M. Botlo
M. Jagielski
A. Romero

VxWorks v5.1 Benchmark Tests

**Superconducting Super Collider
Laboratory**

APPROVED FOR RELEASE OR
PUBLICATION - O.R. PATENT GROUP
BY...*et*.....DATE: *4/3/95*

Disclaimer Notice

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government or any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Superconducting Super Collider Laboratory is an equal opportunity employer.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

VxWorks v5.1 Benchmark Tests

M. Botlo, M. Jagielski, and A. Romero

Superconducting Super Collider Laboratory*
2550 Beckleymeade Ave.
Dallas, TX 75237

May 1993

*Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC35-89ER40486.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED *mw*

MASTER

VxWorks v5.1 Benchmark Tests

M. Botlo, M. Jagielski, and A. Romero

Abstract

We measure the performance of the VxWorks Real-Time Operating System on various VME single-board computers and compare VxWorks (v5.0.2) and VxWorks (v5.1).

1.0 INTRODUCTION

This paper takes a look at the performance of VxWorks^{1,2} (v5.0.2), released in July 1991, and VxWorks (v5.1), released in March 1993. VxWorks is a high-performance, real-time operating system. Some of the features provided by this real-time kernel are multitasking with preemptive priority scheduling, intertask synchronization, and interrupt handling. The speed at which these occur is always a concern to the data acquisition (DAQ) developer. Since today's DAQ systems are moving to VME, single-board computers residing in VME were examined in this study. Most VME single-board computers support three types of external interrupts: VME IRQ, Location Monitors, and Mail Box Interrupts. The latency from when these interrupts were generated until they were serviced is examined, along with software interrupt latency, task dispatching, and context switches.

Following are definitions of the four different categories of latencies examined:

External Interrupt Latency. The delay to activate the first instruction in the interrupt handler following the external interrupt signal. Three types of external interrupts were used: VME IRQ, Location Monitors, and Mailbox Interrupts.

Software Interrupt Latency. The delay to activate the first instruction in the signal handler following the software interrupt signal [kill()].

Task Dispatching. The time to pass control from the handler service routine to the user-mode code of the task. Two types of task dispatching exist: from external interrupt service requests and from software interrupt signal handler routines.

Context Switch. The time to switch the CPU from one task to another. Context switches examined were the result of priority change, binary semaphore give/take, and pause.

2.0 VxWORKS KERNEL CONFIGURATION

In each case the kernel only consisted of the following subsystems as defined in configAll.h:

```
#define INCLUDE_NETWORK          /* network subsystem code */
#define INCLUDE_NET_INIT        /* network subsystem initialization */
#define INCLUDE_NET_SYM_TBL     /* load symbol table from network */
#define INCLUDE_SHELL           /* interactive c-expression interpreter */
#define INCLUDE_STARTUP_SCRIPT  /* execute start-up script */
#define INCLUDE_STAT_SYM_TBL    /* create user-readable error status */
#define INCLUDE_STDIO           /* standard I/O */
#define INCLUDE_SYM_TBL         /* symbol table package */
#define INCLUDE_NFS             /* nfs package */
#define INCLUDE_SIGNALS (5.1 Only) /* software signal library */
#define INCLUDE_LOADER (5.1 Only) /* object module loading */
```

During the context switch tests, all kernel tasks (tNetTask, tPortmapd, and tShell) were suspended with the exception of the tExcTask task. It should be noted that a taskDelay() needs to be called before actually starting the tests to allow the TTY driver to empty any remaining characters from the shell to the screen. Otherwise, periodic bumps will appear in the timing results as the CPU handles these interrupts from the serial port. The source code along with test instructions can be found at the following ftp site: slug.ssc.gov (134.3.33.40) in /pub/vwBench5.1.

3.0 SINGLE-BOARD COMPUTERS

The following VME single-board computers (SBC) were used in the evaluation:

Motorola MVME167B³ is a MC68040-based board with a 25-MHz clock.

Motorola MVME162⁴ is a MC68040-based board with a 25-MHz clock.

Motorola MVM147S/A⁵ is a MC68030-based board with a 32-MHz clock.

General Micro Systems GMS V37⁶ is a MC68030-based board with a 16-MHz clock.

4.0 MEASUREMENT METHODOLOGY AND RESULTS

A VMETRO VBT-321B⁷ VME bus tracer was used to measure the latency times. This bus tracer triggers on the falling edge of the VME DTACK* signal and has a 25-MHz clock, resulting in a 40-ns resolution. To measure the time between action A and action B the test programs wrote to a Motorola MVME224A-2⁸ external memory board. This would cause VME activity, which is then clocked by the VME bus tracer. In the following discussions, the reader should assume the external memory board has a base address of 20000000 (hexidecimal). For all the results presented in this section, the root mean square (rms) was less than 0.5, and MVME16X represents both the MVME167 and the MVME162. It should be noted that these tests were conducted under optimal conditions with a very small test program and with caching enabled. With caching disabled, the context switch times for the MVME16X were 6 times slower and twice as slow for external interrupt latency.

4.1 External Interrupt Latency

Two single-board computers were used to measure the external interrupt latency. The first SBC was the interrupter, and the second was the handler. The first SBC used the following "C" code routine to generate a VME IRQ:

```
        IRQgenerate(IRQ_level, IRQ_vector)
        int IRQ_level, IRQ_vector;
        {
t1 ->    *(int*)(0x20000000) = 1;      /* START TIMING INTERRUPT LATENCY HERE */
        sysBusIntGen(IRQ_level, IRQ_vector);
        }
```

The second SBC would hook an interrupt service routine (ISR) to the appropriate interrupt vector, then wait in a forever loop for the interrupt to occur. The ISR "C" code would simply be a write to the external memory board to end the timing.

```
        IRQisr()
        {
t2 ->    *(int*)(0x20000004) = 1;      /* END TIMING INTERRUPT LATENCY HERE */
        }
```

Hence the time t2-t1 would represent the interrupt latency. Figure 1 shows the interrupt latency times for each SBC for both v5.0.2 and v5.1 when VME IRQs were used. Location monitor and mail box interrupts work in a similar manner. Figures 2 and 3 show results for location monitor and mail box interrupt latency, respectively.

Figure 1. VME IRQ Interrupt Latency.

Figure 2. Location Monitor Interrupt Latency.

TIP-04482

Figure 3. Mail Box Interrupt Latency.

4.2 Software Interrupt Latency

The software interrupt latency test determines the time it takes to enter the signal handler of task B after task A issues a kill() call. The signal handler and task B "C" code looks as follows:

```

SigHandler()
{
t2 ->  *(int*)(0x20000004) = 1;    /* END TIMING INTERRUPT LATENCY HERE */
}

taskB()
{
  struct sigvec vec;

  vec.sv_handler = SigHandler;
  vec.sv_mask = 0;
  vec.sv_flags = 0;
  sigvec(SIGUSR1, &vec, NULL);
  pause();
}

```

Then task A simply executes the following code:

```

taskA()
{
t1 ->  *(int*)(0x20000000) = 1;    /* START TIMING INTERRUPT LATENCY HERE */
      kill(taskID_B, SIGUSR1)
}

```

Again $t_2 - t_1$ would represent the software interrupt latency. Figure 4 shows the software interrupt latency for each SBC.

Figure 4. Software Interrupt Latency.

4.3 Task Dispatching Latency

In Sections 4.1 and 4.2, external and software interrupts were examined by looking at the latency from the time an interrupt is generated until an interrupt handler routine services the interrupt. This section examines task dispatching, which is the time it takes to get from the handler routine back to executing a task.

For example, one could rewrite the ISR in Section 4.1 as follows:

```

IRQisr()
{
    globalFlag = 1;
t1 ->    *(int *) (0x20000000) = 1;    /* START TIMING DISPATCH LATENCY HERE */
}

```

Then the task running on this same SBC could have "C" code as follows:

```

task()
{
    globalFlag = 0;
    /* Hook interrupt vector here */
    do {
        if (globalFlag)
t2 ->    *(int *) (0x20000004) = 1;    /* END TIMING DISPATCH LATENCY */
        } while (1);
}

```

Thus $t_2 - t_1$ would give the dispatch time from an external ISR. Figure 5 shows ISR task dispatching latency for each SBC.

Figure 5. ISR Task Dispatching Latency.

Likewise task B in Section 4.2 could be modified to add a write to VME after the pause() call. This would measure the task dispatching from a signal handler. Figure 6 shows results for task dispatching from a signal handler.

Figure 6. Signal Handler Task Dispatching Latency.

4.4 Context Switch Latency

This section examines the context switch latency—that is, how fast can the kernel stop executing one task and start executing another. Several situations can cause the kernel to swap execution of tasks. Three methods will be examined here. First a test was done with three tasks executing in a loop, with each task simply lowering its own priority and causing a context switch to the next task. Figure 7 shows the results of this test on each SBC.

Figure 7. Priority Change Context Switch.

The second context switch test used binary semaphores. For example, task A writes to an address on the external memory board and then executes a `semTake()`. Task B writes to the external memory and executes a `semGive()`. Since task B runs at a lower priority, a context switch to task A takes place. This sequence executed in a loop in both tasks, causing them to ping back and forth. The code fragments for each task look like this:

```

taskA()
{
i = 0;
do
{
t1 ->  *(int *) (0x20000000) = 0x1A;      /* START semTake() TIMING */
t4 ->  semTake(sem, WAIT_FOREVER);
      *(int *) (0x20000000) = 0x4A;      /* END semGive() TIMING */
      } while (++i < 25);
}

taskB()
{
i = 0;

```

```

t3 ->      do      {
                *(int *) (0x20000000) = 0x3B;      /* START semGive() TIMING */
                semGive(sem);
t2 ->      *(int *) (0x20000000) = 0x2B;      /* END semTake() TIMING */
                } while (++i < 25);
        }

```

Examining the above we see that $t_2 - t_1$ measures the time for a context switch to take place when caused by a `semTake()`, and $t_4 - t_3$ measures the context switch time for a `semGive()`. Figure 8 shows the context switch times for taking a binary semaphore, and Figure 9 shows the times for giving a binary semaphore.

The third context switch measured was caused by the `pause()` call. Looking back at Section 4.2, if the `pause()` in task B and the `kill()` in task A are put in loops like the semaphore example above, one can measure the context switch time caused by `pause()` call. Figure 10 shows results for the pause context switch for each SBC.

Figure 8. Semaphore Take Context Switch.

Figure 9. Semaphore Give Context Switch.

Figure 10. Pause Context Switch.

5.0 CONCLUSION

As expected, the MVME16X SBCs performed better than the other two SBCs in the evaluation. The MVME16X was roughly 100 percent faster than the MVME147 when comparing context switch latencies, and the MVME147 was also 100 percent faster than the GMS-V37.

The biggest improvement in the VxWorks software was in the area of signal interrupt latency and signal task dispatching. The improvement from VxWorks (v5.0.2) to VxWorks (v5.1) was between 60 and 120 percent for signal interrupt latency and between 550 and 930 percent for signal handler task dispatching.

Context switch latencies improved slightly for the MVME147 but showed a greater gain, between 40 and 80 percent, for the MVME16X. One could conclude that the software changes were more beneficial to the pipeline execution scheme of the MC68040 processor.

REFERENCES

1. *VxWorks 5.0 Programmer's Guide*, Wind River Systems.
2. *VxWorks 5.0 Reference Manual*, Wind River Systems.
3. *MVME167/MVME187 Single Board Computers Programmer's Reference Guide*, July 1991 edition, Motorola.
4. *MVME162 Embedded Controller Installation Guide*, March 1993 edition, Motorola.
5. *MVME147S MPU VMEmodule User's Manual*, April 1990 edition, Motorola.
6. *GMS V37-D Special Applications Module User's Manual*, May 1992 edition, General Micro Systems.
7. *User's Manual VBT-321, VBT-321A, VBT-321B, VPC Advanced VMEbus Tracer*, November 1991 edition, VMETRO.
8. *MVME224A-1/-2/-3/-4 Series of DRAM Memory Modules User's Manual*, 1990 edition, Motorola.