses to these process variables are recorded in addition to the normal program flow information. It must also be possible to filter out the other, non-process related, data accesses. As we discussed earlier, the collection of too many data accesses can exceed the transmission capacity of the ETM port.

This method allows the debugger to assign the individual trace entries to particular processes. However, there are a couple of disadvantages. Firstly, in order to assign a segment of trace to a known process, the process variable must already have been written/known. So the segment of trace information from the start of the trace collection until the first write to the process variable cannot be assigned to a known process. As there is no way of telling which process was executing prior to the first recorded process switch. Secondly, there must be an available, correctly programmed, trace filter that can be used for the sole purpose of collecting the accesses to the process variable.

With the introduction of ETM version 1.2, hooks were built into the trace system, enabling yet another method by which the process switches can be tracked. ETM v1.2 includes an additional register, “context ID,” which contains the process ID. It is the responsibility of the operating system to update this “context ID” register whenever there is a process switch. The contents of this register can be collected along with each target branch address, as part of the program flow trace information (See Fig. 3). Given this additional information, now the debugger knows the corresponding process ID for each trace entry, without having to first search for an access to the process variable. Always having knowledge of the process ID, even the beginning of the trace capture will be assigned to its respective process.



```
    record     run address               cycle    data      symbol                               ti.back
                   orr      r2,r1,r2,lsl #0x8
-17248414            R:0027:80019EE4 exes               ..rn\Global\DoProcessSwitch+0x10       0.040us
                   mcr      p15,0x0,r12,c7,c10,0x4
-17248414            R:0027:80019EE8 exes               ..rn\Global\DoProcessSwitch+0x14       0.040us
                   orr      r12,r12,#0x18
-17248414            R:0027:80019EEC exes               ..rn\Global\DoProcessSwitch+0x18       0.040us
                   mcr      p15,0x0,r12,c2,c0,0x0
-17248414            R:0027:80019EF0 exes               ..rn\Global\DoProcessSwitch+0x1C       0.040us
                   bx       r14
-17248410            R:0027:800150A8 exes               ..._ekern\Global\switch_threads_4      0.040us
                   mcr      p15,0x0,r2,c13,c0,0x1
              ------------ task: efile.exe:Main (C804EE28) ------------------------------------
-17248404            R:0009:800150AC exes               ..esched_trampoline_hook_address       0.040us
                   b        0x800150E8          ; resched_trampolin
-17248404            R:0009:800150E8 exes               ..kern\Global\resched_trampoline       0.030us
                   ldr      r1,[r0,#0x120]
-17248403            D:0009:C8000550 rd-long 8006B7B0 ..kern\Global\TheScheduler+0x120     <0.001us
-17248397            R:0009:800150EC exes               ..\Global\resched_trampoline+0x4       0.030us
                   ldr      r0,[r0,#0x130]
-17248396            D:0009:C8000560 rd-long C804EE28 ..kern\Global\TheScheduler+0x130     <0.001us
-17248404            R:0009:800150AC exes               ..esched_trampoline_hook_address       0.040us
```

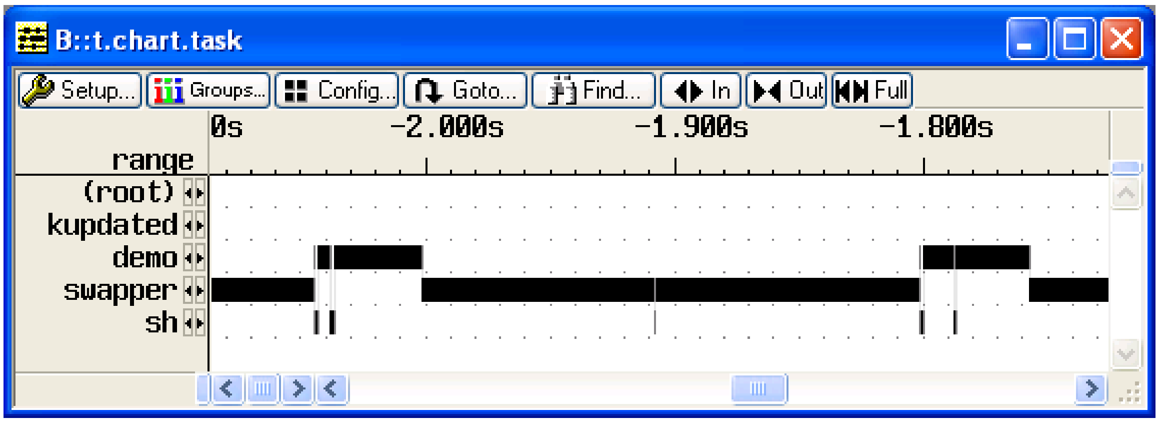
Fig. 3: A process ID is displayed in addition to the branch destination address.

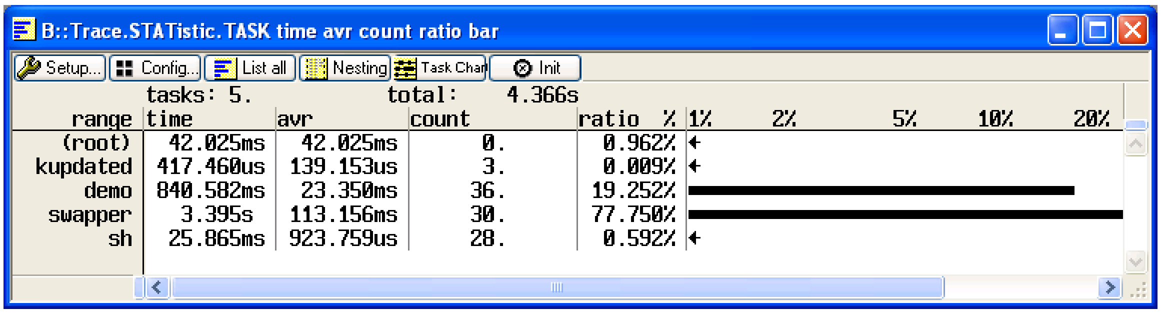
Needless to say, either method still requires the debugger to be informed how to analyze these entries. The Lauterbach debugger is informed of this additional trace information by way of an OS awareness declaration file. This is an operating system specific file that describes the unique characteristics of the used embedded operating system.

Now that we can reliably determine the virtual addresses and corresponding processes of the collected trace, we now have everything necessary to recreate the complete execution history of the processor. Since the ETM only captures the branch destination addresses, and not the entire program flow. The source code can be read from the target memory and used to fill in the gaps in the program flow, that occur between the collected branch messages.

Reading the code of the current process is no problem, since the processor and MMU will already be configured to allow access. However, to read the code of another process, the debugger must somehow bypass the current MMU settings, as the MMU will not allow access to the virtual addresses of an inactive process.

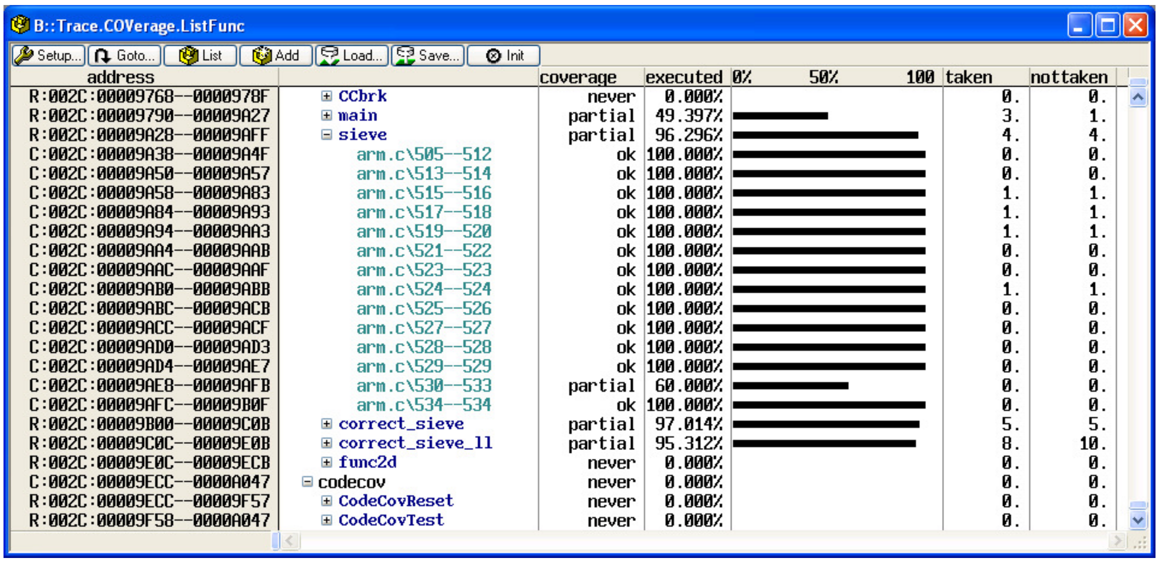
Hidden in the operating system is information relating to the used virtual to physical translation. Reprogramming of this MMU translation is managed by the operating system during each process switch. By scanning the operating system for this information, the debugger can create a conversion table which includes the mapping between the used virtual to physical addresses for all processes. Thus the debugger now knows the mapping between all virtual to physical addresses for all processes. Now, rather than having to change processes to access the required code from an inactive process, this table can be used to determine the related physical address. And by deactivating the MMU, the debugger can directly access the code of an inactive process by way of its equivalent physical address.



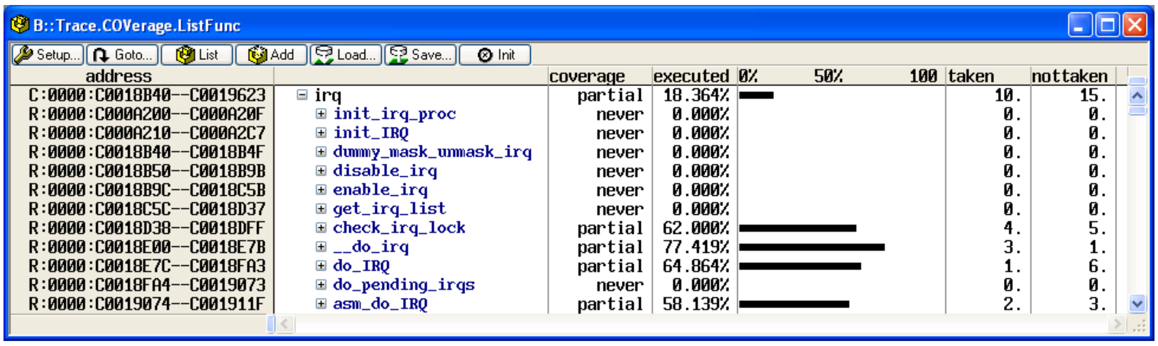


```
          tasks: 5.           total:    4.366s
   range time       avr        count          ratio  % 1%      2%        5%     10%      20%
  (root)  42.025ms   42.025ms        0.         0.962%
kupdated 417.460us  139.153us        3.         0.009%
    demo 840.582ms   23.350ms       36.        19.252%
 swapper    3.395s  113.156ms       30.        77.750%
      sh  25.865ms  923.759us       28.         0.592%
```

Fig. 4: Graphical and statistical analysis of the process switches recorded in the trace.



```
address                        coverage  executed 0%    50%    100 taken  nottaken
R:002C:00009768--0000978F  ⊞ CCbrk              never    0.000%                0.      0.
R:002C:00009790--00009A27  ⊞ main               partial 49.397%                3.      1.
R:002C:00009A28--00009AFF  ⊟ sieve              partial 96.296%                4.      4.
C:002C:00009A38--00009A4F      arm.c\505--512   ok     100.000%                0.      0.
C:002C:00009A50--00009A57      arm.c\513--514   ok     100.000%                0.      0.
C:002C:00009A58--00009A83      arm.c\515--516   ok     100.000%                1.      1.
C:002C:00009A84--00009A93      arm.c\517--518   ok     100.000%                1.      1.
C:002C:00009A94--00009AA3      arm.c\519--520   ok     100.000%                1.      1.
C:002C:00009AA4--00009AAB      arm.c\521--522   ok     100.000%                0.      0.
C:002C:00009AAC--00009AAF      arm.c\523--523   ok     100.000%                0.      0.
C:002C:00009AB0--00009ABB      arm.c\524--524   ok     100.000%                1.      1.
C:002C:00009ABC--00009ACB      arm.c\525--526   ok     100.000%                0.      0.
C:002C:00009ACC--00009ACF      arm.c\527--527   ok     100.000%                0.      0.
C:002C:00009AD0--00009AD3      arm.c\528--528   ok     100.000%                0.      0.
C:002C:00009AD4--00009AE7      arm.c\529--529   ok     100.000%                0.      0.
C:002C:00009AE8--00009AFB      arm.c\530--533   partial 60.000%                0.      0.
C:002C:00009AFC--00009B0F      arm.c\534--534   ok     100.000%                0.      0.
R:002C:00009B00--00009C0B  ⊞ correct_sieve      partial 97.014%                5.      5.
R:002C:00009C0C--00009E0B  ⊞ correct_sieve_ll   partial 95.312%                8.     10.
R:002C:00009E0C--00009ECB  ⊞ func2d             never    0.000%                0.      0.
C:002C:00009ECC--0000A047  ⊟ codecov            never    0.000%                0.      0.
R:002C:00009ECC--00009F57  ⊞ CodeCovReset       never    0.000%                0.      0.
R:002C:00009F58--0000A047  ⊞ CodeCovTest        never    0.000%                0.      0.
```



```
address                        coverage  executed 0%    50%    100 taken  nottaken
C:0000:C0018B40--C0019623  ⊟ irq                   partial 18.364%            10.     15.
R:0000:C000A200--C000A20F    ⊞ init_irq_proc       never    0.000%             0.      0.
R:0000:C000A210--C000A2C7    ⊞ init_IRQ            never    0.000%             0.      0.
R:0000:C0018B40--C0018B4F    ⊞ dummy_mask_unmask_irq never  0.000%             0.      0.
R:0000:C0018B50--C0018B9B    ⊞ disable_irq         never    0.000%             0.      0.
R:0000:C0018B9C--C0018C5B    ⊞ enable_irq          never    0.000%             0.      0.
R:0000:C0018C5C--C0018D37    ⊞ get_irq_list        never    0.000%             0.      0.
R:0000:C0018D38--C0018DFF    ⊞ check_irq_lock      partial 62.000%             4.      5.
R:0000:C0018E00--C0018E7B    ⊞ __do_irq            partial 77.419%             3.      1.
R:0000:C0018E7C--C0018FA3    ⊞ do_IRQ              partial 64.864%             1.      6.
R:0000:C0018FA4--C0019073    ⊞ do_pending_irqs     never    0.000%             0.      0.
R:0000:C0019074--C001911F    ⊞ asm_do_IRQ          partial 58.139%             2.      3.
```

Fig. 5: Code coverage analysis using processes and kernels (segments) based on the trace recording.

Now that the debugger has access to all of the code, including active and inactive processes, the entire program execution can be reconstructed, even through several process switches. With this information, Lauterbach debugging tools can offer the ability to analyze process runtimes (See Fig. 4), cache usage or code coverage (Fig. 5). And with Lauterbach's latest generation of debugging tools, a 2 gigabyte trace memory is available, enabling analysis of quite long run times. For more information, please contact your local Lauterbach representative.

Rudolf Dienstbeck, RTOS Integrations, March 2006