Effects of Reconfiguration on Performance in Configurable Operating Systems : Practical Predictability Strategies ?

Raj Krishnamurthy(rk@cc.gatech.edu) and Karsten Schwan(schwan@cc)

1.0 Introduction

Critical systems must be configured to meet changing functionality, time-criticality and fault-tolerance needs. Configurations may be performed statically at operating system build or boot -time. Dynamic configurations are also possible during 'run-time', when the operating system is loaded and running. For example, operating sytem kernel modules, operating system components, middleware and application programs may all be configured statically or dynamically. Operating systems may be configured to scale from ROM-able versions to full-fledged multiprocessor clusters, which we term, the 'horizontal' configurability feature. In addition, to address different types of applications, operating systems may be configured with enhanced or reduced functionality, which we term the 'vertical' configurability feature. Finally, when such configurations are performed, higher level operating system components, middleware, and other Commercial Off-The-Shelf (COTS) software products expect to experience the gains derived from such configurations in terms of enhanced performance or predictability.

A structured way to study the performance of a configurable target system is by use of benchmarks. The benchmarks designed and used in this paper aim to evaluate the relationship between the functionality, predictability of components and configured systems. The effects of reconfiguration on overall performance may be studied by execution of the benchmarks before and after reconfiguration. This paper considers the performance effects of 'vertical' and 'horizontal' reconfiguration on some operating systems, by executing operating system primitives across a variety of configurations.

The primary contributions of this paper include a benchmark suite for study of performance variations and a 'cost-graph' strategy for performance prediction in configurable operating systems. A 'cost-graph' for each operating system primitive of interest, is built on system creation and re-built each time the system is reconfigured. The purpose of the 'cost-graph' is to capture actual system performance and its derivation relies on both predicted and observed system behavior.

The paper is organized as follows: Section 2 discusses horizontal and vertical configurability. Section 3 discusses the effect of reconfiguration on performance. Section 4 details the benchmark suite. The experimental infrastructure used for studying performance variations is described in Section 5. The results of our reconfiguration-performance variation studies are described in Section 6 and explained in Section 7. Section 8 describes the 'cost-graph' approach for predicting performance and applies it to measurements and configurations appearing in Section 6 and Section 7. Other potential applications of the 'cost-graph' strategy are explained in Section 9. Section 10 concludes the paper.

2.0 'Horizontal' and 'Vertical' Configurability

'Horizontal' configurability allows the operating system to be scaled from small footprint ROMable versions to full-fledged heavyweight multiprocessor clusters. The operating system must provide facilities and services to allow the operating system to scale and run on multiple platforms, to harness in full the architectural capabilities available on a platform. In Vxworks [vxworks] for example, the operating system may be scaled from a uniprocessor installation to a multi-processor installation by use of VxMP. VxMP provides shared objects to allow syncronization across multiple processors. 'Horizontal' is used here to indicate an increase in the scale - number of processors. A highly 'horizontal'-ly configurable operating system would be one, where from a single object or source code base, operating systems may be built to occupy varying footprints to run on a range of hardware platforms. ROMable kernels occupy a small footprint, are lightweight and may contain minimal data structures and minimal operating system facilities. Kernels for uniprocessor systems may be large, with a number of kernel data structures maintaining system state and a full complement of operating system facilities. Kernels for multiprocessors must maintain system state for multiple processors and manage resources across the multi-processor cluster and are expected to be even more heavyweight.

'Vertical' configurability allows the operating system to be scaled in functionality up or down in the 'vertical' direction on a fixed hardware platform(with fixed architectural capabilities). For example, on a hardware platform of choice, to suit the needs of multiple applications, the operating system may be varied in functionality based on footprint-performance constraints of a target application. Kernel richness may be enhanced by layering functionality or functionality may be reduced to suit the needs of an application. Functionality may be specified in terms of system call sets (blocking, non-blocking, preemptable etc.) or operating system facilities(VM management, IPC facilities etc.) .

Reconfiguration to introduce or remove functionality may cause performance variations in certain operating system primitives. This is because enhanced functionality may cause extra data structures to be introduced into the execution path of a primitive, more facilities requiring execution, leading to increased execution time. Actions in the reverse direction may also be true. Fewer data structures may require maintenance and fewer facilities may require execution in the case kernels are scaled down to remove redundant functionality. This can lead to reduced execution times for operating system primitives.

3.0 Reconfiguration and Performance Variations

Consider process dispatch latency time, a popular measure to evaluate operating systems [Furht]. The operations involved when a lower priority task is preempted to run a higher priority task on receipt of an interrupt are shown in figure 1.

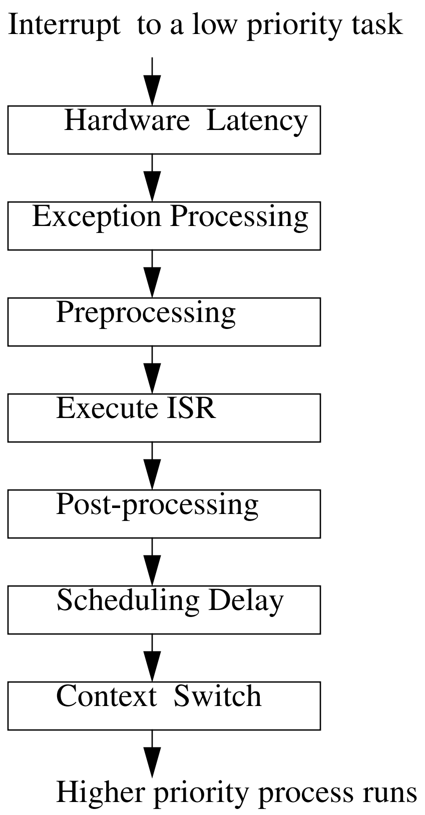


Figure 1: Process Dispatch Latency

Hardware latency lumps the electrical delays, bus delays and other hardware related delays introduced by the system. Exception processing refers to the latencies caused by state maintenance in processors due to precise and imprecise exception handling [patterson]. Preprocessing delay includes state/context saving, setting up a stack for the interrupt service routine, locating the interrupt service routine (may or may not involve trap to operating system, depending on implementation). The interrupt service routine is executed and may be preempted (depending on the implementation) to accommodate nested interrupts. Post processing involves restoration of registers and stack and exit of the interrupt service routine. The scheduler is then invoked to run the higher priority process. A context switch may be required to run the higher priority process.

The effects of reconfiguration of the operating system on this metric may be incurred as a direct cost and/or as an indirect cost. The next two sections describe the direct and indirect costs of operating system reconfiguration on the process dispatch latency metric.

3.1 Direct Costs

Direct costs are incurred due to components directly in the execution path of a metric. In the case of the process dispatch latency metric, this could be because of the facility currently executing, preprocessing, the interrupt service routine itself, post processing, scheduling or the final context switch. Effects of reconfiguration due to components directly in the execution path of the metric are considered in the following sections.

3.1.1 Preemptable Services

Operating system kernels may be fully preemptable, non-preemptable or non-interruptible. Fully preemptable kernels require multiple kernel stacks and additional kernel structures so that the execution context of any service may be saved and resumed later upon interruption. Certain sections of a preemptable service may require preemption or interrupt lock-out to protect critical sections. Extra overheads in terms of execution time may be incurred by implementation of a fully preemptable kernel or service. A non-preemptable kernel may not need additional kernel stacks and data structures and will have lower space requirements.

Consider a configuration where a non-preemptable service is in operation. This may have been chosen in the initial configuration to reduce kernel space requirements. Upon receipt of the interrupt, the service routine is run. The service routine is completed and the non-preemptable service initially in operation may have to be completed, before the higher priority process is scheduled. The time taken to complete the non-preemptable service adds to the dispatch latency metric. The same operating system kernel is then reconfigured statically or dynamically to include a fully preemptable version of the same service as the above configuration. In this configuration, the higher priority process may be scheduled immediately after the interrupt service routine is completed. By trading speed for size, the process dispatch latency metric is improved in the second configuration. The effects of reconfiguration in this case are to improve the process dispatch latency metric.

Similarly, in the case a fully preemptable service replaces a non-interruptible service, improvement will be seen in the metric as the interrupt service routine may be executed directly upon receipt of the interrupt. The effects of reconfiguration again in this case are to improve the process dispatch latency metric.

3.1.2 POSIX and Native Services

Commercial operating systems implement native services and corresponding POSIX services in a number of ways. Solaris implements POSIX message queues as wrappers around SVR4 message queue calls, QNX implements POSIX message queues as an external process outside of the kernel. Vxworks implements POSIX message queues and native message queues in the kernel.

Consider the process dispatch latency metric. The interrupt service routine is invoked and sends a message using the native message queue facility. If the kernel is reconfigured to include the POSIX message queue facility then, variations in the latency metric may be observed. This is could be because of the difference in implementation of the native message queue (in the kernel) and the POSIX message queue (as a separate process).

3.1.3 Scheduling Delays

Operating systems may be built as cyclic executives, with fixed or dynamic priority schedulers. A high level scheduling policy may also be implemented with processes running under fixed or dynamic process scheduling. Preemptive or non-preemptive schedulers are also implemented. Schedulers are also implemented with different data structures and this can affect the scheduling delays. Consider a configuration built with a certain scheduler (with a single or multiple scheduling policy). If the kernel is reconfigured to include a different scheduler then, variations in the process dispatch latency metric may be seen. This may be because of implementation costs of a different scheduler and/or the scheduling policy. The implementation costs a scheduler may be simply because of different data structures used in the scheduler. In Vxworks for example, a constant ready queue insert time implementation may be included. This adds 2K to the size of the kernel but, helps in overall scheduling when large number of task are ready to run.

3.2 Indirect Costs

Reconfiguration of the kernel or any service may also affect the performance of a metric in an indirect way. Direct costs are because of components directly in the execution path of a metric, whereas indirect costs are due to components not directly in the execution path of a metric. The indirect influence of components may be pronounced in some cases and cannot be neglected. Inclusion of virtual memory, Shared objects or network services can all affect the performance of a metric in an indirect manner. This section identifies some indirect costs in the case of the process dispatch latency metric.

3.2.1 VM Subsystem and Network Services

Consider the process dispatch latency metric, if the interrupt handler is in physical memory or is pinned in memory then, the interrupt handler may be invoked directly. If reconfiguration of the kernel leads to inclusion of the VM subsystem, then, it is possible that the interrupt handler may not be locatable in physical memory and may have to invoked after a page-in leading to poor overall performance of the metric.

Similarly, consider a configuration where the kernel has been reconfigured to include network services. During execution of the components leading to dispatch of the higher priority process, the network may require service (queues full because of packet arrivals). This can add to the execution cost of the metric.

3.2.2 Interrupt Priority Management

A well-designed interrupt handling mechanism, will allow the currently executing task to be interrupted by interrupts tied to only a higher priority task. With hardware support for interrupt priority 'bridging'(between software tasks and hardware interrupts) in the Intel i960 processor for example, software task priorities are saved in registers so that, only interrupts tied to a higher priority task may interrupt the currently executing task. Consider a configuration where support for interrupt priority 'bridging' is not included. In this case, interrupts from lower priority sources than the currently executing task may repeatedly interrupt leading to possibly poor dispatch latency times. If the kernel is reconfigured to allow priority ‘bridging’, only interrupts from higher priority tasks than the task currently executing may be received, resulting in improved dispatch latency times.

3.3 Tracking Performance Variations

The preceding sections detail the effect of reconfiguration on the process dispatch latency metric. The costs may be incurred in terms of direct and indirect costs. Similarly, this can be easily extended to other operating system primitives of interest. This is easily seen, if we consider the execution path of a primitive to be decomposable into one or more stages. Reconfiguration may affect the configuration of one or more stages in the execution path of the primitive. This may increase or decrease the execution latency of one or more stages, resulting in an effect on the execution latency of the primitive as a whole. This is incurred as a direct cost. Reconfiguration costs may also be incurred indirectly. Scheduling of a service during the execution of a primitive, increases the execution latency of a primitive. This may not be incurred directly, as the service may not be in the execution path of the primitive.

If the performance of certain operating system primitives are of interest to a target system and application, then it is essential to track the performance variations due to reconfiguration. This may be done by maintaining a cost-graph data structure. Each primitive is described by a cost-graph. The cost-graph records the execution latency of each stage in the execution path of a primitive. Execution latencies for stages may be lumped in the case that latencies for individual stages are not individually measurable. Each direct reconfiguration action may introduce an additional stage or it may affect the execution latency of an individual stage. If an additional stage is introduced then , the stage may supply to the cost-graph, the worst-case execution time on that particular platform. This is important in the case apriori performance measurements are needed. Also, if desired, the cost-graph may be updated or supplemented with execution latencies obtained by execution of the primitive immediately after reconfiguration. Indirect costs may be represented in the cost-graph by lumped worst-case execution times.

Each reconfiguration action is represented in the cost-graph by its corresponding cost. The cost may be supplied apriori by the reconfiguration action or may be determined by execution of the primitive. By recording each reconfiguration action and representing the action by its corresponding execution cost in the cost-graph, the variations may be tracked.

A benchmark suite for measuring the effects of reconfiguration on some operating system primitives of interest is described in the next section.

4.0 Benchmark Suite

The objective of this paper is to study the effects of reconfiguration on certain operating system primitives. A set of metrics has been chosen that capture important aspects of an operating system‘s behavior. These metrics have been selected from the rhealstone benchmark suite [Dr dobbs] , the SSC benchmarks [nuclear physics citation] and the Hartstone benchmark suite [cmu]. The basic idea is to select or choose different operating system configurations and study the effect of varying the configurations on operating system performance. The variations (if any) in the selected metrics will be indicative of the effects of reconfiguration on operating system performance.

The benchmark suite consists of low-level benchmarks and high-level benchmarks. Low-level benchmarks are based on the Rhealstone and SSC benchmarks, whereas the high-level benchmarks are based on the Hartstone benchmarks. The following sub-sections describe the level one (low-level suite) and the level two(high-level suite).

4.1 Level one Suite

The low-level suite is based on basic operating system primitives. The low-level metrics chosen are : context switch time < task preemption time >, semaphore shuffle time, inter-task message latency, memory allocation overhead and process dispatch latency. The metrics are measured in the following manner :

4.1.1 Context Switch Time

This metric measures the time to context switch between two tasks of equal priority. This involves saving the context of the current task, loading the context of the other task and rescheduling. A controller task creates two tasks of equal priority and allows scheduling of the two tasks by lowering its priority. The two tasks ping control from one another over a large number of iterations. The average context switch time may be measured along with the min and max context switch times.

5.0 Providing an Experimentation Infrastructure

The VxWorks development infrastructure consists of a Sun Ultra based host and Pentium II based dual processor target. The development host consists of a Tornado shell, Target Server, Host Symbol table (may be synchronized with target symbol table). The Vxworks target consists of an agent that communicates with the host environment. The target may also contain a symbol table for synchronization with the host. New objects may be dropped into the target. The target dynamically links the object with the operating environment and executes the code. This interaction is shown in figure 2.

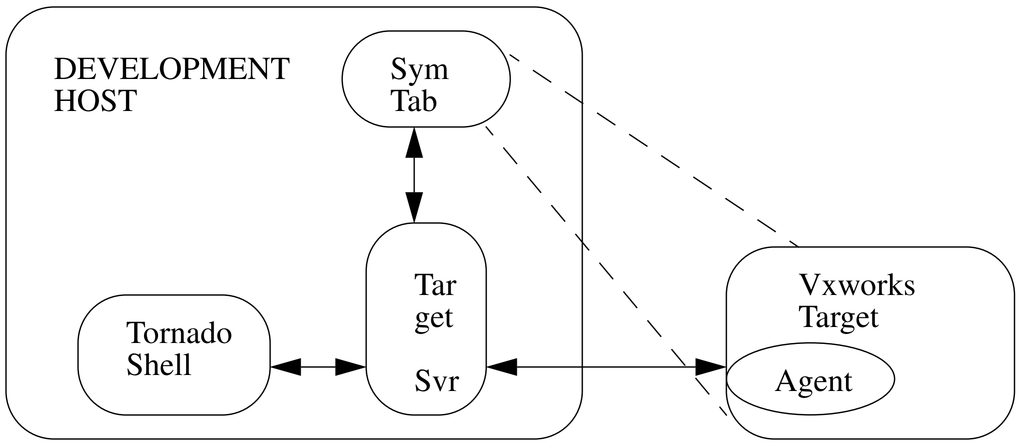


Fig.2 . The Vxworks Infrastructure

5.1 A Device Handler to Support Kernel Instrumentation

As the figure shows, the target interacts with the development environment. The development environment allows the ability to launch instrumentation probes into the kernel. The kernel must be instrumented to accept the probes. The target contains a network interface to interact with the host development environment. The Vxworks development environment provides a PCI Bus library [vxworks], to support PCI bus transactions on a PCI based host. The device handler provides the interface between the Operating System and the Network interface card.

5.2 A Timestamp Driver

The Pentium II has hardware counters and for fine-grain resoultion, it is important to interface to the Pentium II hardware counters directly. The Timestamp driver interface from the Operating System to the API have already been constructed. The interface from the Operating System to the hardware counters are being constructed.

5.3 Cross Development Environment

A Cross Development environment has been constructed to compile code directly from Sun Ultras to the Pentium II. Upon compilation, the code is downloaded to the target.

6.0 Performance Methodology

First, core operating system primitives are identifed. In this case, they are interrupt dispatch latency(process dispatch latency), Semaphore ping and Message Passing. The Operating system is then reconfigured (different facilities, services added). For each reconfiguration, the performance of the metrics is measured. The measurement of the metrics is performed by performing an operation and iterating over large iterations (around 10000)

7.0 Performance Measurements

The performance measurements were made for the following three core operating system primitives - Process Dispatch Latency, Semaphore Ping and Message Passing.

7.1 Process Dispatch Latency

This is shown in Table 1 . This benchmark is measured by using three interrupt sources - Timer, Bus and Interprocessor. A task is spawned and the task blocks on a semaphore. Once the interrupt arrives, the interrupt service routine releases the semaphore. The task with the blocked semaphore then, timestamps the event.

The columns denote Wind scheduler, POSIX scheduler and a combination of the Shared

Table 1. Interrupt Source and PDLT

Interr upt Source	Wind sched	POSIX sched	SM Wind sched
	<------- microsecs -------->		
PIT	17	34	40
Bus	50	90	120
Inter- proces- sor	55	95	130

Memory Objects and the wind scheduler.

PIT is the PIT clock that generates interrupts at the rate of 60,000 per second. A bus interrupt is generated from an external source. In this case it is the second CPU in the machine. In the case of the Interprocessor interrupt, a mailbox interrupt is used and one CPU places a data value into a one-byte write mailbox. This is a "location monitor" style of interrupt management. The write mailbox is basically a bus address, that interrupts another CPU.

In the case of the PIT, use of the POSIX scheduler seems to increase the Process Dispatch latency. As this may be layered on top of the native wind scheduler. Also, use of Shared memory objects further increases the Process dispatch latency to 40 microseconds.

In the case of the bus interrupt, the same the increasing trend is seen. Here, it must be noted that the differentials are much larger than the case of the PIT timer. This might be due to the effect of bus delays, I/O bridging etc.

This is also true in the case of the interprocessor interrupt. The same increasing trend is seen, all the way from the wind scheduler to the combination of the Shared memory objects and the Wind scheduler case.

Fig. 2 graphs the OS state as a function of time. levels 1 and 2 are for tasks. They may represent priority levels. Level 3 is for interrupts and interrupt service routines.

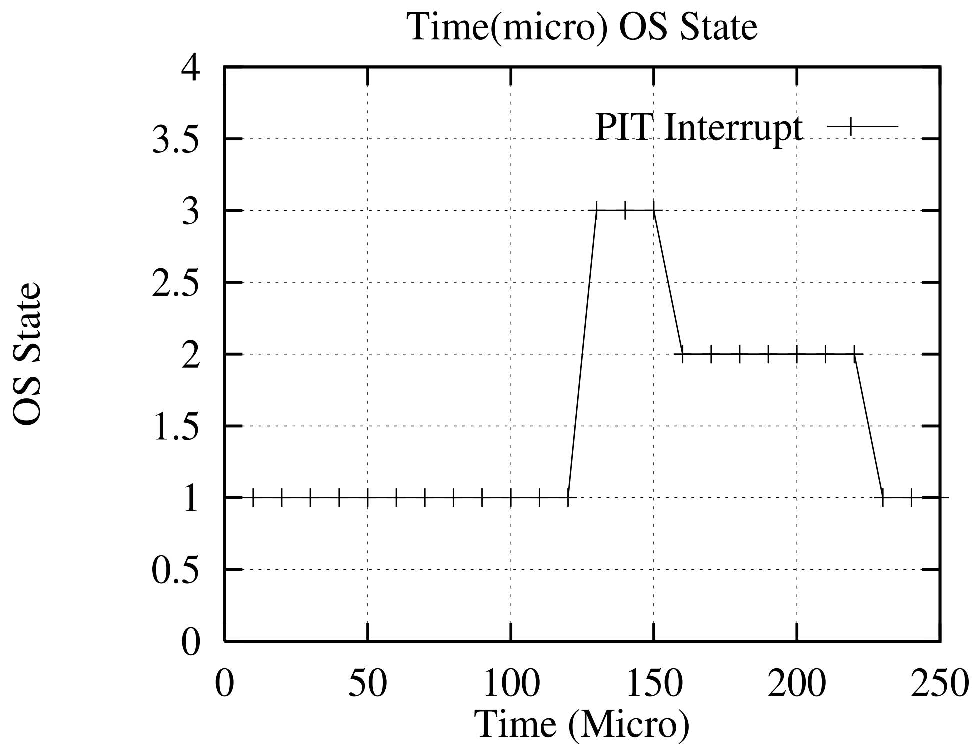
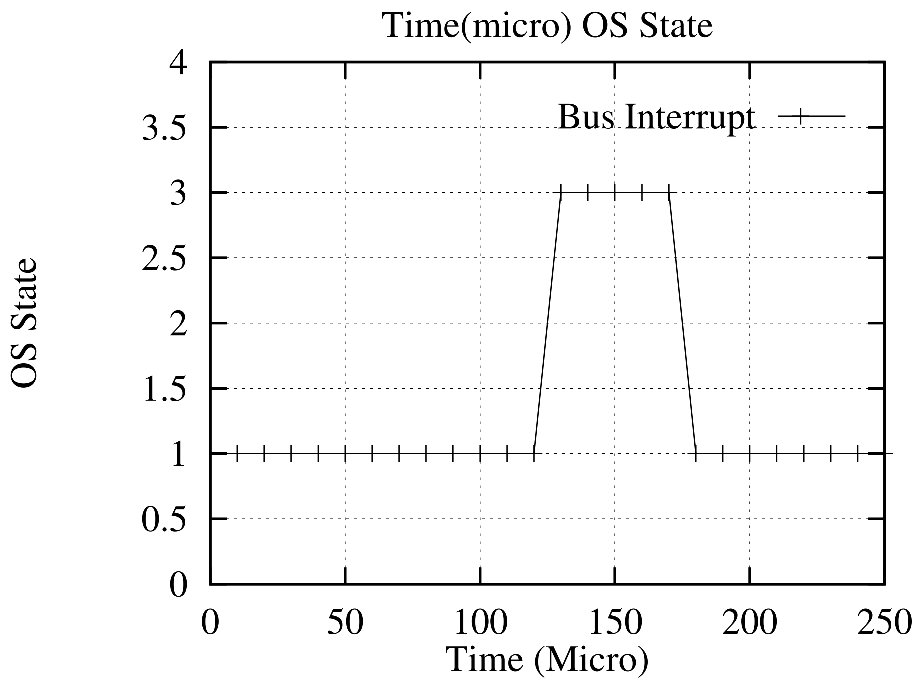


Fig. 2 OS State as a function of time

Fig.2 shows in the case of the PIT timer interrupt, the variation of the OS state and time. The task is initially, in level 1, transitions to level three in the case of an interrupt, flips back to state two (in this case a process is spawned). The process or task may be deleted and OS state may return back to state 1 again.

The OS state transition is shown again in figure for the case of bus interrupts. In this case, the OS transitions back to the level of task, after executing the interrupt in state 3.



7.2 Ping Semaphore

For this metric, two tasks ping a semaphore between each other. This is done over large number of iterations. The results are shown in the table below. The latency for POSIX semaphores is higher than that for wind native semaphores. This is true in the case of using either the wind scheduler or the POSIX scheduler.

Table 2. Semaphore Ping

Sched	wind	Posix
	<-- microsecs	-->
Wind	225	250
Posix	225	240

7.3 Message Passing

For this metric, two tasks exchange messages between each other. This is done over large number of iterations. The results are shown in the table below. The latency for POSIX message queues is higher than that for wind native message queues. This is true in the case of using either the wind scheduler or the POSIX scheduler. The queues in both the cases are of fixed length.

Table 3. Message Passing

Sched	wind	Posix
	<-- microsecs	-->
Wind	300	350
Posix	300	340

7.4 Effect of Operating System Stacking on Context Switch Time

Table 4: Effect of "Stacking" on Context Switch Time

Module	Change (micro)
For- matted I/O	-
HW FP	
INSTR UMEN TATIO N	+20%
I/O	-
LOAD ER	
LOG	-
MEM_ MGER	-
MMU_ BASIC	-
MMU FULL	-
MSG Q	-
NET- WORK	-
NFS SERV ER	-
PIPES	-

Table 4: Effect of “Stacking” on Context Switch Time

Module	Change (micro)
SEM_ BINAE Y	-
SEM_ COUN TING	-
SEM_ MUTE X	-
SIG- NALS	-
SM_O BJ	+10%
SYM_ TAB	-
TASK_ HOOK S	+10%
CONS TANT_ RDY_ Q	10 %

Table 4 shows the “stacking” effect on the Vxworks operating system. In the case of introducing the shared memory objects, the context switch time goes up by about 10%. This may be due to additional processing being performed during a context switch. Task hooks are kernel call-outs. They are executed at the time of a context switch. Even for a null or kernel call-out with no function defined, an increase of 10% in the context switch time is seen. In the case of the constant ready queue, a 10% enhancement in performance is seen by use of constant insert time ready queues.

8.0 The Cost-graph Approach to Performance Predictability

We implement a cost-graph, which is placed into usrConfig.c. As the kernel is reconfigured, the Cost-graph is updated statically. The same function may be used for dynamic reconfigurations. Upon completion of benchmarks, the cost-graph may be downloaded to the host for analysis.

9.0 Conclusions and Future Work

- Developed infrastructure for real-time experimentation - NI handler and Timestamp handler (port to MMX counters required).
- Reconfiguration affects performance !
- Both direct and indirect
- Large reconfiguration space (cross product) - Find useful ones
- Enhance timer resolution
- Look at Shared memory backplane network closely (on Vxworks adds constant or varying overhead).

References

[Furht] Furht et al, “Real-time Unix”, Kluwer Academic Publishers, 1991.

[VxWorks] Wind River Corp. “Vxworks Programmer’s Guide”, 1997.