Real-Time Performance of the HARRIS RTX 2000 Stack Architecture Versus the Sun 4 SPARC and the Sun 3 M68020 Architectures With a Proposed Real-Time Performance BenchMark

William F. Keown, Jr., Philip Koopman, Jr., Aaron Collins[1]

ABSTRACT

This study compares a stack machine, the Harris RTX 2000, a RISC machine, the Sun 4/SPARC, and a CISC machine, the Sun3/M68020 for real-time applications. An attempt is made to compare the generic features of each machine which are characteristic of their architectural classes as opposed to being characteristic of the individual machine only. Performance is compared based on execution of the Stanford Integer Benchmark series and on interrupt response characteristics. A simple Real-Time Performance BenchMark which integrates raw compute power and interrupt response is proposed, then used to estimate the real-time performance of the machines. It is shown that the RTX 2000 outperforms the others for applications which have a very large number of interrupts per second, confirming that stack architectures should perform well in real-time applications such as high-speed computer communication systems. For less interrupt intensive applications, the Sun 4 SPARC performs better.

Introduction

Microprocessors can be used to control real-time systems, and normally that means that meeting time constraints is the most important computer system characteristic. Both raw compute power and interrupt processing delays are important when attempting to meet real-time system response requirements. First, a brief summary of execution time comparisons for the Harris RTX 2000 (16) versus the Sun 4 SPARC (17,18) and the Sun 3 M68020 (19) machines is presented. Then the interrupt handling mechanisms of the three machines are discussed. Next, interrupt response time measurement procedures are explained. Interrupt response time results are given. The proposed Real-Time Performance BenchMark (RTPBM) is presented, customized for purely interrupt driven processes, then used to predict real-time performance of the computer systems being evaluated. Conclusions are drawn which appear to have general applicability for the classes of computers known as CISC, RISC, and Stack Machines.

The literature has reports which criticize stack architectures as general purpose machines (1,2,3) and which support stack architectures (4,5,6,7,8). The conclusions drawn below which are favorable to stack machines apply primarily to an important class of real-time applications.

Summary of Execution Time Comparisons for Harris RTX 2000, Sun 4 SPARC, and Sun 3 M68020

As a precursor to this evaluation, the Stanford Integer Benchmark series (10) was executed on the three machines listed in the title of this section with the Harris RTX 2000 being evaluated in three different configurations(9). The C compiler for that machine was still under development at the time of the testing, and some inefficiencies were identified in in critical parts of the code such as in the implementation of conditional loops. Specifically, the loop index was saved to memory repeatedly instead of being kept on the stack in the CPU. Consequently, the

RTX was tested with its original pre-release compiler, then later tested with its improved compiler. The RTX was designed primarily to execute the Forth programming language. Although this language has excellent primitives for implementing other languages, it is not optimal for C. Some instruction set changes were proposed by the RTX design team and their effects were simulated in order to better evaluate the potential of stack architectures as opposed to evaluation of this particular stack architecture. The only proposed change of significant consequence was an increment/decrement by N operation for the register used as the C frame pointer. Results were projected for the improved instruction set machine with the assumption that the improved compiler was also available. This case was included because it better represents the potential of stack computer architectures.

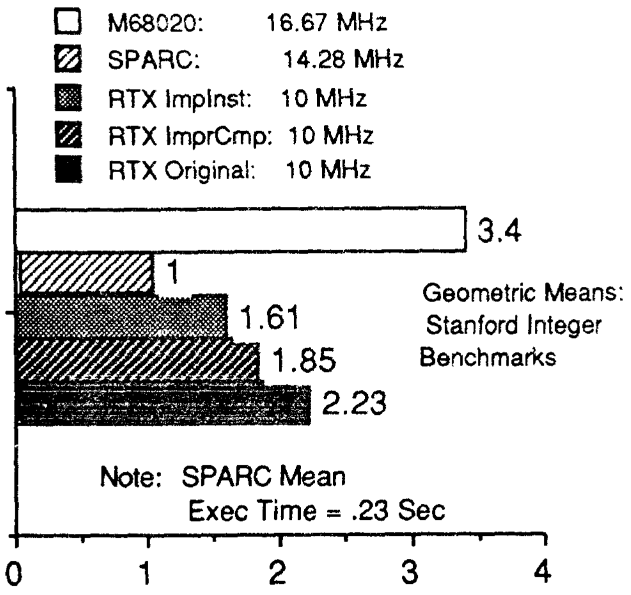


Fig 1. Exec Time Ratio

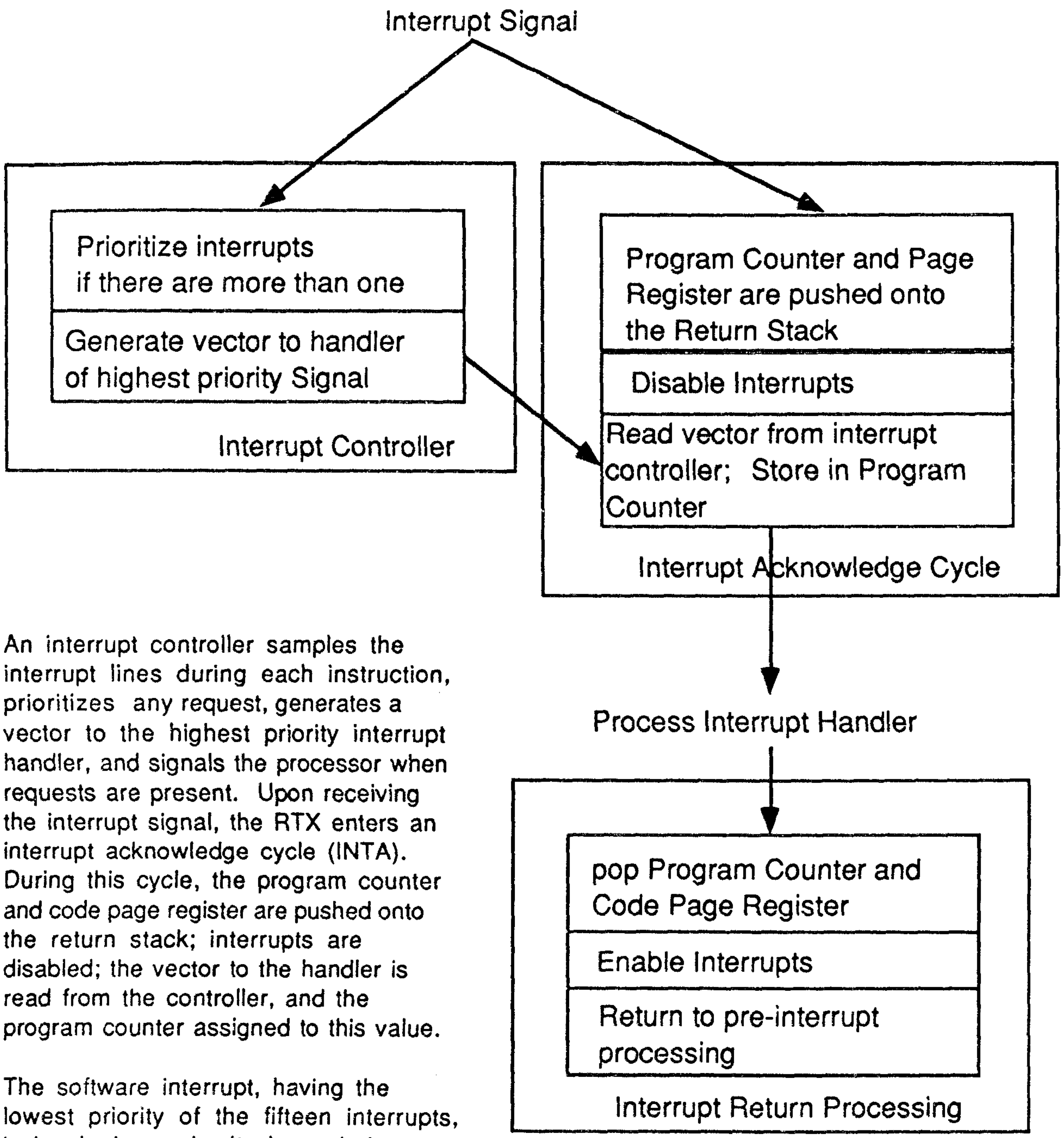
The most relevant results from this study for the purposes of evaluating the real-time processing capabilities of these machines are presented in Figure 1 which shows the execution time of each machine averaged over the six benchmark series. It should be noted that the RTX 2000, a 16-bit machine, operated with a 10 MHz clock, whereas the Sun 4 SPARC had a 14.28 MHz clock, and the Sun 3 M68020 had a 16.67 MHz clock. The latter two machines are 32 bit computers.

The difference in clock speeds between the processors is primarily due to the fact that the fact that the RTX 2000 is fabricated with slower technology, 2.0 micron CMOS standard cell design, than the other microprocessors, and therefore is at a disadvantage for these comparisons. An even more accurate comparison of the three architectures is probably given by the clock cycle count which was obtained in the earlier study. The RTX 2000 was found to execute the average benchmark in 113% as many clock cycles as were required for the Sun4 SPARC, and the Sun3 M68020 required 397% as many clock cycles as did the Sun 4 SPARC. This indicates that the SPARC and RTX architectures would be are quite close in execution time if implemented in equal technololgies.

Interrupt Handling Mechanism

Harris RTX 2000 Interrupt Handling Mechanism:

The RTX 2000 has fourteen levels of interrupts. The highest priority '0' is assigned to a non-maskable interrupt. Three of the interrupt lines are signals from on-chip counters. The five external interrupt inputs have priority 1, 6, and 10-12. Other interrupts are stack overflow and underflow signals for each stack and a software interrupt. Figure 2 illustrates the response of the RTX to an interrupt, and the details of the response sequence are described below.


Interrupt Signal
Prioritize interrupts if there are more than one
Generate vector to handler of highest priority Signal
Interrupt Controller
Program Counter and Page Register are pushed onto the Return Stack
Disable Interrupts
Read vector from interrupt controller; Store in Program Counter
Interrupt Acknowledge Cycle
Process Interrupt Handler
pop Program Counter and Code Page Register
Enable Interrupts
Return to pre-interrupt processing
Interrupt Return Processing

An interrupt controller samples the interrupt lines during each instruction, prioritizes any request, generates a vector to the highest priority interrupt handler, and signals the processor when requests are present. Upon receiving the interrupt signal, the RTX enters an interrupt acknowledge cycle (INTA). During this cycle, the program counter and code page register are pushed onto the return stack; interrupts are disabled; the vector to the handler is read from the controller, and the program counter assigned to this value.

The software interrupt, having the lowest priority of the fifteen interrupts, is level triggered. Its low priority means that it may not be serviced for two instructions following the interrupt instruction. It is also necessary to clear the interrupt signal in the handler before exiting.

Figure 2. RTX Interrupt Handling Response

Returning from an interrupt is the same as returning from a procedure call, except that the interrupts must be enabled. This is done by recognizing that the least significant bit of the return address has been set to a 1 during the INTA cycle.

Sun 4 SPARC:
The SPARC has three types of interrupts, which are referred to as traps. Synchronous and floating point traps are a result of the execution of an instruction. These are assigned the highest priority, priority 1-12. The synchronous traps are usually taken before the next instruction. Asynchronous traps are in response to an external signal. They have priority 13-27. All but the highest priority external interrupt may be masked. In response to any trap, the following set of operations, Figure 3, must occur:

1. disable interrupts (bit ET = 0),
2. save supervisor's mode bit and then set the bit to 1,
3. decrement the register window pointer; a window overflow exception occurs if all windows are active,
4. save program counter(PC) and address to the next instruction (nPC) in the new window,
5. based on the type of trap, set trap buffer register (TBR),
6. save the content of all active global registers, and
7. set PC to the value of TBR and nPC to TBR+4.

When returning from the trap handler, the current window pointer must be incremented modulo the number of windows, This will trigger a window underflow trap if the new window has been assigned to another process since the interrupt occurred. Also, any global registers which were saved before the context switch must be restored.

The software trap instruction, ticc, traps based on the condition specified in 'icc'. The ticc trap has the lowest priority of the synchronous traps, its priority being just above that of the external interrupts.

Sun 3 M68020:
The Sun 3 M68020 has eight external interrupt levels. Except for the reset, the external interrupts all have a lower priority than all synchronous traps except the software interrupt instruction (trap). There are four steps in processing an exception on the M68020. Figure 4 illustrates the process.

1. Copy the status register, set the supervisor bit, inhibit the tracing of the handler, and update the interrupt priority mask.
2. Determine the vector number of the exception and perform an INT_ACK cycle.
3. Save the current process context in an exception stack frame, created on the supervisor stack.
4. Execute the handler by determining the exception vector offset (vector number x 4) and adding it to the vector base register. The program counter is assigned this address and the first three words are prefetched, filling the instruction pipeline.

Interrupt Response Time Measurement

On each of the three systems, a program was executed which was used to measure the time it took the system to respond to an interrupt. Attaching a device to the SPARC to generate external interrupts was not possible, since the machines were housed in a heavily used lab. The alternative was to generate a synchronous interrupt. The first attempt was to generate an instruction exception.

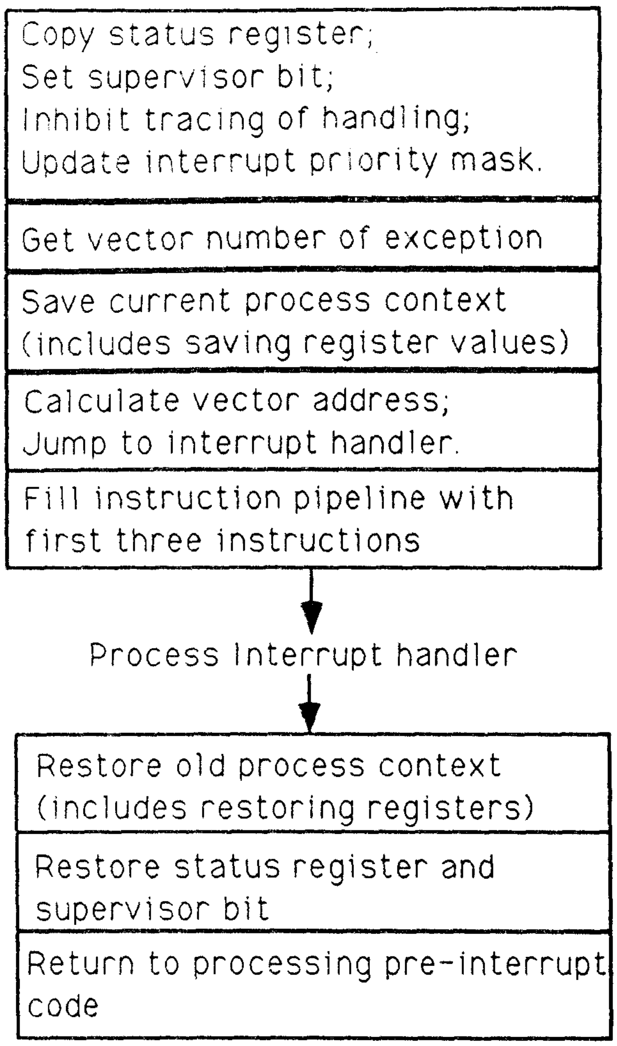

Copy status register;
Set supervisor bit;
Inhibit tracing of handling;
Update interrupt priority mask.
Get vector number of exception
Save current process context (includes saving register values)
Calculate vector address;
Jump to interrupt handler.
Fill instruction pipeline with first three instructions
Process Interrupt handler
Restore old process context (includes restoring registers)
Restore status register and supervisor bit
Return to processing pre-interrupt code

Fig 3 Sun 3 M68020 Interrupt Processing

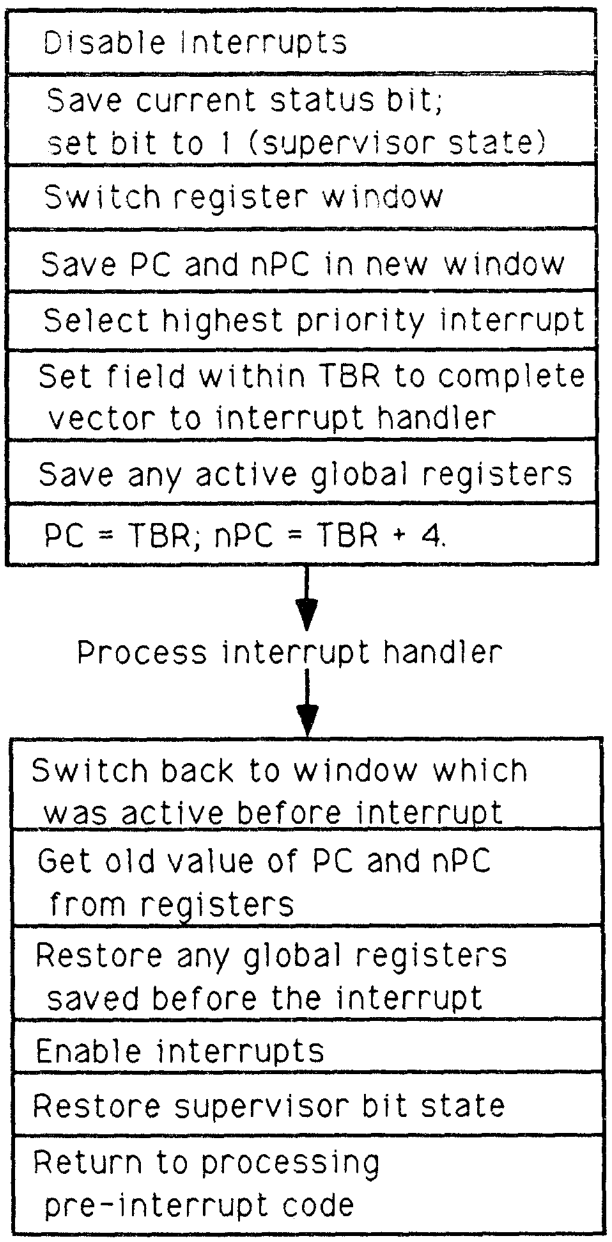

Disable Interrupts
Save current status bit; set bit to 1 (supervisor state)
Switch register window
Save PC and nPC in new window
Select highest priority interrupt
Set field within TBR to complete vector to interrupt handler
Save any active global registers
PC = TBR; nPC = TBR + 4.
Process interrupt handler
Switch back to window which was active before interrupt
Get old value of PC and nPC from registers
Restore any global registers saved before the interrupt
Enable interrupts
Restore supervisor bit state
Return to processing pre-interrupt code

Fig 4. Sun 4 SPARC Interrupt Processing

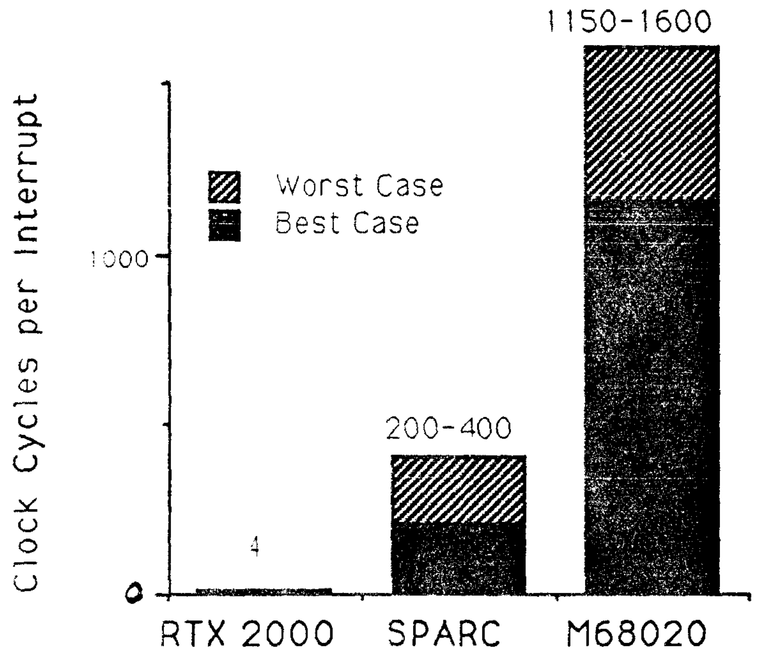


Figure 5. Clock Cycles for Interrupt Response: Two Context Switches Plus Register Saves

The divide by zero exception was initially tried. After adding statements to prevent the compilers from moving a constant expression outside the loop, results were obtained on the Sun 4 SPARC and the Sun 3 M68020. However, the Harris RTX 2000 does not have a divide by zero exception. It assigns zero to any such expression. Therefore to be consistent, the software interrupt instruction of each processor was used. This had two advantages. First, it was not known how long into the execution of the divide instruction the zero divide is recognized on the M68020 and SPARC. Second, except for the RTX, the priority of the software interrupt is either immediately before or immediately after the priorities of the external interrupts.

The program used consisted of a loop of 50,000 iterations, with the interrupt instruction embedded in the loop. In addition, other statements were placed on each side of the interrupt statement. This was done to determine if the type of activity which was occurring before and after the interrupt occurred affected the results of the measurements. Specifically, the effect on the RISC pipeline was a concern. The interrupt handler consisted of a loop of 101 iterations, taking the summation of all loop counter values. This provided a handler of measurable execution time.

The program was executed twice on each machine, with two types of instructions surrounding the interrupt in the loop of the monitor on each machine. One execution had instructions in the monitor which performed arithmetic operations, optimized for register usage on the two SUN systems. The other execution performed a sequence of memory loads and stores. The type of program executed did not affect the time required to perform a context switch on the SPARC for the programs used for this test. In general, on RISC machines, both window effects and pipeline effects affect interrupt response. Specifically, register overflow and underflow (11) can result in substantial interrupt penalties if subroutine calls and ISR calls are nested to sufficient depth to necessitate storing or restoring complete register windows of perhaps sixteen registers in response to a new subroutine call or interrupt. Also, on RISC machine additional overhead or invalidation of some completed work may be required in order to achieve precise interrupts(12). Applications which are significantly affected by these RISC complications will suffer reduced performance as compared to these results, and they will have less predictable interrupt response times.

Interrupt Response Time Results

The RTX is designed to respond optimally to interrupts, and, as shown in Figure 5, incurs almost no overhead for context switching. To switch tasks, only the program counter and code page register must be pushed onto the return stack. Registers do not need to be saved when processing an interrupt on the RTX, because the interrupt service routine is able to simply push its values on top of the stack being used by the foreground task without disturbing the interrupted computation. On RISC and CISC machines, registers must be saved and restored to avoid corruption prior to use by interrupt service routines. Therefore, the time lost performing the two context switches required to service an interrupt and return on the RTX is four cycles, or 0.4 microseconds. The SPARC and M68020 have a much slower response. The SPARC context switching cost for each interrupt is 200-400 clock cycles, or 14 microseconds in the best case. The M68020 costs 1150-1600 clock cycles, or 81 microseconds in the best case. This high cost is due in part to the system configuration. The efficiency of SPARC could be improved by configuring the registers into different window sizes. This would reduce the frequency of window overflow/underflow but it would mean fewer registers per window. This would increase the efficiency of the system unless the reduced number of registers results in more memory traffic. For a real-time system, the system must be tuned for the particular application to get maximum efficiency. The fact that stack machines do not need to save registers prior to entering an Interrupt Service Routine is the primary reason for the superior performance of stack machines in interrupt intensive applications.

These results include the time to perform two context switches plus the other overhead involved in handling an interrupt. The 14 microsecond SPARC result is consistent with results reported by Ingram (13) from SUN Microsystems. She gives the context switch time for a real-time SPARC board to be 5 microseconds. Two 5 microsecond context switches leaves 4 microseconds to evaluate the interrupt, save any global registers, dump any window if no window is free, and restore global registers upon completion. The RTX has no extra overhead other than the two cycles per context switch which is needed to save two registers.

Predicting Real-Time Response

Previously Proposed Benchmarks:

Although benchmarks for evaluating general processor performance abound, few real-time benchmarks have been proposed. The following are components of a real-time benchmark as proposed by Kar in the Rhealstone benchmark (14):

1. task-switch time,
2. preemption time,
3. interrupt latency,
4. semaphore-shuffle time,
5. deadlock-break time, and
6. intertask message latency.

Kar omits the speed at which the processor can execute code and the I/O speed.

A Real-Time Benchmark Proposal:

Since this study is focusing on comparing the basic computer architecture, the list proposed by Kar will be modified as shown below, omitting components which are optional components of real-time systems, and adding CPU performance to the list. Accordingly, the following two items are proposed as components of a simple, but effective, real-time benchmark for comparing competing machines for performance of real-time tasks:

1. interrupt context switch time (sec), T,
2. processor execution time (sec), P.

In equation form, the proposed real-time benchmark can be expressed as:

$$\text{Real-Time Performance BenchMark (sec)} = n * T + m * P$$

where n and m are constants to be selected based the type of application to be considered. Interrupt context switch time is defined as the time to switch into an ISR plus the time to return from the ISR, and could be interpreted as an alternative, but more or less equivalent, to interrupt latency from Kar's list of real-time benchmark components. Its advantage over interrupt latency is that it incorporates all interrupt overhead other than ISR execution time. Processor execution time does not appear in Kar's list, but can be the dominant component of a real-time benchmark for applications which have a substantial amount of code to execute. It should be expressed as proportional to execution time as defined by Hennessey and Patterson (15):

$$P = \text{clock cycle period} * \text{clock cycles per instruction} * \text{number of instructions}$$

Interrupt switch time should be computed based on an evaluation of the architecture or measured. The value computed by this real-time performance benchmark, RTPBM, will increase if interrupt delays increase or if the processor execution time increases. Accordingly, a smaller RTPBM indicates better performance. Note that processor execution could be easily merged into Kar's Rhealstone Real-Time Benchmark by simply listing it as a seventh component of the benchmark. There appear to be many applications which would benefit from that augmentation of the Rhealstone benchmark.

Real-Time Performance Prediction for Interrupt Driven Real-Time Processes:

For the purpose of comparing these three computers, it is assumed that the real-time task of interest is completely interrupt driven. That is, the only processing which takes place occurs in interrupt service routines, and that there is one interrupt service routine to be executed per interrupt occurrence. These assumptions are consistent with a large number of real-time control and communication systems. It would be appropriate to add in many of Kar's components for more complex real-time systems. These assumptions suggest the following values for the constants in RTPBM:

$$m = n = \text{number of interrupts/second}$$

The obvious advantage of this choice of values for m and n is that, if P is chosen as the ISR execution time, the resulting value of RTPBM is the predicted execution time on each processor. This contrasts with the Rhealstone benchmark which yields a rate in Rhealstones/second.

The new RTPBM is expressed as:

$$\text{RTPBM} = n * (T + P)$$

where: n = number of interrupts per second
T = interrupt switch time
P = ISR execution time

Real-Time Performance Prediction Results

Estimation of ISR execution time for the RTX, SPARC, and M68020:

The average execution time ratio for the suite of benchmark programs and the SPARC execution time as determined in the processor performance study (9) are given in Figure 1. If the ISR execution time is measured or estimated for one of the machines, the estimated execution time for the same ISR, when run on another machine, may be estimated using ratios of execution time obtained from Figure 1.

<u>Processor Utilization as Function of Interrupt Frequency:</u>
In Figure 6, the percent processor utilization is plotted against interrupts per second for the SPARC, RTX with improved compiler, RTX with improved instruction set, RTX original, and M68020.

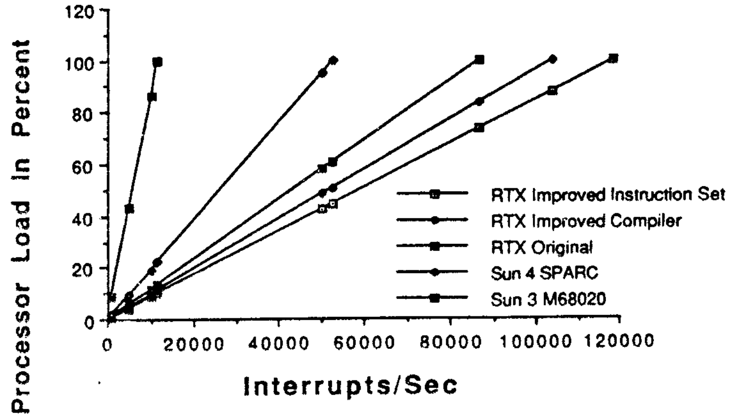


Fig. 6 Processor Load vs Interrupts/Sec

It is assumed that the ISR requires 5 microseconds to execute on the SPARC, and that it therefore requires 1.85 * 5 or 9.25 microseconds to execute on the RTX with an improved compiler. Other ISR execution times are computed similarly.

A task switch time of 400 nsec is obtained for the RTX by multiplying 4 clock cycles times the clock period with a 10 MHz clock frequency. A task switch time of 14 microseconds is obtained for the SPARC by multiplying 200 clock cycles times the clock period with a 14.28 MHz clock frequency. A task switch time of 68.99 microseconds is obtained for the M68020 by multiplying 1150 clock cycles by the clock period with a 16.67 MHz clock frequency. The values for the number of clock cycles per clock switch is assumed to be equal to the best case times expressed in Figure 5 for response to an interrupt.

It may be seen from Figure 6 that the RTX configurations can handle a much higher interrupt rate than can the SPARC machine. The M68020 requires much longer than either of the others. As an example of the application of this information, suppose that it is desired to use a microprocessor as the controller for for the date transfer hardware in a packet processing computer communication system. The system must receive packets which are 256 bytes long at a rate of 30 Megabits per second. Assume that an interrupt is to be generated at the start of each packet, and the end of each packet header, and at the end of each packet. Also, assume that each ISR requires 5 microseconds of SPARC execution time. Five microseconds at 14.28 MHz is 71 clock cycles which yields 40 instructions at 1.76 clock cycles/instruction (9). Which Processors can handle this task, and how heavily are they loaded?

This system will receive 43945 interrupts per second. As shown in Figure 6, the RTX with an improved compiler will be loaded at about 40 percent of CPU capacity. The Sun 4 SPARC will be loaded at about 85 percent of CPU capacity. The Sun 3 M68020 is unable to handle this task. If the length of the ISR is increased, the SPARC will become the machine of choice.

<u>Effect of Interrupt Service Routine On Execution Time for Each Processor:</u>
Figure 7 shows the processor execution time versus the execution time of the ISR on a Sun 4 SPARC processor. When the SPARC ISR execution time is less than 22 microseconds, the RTX with an improved compiler is faster, but for a SPARC ISR execution time greater than 22 microseconds, the SPARC machine is faster. The RTX is superior if there are many interrupts, but the ISR for each interrupt is short. The SPARC is superior if there are few interrupts, and the ISR for each interrupt is long. This result was obtained with the SPARC clock fixed at 14.28 MHz and the RTX clock fixed at 10 MHz.

If the RTX were implemented with a clock speed equal to the SPARC clock, the crossover point would shift to the right, and the stack machine would prove superior up to a much larger ISR execution time. This is illustrated in Figure 8, and represents a better comparison of generic stack architectures versus generic RISC architectures.

Conclusions

Several conclusions can be derived from this data.

The Interrupt handling mechanism on RTX is trivial compared to SPARC and M68020. It requires only 4 clock cycles whereas best case for the SPARC requires 200 clock cycles and the best case for the M68020 requires 1150 clock cycles for its best case.

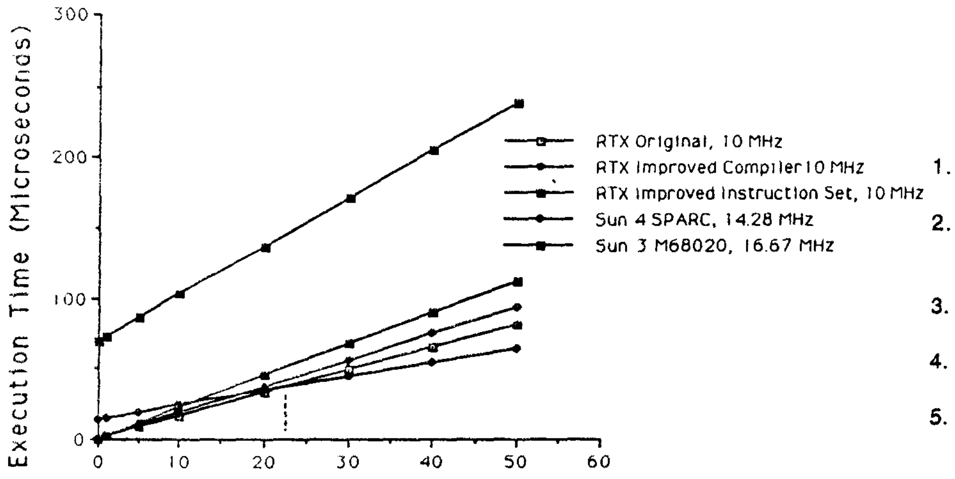
The Rhealstone Benchmark, while useful, requires too much information for basic Real-Time system comparisons. It includes real-time kernel issues that are often too complex and inappropriate for simple real-time controller applications seen in practice. It also fails to include processor performance which may be the dominant component of actual execution time.

A simple benchmark has been proposed which isolates the architectural properties of interest for many applications and which can be customized to provide an actual measure of execution time for some real-time systems.

The RTX can handle much higher interrupt rate than the other two processors. It was shown that for high speed computer communication applications, this can be significant.

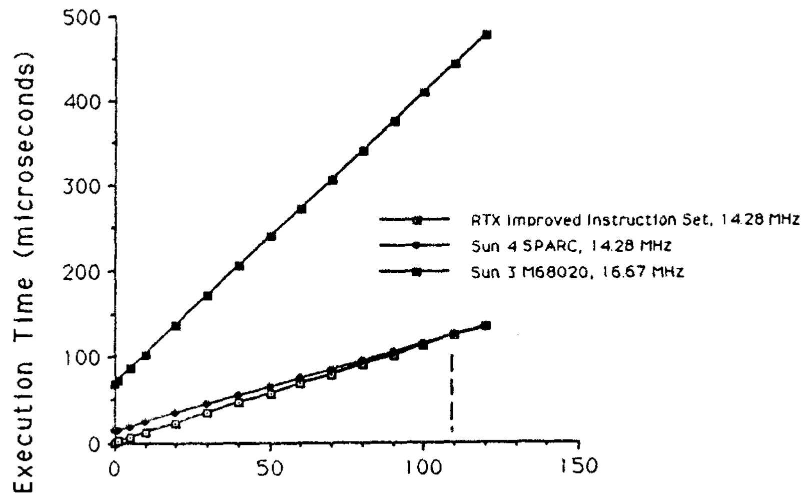
The size of the ISR and interrupt rate were demonstrated to be the proper criteria for choosing between a stack machine or a RISC machine for interrupt intensive real-time applications.

The CISC machine was not competitive with either the RISC or the stack machine in either raw compute power for these benchmarks, or for real-time applications.



ISR Execution Time (SPARC MicroSeconds)

Fig. 7. Execution Time for One Interrupt



ISR Execution Time (SPARC microseconds)

Fig. 8. Execution Time for One Interrupt

References

1. Amdahl, G.M., G.A. Blaauw, and F.P.Brooks, Jr., "Architecture of the IBM System 360," IBM J. Research and Development 8:2, p87-101, April 1964.
2. Bell, G., R. Cady, H. McFarland, B Delagi, J.O'Laughlin, R. Noonan, and W. Wulf, "A new architecture for mini-computers: The DEC PDP-11," Proc. 1970 AFIPS SJCC, 657-675.
3. Myers, G.J., "The evaluation of expressions in a storage-to-storage architecture," Computer Architecture News 7:3, p20-23, October 1978.
4. Koopman, P.J., Stack Computers: The New Wave, Halsted Press, New York, 1989.
5. Keedy, J.L. "On Evaluation of Expressions Using Accumulators, Stacks, and Store-to-Store Instructions," Computer Architecture News, 7:24-27, Dec 1978.
6. Keedy, J.L. "More on the Use of Stacks in the Evaluation of Expressions," Computer Architecture News,7:18-21, June 1979.
7. Tanenbaum, A.S. "Implications of Structured Programming for Machine Architecture," Comm ACM, 21:237-243, March 1978.
8. Miller, D.L. "Stack Machines and Compiler Design, " Byte, 12:177-185, April 1987.
9. Keown, William F., Jr., Philip Koopman, Jr., and Aaron Collins, "Performance of the Harris RTX 2000 Stack Architecture versus the Sun 4 SPARC and the Sun 3 M68020 Architectures", Submitted to Computer Architecture News.
10. Hennesey, John and Peter Nye, "Stanford Integer Benchmarks," Stanford University.
11. Hennessy, John L. and David A. Patterson, Computer Architecture, A Quantitative Approach, p450-454, Morgan Kaufmann Publishers, Inc., 1990.
12. Hennessy, John L. and David A. Patterson, Computer Architecture, A Quantitative Approach, p278-280, Morgan Kaufmann Publishers, Inc., 1990
13. Ingram, K. "Register Windows Set SPARC Apart for Real-Time Applications." VX Works (Wind River Systems), Vol.2, No.2, 1990.
14. Kar, R.P. "Implementing the Rhealstone Real-Time Benchmark," Dr. Dobb's Journal, 46-55, April 1990.
15. Hennessy, John L. and David A. Patterson, Computer Architecture, A Quantitative Approach, p36, Morgan Kaufmann Publishers, Inc., 1990.
16. RTX Reference Manual. Harris Corp. 1988.
17. The SPARC Architecture Manual. Version 7. Sun Microsystems. 1987.
18. SPARC Release Notes for 4.0. Sun Microsystems. 1988.
19. Sun 3 Reference Manuals, Sun Microsystems, 1987.

PERFORMANCE EVALUATION REVIEW

a
c
sigmetrics

A Publication of the ACM Special Interest Group on Measurement and Evaluation

acm PRESS

Volume 19 Number 4 May 1992

* Refereed