1. Soft Affinity Design

1.1 Introduction

The definition of soft affinity is to allow a process/kernel thread to be scheduled on the last processor that it ran on thereby preserving its investment in the cache and TB. As a result, one can expect the process to execute more instructions per quantum by effectively using the data in the caches and TBs. The purpose of this design is to provide a soft affinity algorithm which allows the user to enable this feature and improves process performance while at the same time not negatively impacting system performance.

An investigation report was done prior to this design. This report showed that the current soft affinity algorithm in VMS does not yield a performance win and that a new algorithm must be designed to yield the benefits of soft affinity for a process. This report also investigated possible algorithms. This design is a continuation of the investigation report and a detailed write-up of the final algorithm chosen. At the end of the investigation report, several design issues were mentioned. This design report will address these issues and their resolution in section 1.4.

One conclusion that I have drawn since writing the investigation report and prototyping a significant portion of this design is that the coding changes are fairly straightforward. The analysis and characterization of how this algorithm/design works is more difficult and time-consuming. This is due to the fact that even the smallest change in the code or a system parameter can yield vastly different results. This coupled with how frequently the code paths are executed make for a time-consuming task to explain the results. The reason for mentioning this is that even though this design spec is complete and accurate as possible, there may be a need for some design "tweaking" once this design is implemented and its behaviour in many different environments can be observed.

It should be stated here that this project is Y2K compliant.

1.2 Design Overview

Currently, there is a system service, $SET_PROCESS_AFFINITY, which enables soft affinity for a process/kernel thread. This system service today randomly chooses a soft affinity CPU for the process if none is specified on input. With this design, a more intelligent guess is possible. In addition to this system service, there should also be a DCL interface for soft affinity. This interface will take the form of a new keyword with the /AFFINITY qualifier on the SET PROCESS command as well as the affinity setting being displayed in the SHOW PROCESS command. Today, the DCL command, SHOW CPU/FULL, will show which processes are bound either with hard or soft affinity to each CPU.

Looking beyond the user interface for soft affinity, there are two major pieces to this design. The first piece is the data collection and the second is the soft affinity algorithm itself. The data collection is important since the soft affinity algorithm bases its decisions on this data. From the investigation report, it was determined that the following were significant pieces of data which influences where a process should run:

- run, wait, and interrupt times for each soft affinity process/kernel thread

- number of voluntary waits over a specified time interval
- number of times a process/kernel thread has run on a particular CPU
- run, wait, and interrupt times for all process/kernel threads based on priority level and kept on a per-CPU basis
- number of processes/kernel threads run on each CPU

The data for the first three items in the list above will be stored in the KTB. The specified time interval mentioned in the second item will be controlled by a system parameter, AFFINITY_TIME. The reason for collecting the number of voluntary waits is to determine if the process is compute or I/O bound. The soft affinity algorithm treats these two types of processes differently. As for the fourth item it is deemed important to know how processes/kernel threads in general are performing on the individual CPUs. Therefore, the per-CPU database will contain an array indexed by priority (priorities 0 - 16). The real time processes will all contribute their data to the entry in the array indexed by priority 16. The reason for this is it is believed that soft affinity applies more to time sharing processes/kernel threads.

The second major piece is the soft affinity algorithm. This algorithm is present when choosing a process/kernel thread to run on a CPU. The VMS scheduler will compare the capability/affinity requirements of a process/kernel thread to that of the CPU. If there is a mismatch because of soft affinity, the scheduler will check to see if soft affinity is for the current CPU. If so, the process will be chosen for execution (assuming no other capability mismatches). If not, then this new soft affinity algorithm will be executed. The first decision this new algorithm will make is to determine whether the process/kernel thread is I/O bound or compute bound. It does this by determining if the number of voluntary waits collected over a specified time period exceed the value of (1/2*(quantum/iota)). If so, then the process/kernel thread is I/O bound. Otherwise, it assumes it is compute bound. The reason for this distinction is that compute bound processes/kernel threads need to run on a CPU which can give it the longest run time with the shortest wait time. Compute bound processes/kernel threads typically need to run until quantum end. On the other hand I/O bound processes/kernel threads don't necessarily need the longest run time. These type of processes/kernel threads will only run for a few soft ticks and then voluntarily give up the CPU waiting for an I/O to complete for instance. So as long as they can get enough run time to voluntarily give up the CPU, then the process requirements are satisfied. Therefore, the algorithm for I/O bound processes/kernel threads is based on the average runtime and for a CPU who can satisfy the process' average runtime but not necessarily choose it because it can give it the highest average runtime. For I/O bound processes/kernel threads, the algorithm is:

1. Compute the average runtime for the current CPU (based on scheduling data from the CPU database).
2. Compute the average runtime for the process/kernel thread on the soft affinity CPU (based on data from the scheduling data pointed to by KTB$AR_SCHED_DATA).
3. Compare the averages calculated in 1. and 2. If value of 2. is greater than the value of 1., then stay with the soft affinity CPU since the current CPU cannot give the process/kernel thread the runtime it needs.

4. Compute the average runtime of processes/kernel threads on the soft affinity CPU (at the same priority as the process/kernel thread being scheduled).
5. Determine if the runtimes of the 2 CPUs are close where close is defined as within 10% of the soft affinity CPU.
6. If the run times are not close and the average runtime of the current CPU is greater than the soft affinity CPU, then stay with the soft affinity CPU. Otherwise, schedule the process/kernel thread on the current CPU.
7. If the run times are close, then the wait time will be the deciding factor. Calculate the average wait time for both CPUs. If the average wait time of the current CPU is greater than the average wait time of the soft affinity CPU, then stay with the soft affinity CPU. Otherwise, schedule the process/kernel thread on the current CPU.

On the other hand, the algorithm for compute bound processes is based on the best run/wait ratio since the process is looking for the CPU who has the best run time. Here's the steps in this algorithm:

1. Compute the (run-interrupt)/wait ratio of the current CPU
2. Compute the (run-interrupt)/wait ratio of the process on the soft affinity CPU
3. If the (run-interrupt)/wait ratio of the current CPU is greater (by 10%) than the (run-interrupt)/wait ratio of the process on the soft affinity CPU, then break soft affinity and schedule the process. Otherwise place the process back on the compute queue and search for another process to run.

The value of 10% in step 3 above is called the bias value. In order to break soft affinity, there needs to be a significant gain in performance to justify the loss of the investment in the cache and Tbs. A process' soft affinity can be broken in two different ways:

- if either of the above algorithms indicate that a process is a better candidate for the current CPU rather than the CPU the process has soft affinity for.
- if the current CPU can not find a process to run in the COM queues (due to the fact that all processes have soft affinity for a different CPU and even after evaluation, results indicate that the process should remain with the soft affinity CPU), it will choose the first soft affinity proc ess in the queue and break its' current soft affinity setting.

The second item in the list above is the only piece of the previous soft affinity design/implementation that has remained.

It should be noted here that the soft affinity algorithm uses floating point instructions to perform the quadword divisions. The algorithm will enable floating point prior to performing the divides if it is not already enabled and disable floating point when it is done (if need be). Since the Macro-32 compiler does not support floating point instructions, the routine, SCH$FIND_NEXT_PROC, in SCHED_ROUTINES.MAR, invokes the routine, SCH$SOFT_AFFINITY, in module, SCHEDULER.M64.

1.3 Detailed Changes

1.3.1 Data Structure Changes

As mentioned in the Design Overview section, data collection plays an important role in implementing soft affinity. There are several changes related to the scheduling data structures in support of data collection. The first change is to the per-CPU database. An array containing scheduling data indexed by priority level will be added to the per-CPU database. The first quadword of the array will be CPU$Q_SCHED_DATA. Below is the format of this array:

accumulated run time for priority 0 process

# of processes run at priority 0

accumulated interrupt time for priority 0 processes

accumulated wait time for priority 0 processes

…..

…..

accumulated run time for priority 16-63 processes

# of processes run at priority 16-63 processes

accumulated interrupt time for priority 16-63 processes

accumulated wait time for priority 16-63 processes

The next data structure change will occur in the KTB. First, a longword field, KTB$AR_SCHED_DATA, will be added. This field will be a pointer to an array indexed by CPU id. Below is the format of the array:

accumulated run time on CPU 0

# of times run on CPU 0

accumulated interrupt time on CPU 0

accumulated wait time on CPU 0

.....

.....

accumulated run time on CPU n

# of times run on CPU n

accumulated interrupt time on CPU n

accumulated wait time on CPU n

The symbol definitions for this array will be defined in $PCBDEF. Besides the pointer to this array, the KTB will have the following fields added to it in support of tracking the # of voluntary waits the kernel thread has done over the time period specified by SCH$GL_AFFINITY_TIME:

KTB$L_VOL_WAITS	Total # of voluntary waits the kernel thread has done over a specified time period
KTB$L_CURR_VOL_WAITS	Current count of voluntary waits
KTB$L_QEND_COUNT	# of quantum ends this process has incurred since the last time period

In addition to these 3 new fields above, the KTB will also need to contain 4 new fields to record the starting times for run, wait, and interrupt. Below is the name of each field and it's description:

KTB$Q_COMQ_WAIT	Contains the SCC at the time the kernel thread was placed on the COM queue
KTB$L_COMQ_WAIT_SOFT	Contains the absolute time the kernel thread was placed on the COM queue (measured in soft ticks)
KTB$Q_START_RUNT	Contains the SCC at the time the kernel thread was placed into execution
KTB$Q_START_INTERRUPT	Contains the number of hard ticks stored in the 5th entry of the CPU$Q_KERNEL array (kernel mode in system context) when the kernel thread is placed into execution

Finally, the number of times a process/kernel thread has had soft affinity broken will be an interesting data point that should be kept track of. Therefore, the cell, KTB$L_SOFT_BROKEN, will be added for this purpose. The data will be kept for the life of the process/kernel thread.

1.3.2 Source Code Changes

There are a fair number of source code changes that need to be made and in code paths that are highly performance sensitive. Efficient code must be written in order to impact the performance of these code paths the least. Performance analysis of these code paths should be done after implementation.

1.3.3 Per-Process and Per-CPU data arrays

As mentioned in the previous section, there are 2 data arrays that will be located via the per-CPU database ($CPUDEF) and the kernel thread block ($KTBDEF) data structures. The array located via the per-CPU database will be allocated as part of the CPU database allocation. So nothing special needs to be done other than modifying $CPUDEF.SDL. The array located via, KTB$AR_SCHED_DATA, will be allocated during process creation in the $SYSCREPRC system service if soft affinity is enabled for all processes and in the $CREATE_KERNEL_THREAD system service if the kernel thread is to have soft affinity enabled. However, a process/kernel thread may request soft affinity be enabled after it has already been activated. In this case, the

system routine SCH$ACQUIRE_IMPLICIT_AFFINITY, will allocated this array and place its address in KTB$AR_SCHED_DATA. On the other end, when a process/kernel thread is being deleted, this array must be deallocated at the same time that the PCB/KTB is being deallocated. For kernel threads other than the initial kernel thread, this occurs in the system routine, EXE$KTHREAD_DELETE_CALLBACK. For the initial kernel thread, this occurs as part of the $DELPRC system service, specifically, in the system routine, DELETE_IN_SYS_CONTEXT.

1.3.4 System Parameters and System Data Cells

The existing system parameter, AFFINITY_TIME, will remain (it currently has no use). It will contain the time period (specified in seconds) in which to count the number of voluntary waits for the soft affinity process. The default value for this parameter will be 3 seconds. Internally, this parameter will be converted to the # of quantum ends which can occur in this specified time interval. This internal value will be stored in the system data cell, SCH$GL_QENDS_FOR_VOLWAITS.

In addition to the data cell mentioned above, several new cells will be added in support of this design and they are:

- SCH$GL_IO_VS_COMPUTE, will be created to hold the value which determines if a kernel thread is compute bound or I/O bound. This value will be calculated as:

  1/2 * (quantum/iota).

- SCH$GQ_CC_PER_MS will contain the number of system cycles per millisecond. The cycle count in the HWRPB is per second. To determine the number of cycles per millisecond, the field HWRPB$IQ_CYCLE_COUNT_FREQ will be divided by 1000 and the result stored in this new field.

- SCH$GQ_CC_PER_HARDTICK will contain the number of system cycles per 1 hard tick. This number will be calculated via the equation:

  EXE$GL_TICKWIDTH * 10,000 = # of hard ticks per 1 millisecond

  SCH$GQ_CC_PER_MS/ (EXE$GL_TICKWIDTH * 10,000) = # of cycles per 1 hard tick

  The interrupt time accumulated by the process is measured in hard ticks. In order to subtract out the interrupt time from the run time (which is measured in cycles), the units of measure ment must be the same. Therefore, this value in this cell will be used as a multiplier to convert the interrupt time to a value measured in system cycles before subtracting it from the accu mulated run time.

- SCH$GQ_CC_PER_QUANTUM will contain the number of cycles it takes to reach quantum end. This value is simply SCH$GQ_CC_PER_MS * the value of quantum.

These data cells will be initialized in the scheduler's initialization routine, SCHED_INIT, in module SCHED_ROUTINES.MAR. However, the two system parameters, quantum and IOTA, which are used in the above calculations of three of the data cells are dynamic, meaning that either value could change at any time. Therefore these three system data cells (SCH$GL_QENDS_FOR_VOLWAITS, SCH$GL_IO_VS_COMPUTE, and SCH$GQ_CC_PER_QUANTUM) will be recalculated when a WRITE ACTIVE command from

SYSGEN or a PARAMETER WRITE ACTIVE command from SYSMAN is issued. The SYSGEN routine, WRTACT in module SYSGEN.MAR and the SYSMAN routine, WRITE_ACTIVE in module SYSPARM.B32, will call a new system routine, EXE$SYSPARAM_UPDATE, which will recalculate these system data cells.

The system parameter, AFFINITY_SKIP, has no purpose and therefore is obsolete.

1.3.5 Voluntary Wait Code

In OpenVMS, a kernel thread enters a voluntary wait by either invoking the system routine, SCH$WAIT or by invoking the pagefault routine, MMG$PGFLTWAIT. In both these routines, the system parameter, IOTA, is charged against the process for incurring a voluntary wait. In order to calculate how often a kernel thread is voluntarily giving up the CPU, the code in these 2 system routines will now increment the field, KTB$L_CURR_VOL_WAITS for soft affinity processes.

Since the time period for collecting the voluntary wait information is measured internally in the # of quantum ends, the following code will be added to the system routine, SCH$QEND in the RSE module:

1. If the current process has soft affinity, then proceed with the following steps. Otherwise, skip to step 5.
2. Increment KTB$L_QEND_COUNT.
3. Compare the value in KTB$L_QEND_COUNT with the value in SCH$GL_AFFINITY_TIME (# of quantums to count voluntary waits)
4. If the KTB count is equal to SCH$GL_AFFINITY_TIME, then
    a. Copy KTB$L_CURR_VOL_WAITS to KTB$L_VOL_WAITS
    b. Initialize KTB$L_CURR_VOL_WAITS to zero
    c. Initialize KTB$L_QEND_COUNT to zero
5. Continue on with quantum end processing

1.3.6 Scheduling a Process to Run

The data collection portion of the soft affinity design takes place in several system routines, namely, SWAP_PROCESS and ENTER_SYSTEM_CONTEXT in module SCHEDULER.M64, SCH$STATE_TO_COM in module SCHED_ROUTINES.MAR, and SCH$CHSEP in module RSE.MAR. The first two routines are responsible for taking the current process out of execution. For the old process (the one being taken out of execution) , these routines must update the appropriate CPU's scheduling data array. This update includes adding in the amount of run time and interrupt time this process accumulated while current and incrementing the number of processes that have run on this CPU at this process' priority. Additionally, if soft affinity is enabled for the process, the run and interrupt times must be added into the scheduling array pointed to by KTB$AR_SCHED_DATA. In the SWAP_PROCESS routine, the code will place a new process into execution. For this process, the code will calculate the process' wait time and add this value into the appropriate accumulated wait time cell in the CPU's scheduling array. If the process has

soft affinity enabled, then this value will also be added into the process' accumulated wait time cell for the current CPU.

The starting wait time will be recorded in the SCH$CHSEP and SCH$STATE_TO_COM routines which place a process on a COM queue. The starting wait time will be stored in the KTB fields, KTB$Q_COMQ_WAIT (to contain current value of SCC), and KTB$L_COMQ_WAIT_SOFT (to contain current value of EXE$GL_ABSTIM_TICS).

During the investigation phase of this project, two issues were uncovered which relate to the data collection being described in this section and in the following section. The first issue concerns the use of the SCC as the mechanism to keep track of how much time the process consumes either waiting. Since the SCC is specific to a CPU, a problem arises when attempting to calculate the amount of wait time a process has accumulated if the process is placed into a wait state on one CPU and subsequently brought into execution on a different CPU. The values of the SCC do not match. When this situation arises, the solution is to compute the number of soft ticks spent executing and then convert this value into the number of cycles before adding it to the accumulated wait time. In the KTB, a new field, KTB$L_COMQ_WAIT_SOFT, will record the value of EXE$GL_ABSTIM_TICS when the process/kernel thread is placed on a COM queue. The granularity of EXE$GL_ABSTIM_TICS is a soft tick (10ms). So to calculate how much time a process has waited, the following calculation will be done: EXE$GL_ABSTIM_TICS - KTB$L_COMQ_WAIT_SOFT = result A. Result A will then be multiplied by SCH$GQ_CC_PER_MS to yield the number of system cycles spent waiting.

The second issue is also related to this data collection. What happens when one of the fields,ie the accumulated runtime field, for instance, overflows? Since the algorithm uses floating point divides for its arithmetic, the high bit, bit 64, is interpreted as a sign value. Hence, the data length is really only 63 bits. The algorithm must detect data overflow in these fields. The overflow is detected when bit 64 becomes set. When an overflow is about to occur (in this case for the accumulated run time), the following steps are taken:

1. Add in the # of cyles the process has executed for to the appropriate priority based runtime field in the per-CPU data base array. Is bit 64 set, ie is the number negative? If no, continue on.

2. If yes, the number is negative. Then an overflow has occurred. Calculate the average run time for this priority process. Store the average runtime in the accumulated run time field. Do the same for the accumulated wait time. Then set the # of processes which have run at this priority to 1.

3. Now the overflow situation has been resolved and the accumulated times can now increment in the normal fashion.

1.3.7 Process Priority Changes

Whenever the current process' priority is about to change, an update to the array referenced by CPU$Q_SCHED_DATA needs to be made since this array is based on the process' priority. Luckily, all process priority changes are funneled through the system routine, SCH$CHANGE_CUR_PRIORITY. This routine should perform the following steps to ensure that the data recorded in the CPU's scheduling data array is correct:

1. Read the SCC (to determine the current time).

2. Calculate the amount of time that the process has run (the current time minus the starting time (from the scheduling array)).
3. Add this value to the appropriate accumulated runtime quadword using the process' priority before it has been adjusted. If soft affinity is enabled, then add this value into the appropriate entry in the array pointed to by KTB$AR_SCHED_DATA. Check for data overflows to ensure that the accumulated times do not go negative.
4. Record the current time as the starting run time for the process in the KTB field, KTB$Q_START_RUNT.
5. Perform steps 2 - 5 for the process's interrupt time.

The $SETPRI system service allows one to change the priority of a process/kernel thread which is not current, ie it's on a wait queue. The only interesting case here is if the process/kernel thread is on a COM queue prior to the $SETPRI. The effect of the $SETPRI will move the process/kernel thread from one COM queue to another. In this case, the values of KTB$Q_COMQ_WAIT and KTB$Q_COMQ_WAIT_SOFT will be reset to indicate a new starting wait time.

1.3.8 Idle loop

One of the design issues brought to light in the investigation report was: How is the scheduling data affected by an idle CPU? If the CPU remains idle for a significant period of time, the data in the CPU's scheduling array becomes stale. As the CPU remains idle, a way is needed to "age" the data, ie show that the CPU has "increasing good" run time. The solution chosen is that for each pass through the idle loop, a full quantum will be given to one of the run-time fields in the CPU scheduling array (starting with priority 0 processes) and the number of processes run for that priority level will be incremented by 1. The goal is to show that the CPU can handle a bigger load. Each time through the idle loop the next higher priority level in the CPU scheduling array will have its run time and # of processes adjusted accordingly. The wait time will not be adjusted.

1.3.9 Soft affinity system service

One can enable soft affinity for all subsequently created processes by invoking the $SET_IMPLICIT_AFFINITY system service and specifying the state argument as an OR of the two flags, CAP$M_IMPLICIT_AFFINITY_SET and CAP$M_DEFAULT_ONLY. In this case, the service will not have to do anymore work. At process creation time, the necessary data structures will be allocated to the process and all subsequent kernel threads in which to collect the soft affinity data. On the other hand, if one invokes the system service to enable soft affinity for an existing kernel thread, then this service will need to allocate the necessary pool for the scheduling data array and write the address of this data structure into the field, KTB$AR_SCHED_DATA.

The service also allows the user to specify which CPU the process should initially have soft affinity for. If no CPU is specified, the service will choose a CPU at random by calling the system routine, SCH$SELECT_IMPLICIT_CPU. Rather than selecting the CPU at random, select the CPU with the best run/wait ratio.

1.3.10 DCL

Today, if a user wants to set hard affinity for a CPU such as CPU 1, he/she would type:

$ SET PROCESS/AFF/SET=1/PERM

However, there is currently no equivalent for soft affinity. This design proposes a command of the following format:

$ SET PROCESS/AFF=SOFT/SET=1 or $ SET PROCESS/AFF=SOFT

The first command would enable soft affinity for the initial kernel thread and set affinity to CPU 1. Whereas the second command would enable soft affinity but let the system decide which CPU the process should "affinitize" to. If a user types:

$SET PROCESS/AFF

the default for the /AFF qualifier would be to set hard affinity. The preserves the current behaviour as mentioned in the first sentence of this section.

For the Raven release, there is now a /ALL qualifier for the $SET PROCESS command. This qualifier when used in conjunction with SET PROCESS/PRIORITY will change the priority of all kernel threads in the process. This qualifier will now work with SET PROCESS/AFFINITY as well.

Besides the SET PROCESS command, the SHOW PROCESS command will also be enhanced. If the process has hard or soft affinity, the SHOW PROCESS/ALL command will display this information as well as which CPU(s) the process is bound to.

The files which would require a change to support this work are: [CLIUTL]SETPROCES.B32 ,SHOWPROC.B32, and [SYS]SYS_AFFINITY.C as well as [CLD]SET.CLD and SHOW.CLD.

1.4 Resolution of Design Issues

The investigation report revealed several design issues that I believe have been addressed in the document. Below is the list of these issues as well as the solution/answer to them.

1. Can the use of the OTS$DIV routines be reduced/optimized/inlined to reduce the overhead of the soft affinity algorithm? Are there other ways to make the soft affinity algorithm as efficient as possible?

A: The solution was to remove the integer divides via OTS$DIV routines and instead use floating point divide instructions. This has shown to reduce the soft affinity overhead significantly when measured with the prototype code.

2. Using the equation, (run - interrupt times)/wait time, it was observed that the (run-interrupt) value may be smaller than the wait time resulting in a zero value on both CPUs.

   A: As it turns out, the solution to the previous issue has also eliminated this problem. The floating point values give us more than an adequate granularity for the result since the fractional part of the number can be represented in this format.

3. When recording the wait time in the per-CPU database, a problem was uncovered due to the fact that the system cycle counter is per-CPU and may contain different values on different CPUs. As a result, if the starting wait time is recorded on CPU A and CPU B places the process into execution, the wait time can not be added in. In fact, to keep everything in bal-

ance, the current solution is to not record the wait time, run time and interrupt time for the process for one pass through the CPU. Another possible solution is to record the system cycle counter and the current value of soft ticks for the starting wait time. Then when it is time to determine the wait time, if the system cycle counter can not be used, the # of soft ticks which occurred while the process was waiting could be calculated and converted into the # of cycles?

A: Both solutions mentioned above were considered. The first one, ie. using the prototype code and if the process' starting wait time was recorded on 1 CPU and placed into execution on another CPU, then the wait time and subsequent run and interrupt times would be ignored. The concern with this solution is that a fair amount of data would be lost. If most processes had hard or soft affinity enabled, then this solution might be acceptable. Since in this case, the process would most likely be rescheduled on the same CPU as it previously ran on. However with round robin scheduling and a large number of CPUs, the potential for data loss becomes much greater. The second solution seems to be the best choice. It eliminates the data loss potential but at the same time, the tradeoff is that the granularity isn't as fine when measuring in soft ticks. It does appear to be the best solution to this problem. Another solution considered was the idea of determining the difference (delta # of cycles) between the SCC on each CPU. To elaborate further, the idea was that when a CPU was started, it would calculate the delta value between it's current SCC value and the primary's SCC value and save this delta value. This delta value would be used when trying to make a comparision between the SCC values from two different SCCs. My decision is to go with the soft tick value if the wait times are from 2 different CPUs.

4. Since the data in the per-CPU database is based on the priority of the process, what happens if the process' priority changes while the process is executing?

   A: Whenever a process' priority changes, the scheduling data kept in the CPU scheduling data array must be updated so that it remains correct for each priority level.

5. What happens when a CPU goes idle? How does idle time affect the run/wait ratio? One suggestion is that the amount of time in the idle loop should be taken away from the cumulative wait time stored in the per-CPU database so that the run/wait ratio will increase.

   A: See section 1.3.8 for the solution to this problem.

6. Currently, the prototype code uses a bias value of 20%. Is this reasonable?

   A: It's hard to decide what is a reasonable value. Once more testing/performance analysis is done, a better choice can be made. 20% does seem rather high. Initially, the system parameter, AFFINITY_SKIP, will be used to set the bias value. This will allow us to collect data with different settings of the bias to determine its final value.

1.5 Impact Statements

1.5.1 Performance Impact

The performance of soft affinity will need to be measured carefully. Hopefully, someone from the performance group will be assigned to this project. It is extremely important that we do not de-

grade system and process performance noticably. The soft affinity algorithm will take longer to execute than the round-robin algorithm. How this affects the system will need to be studied.

The TPC benchmarks (or the appropriate subset) should be run with soft affinity enabled. If these benchmarks already enable hard affinity, then a performance improvement when using soft affinity may not be seen. However, under no circumstances should soft affinity degrade these benchmarks.

1.5.2 Verification Impact

Since this feature is a significant change to the system, the verification team will need to run their tests with soft affinity enabled. This will give this feature more exposure to a greater number of environments.

1.5.3 Documentation Impact

The new features manual for Raven should include a description of soft affinity. The system parameter, AFFINITY_TIME, should be documented in the appropriate manual. The DCL manual will need to be updated for the SET and SHOW PROCESS commands.

1.6 Revision History

Version	Modified by	Date	Comments
X1.0	Ellen M. Batbouta	15-Apr-1997	Initial Draft
V2.0	Ellen M. Batbouta	28-Apr-1997	Incorporate review comments