WSCLOCK – A Simple and Effective Algorithm for Virtual Memory Management

Richard W. Carr[1]
Department of Computer Science
Stanford University

John L. Hennessy
Computer Systems Laboratory
Stanford University

Abstract

A new virtual memory management algorithm WSCLOCK has been synthesized from the local working set (WS) algorithm, the global CLOCK algorithm, and a new load control mechanism for auxiliary memory access. The new algorithm combines the most useful feature of WS – a natural and effective load control that prevents thrashing – with the simplicity and efficiency of CLOCK. Studies are presented to show that the performance of WS and WSCLOCK are equivalent, even if the savings in overhead are ignored.

Introduction

Modern memory management policies optimize performance by varying the space allocated to each task as its perceived need changes. Such policies also vary the load (i.e., the number of active tasks) to achieve high levels of multiprogramming while avoiding thrashing. Modern variable-space, variable-load memory management policies have been divided into *local* policies and *global* policies. Ideally, a local policy estimates the memory needs, or *locality*, of each task independently of other tasks and allocates sufficient main memory to hold the each active task's locality. A global policy correlates a task's memory allocation with its locality, but makes no explicit, independent measure of the locality, and does not necessarily allocate sufficient main memory for each active task's locality.

Local policies are typified by the working set (WS) policy which was first defined by Denning [DENN68] and has been the object of much study [DENN72, RODR73, PRIE73, SMIT76, MARS79]. Global policies are typified by the global least-recently-used (LRU) approximation algorithm CLOCK that is used in MULTICS; studies of CLOCK have appeared infrequently [CORB68, EAST76]. Although a local policy, such as WS, isolates tasks from each other and may be better at preventing thrashing, a global policy is often used in real systems because it is simpler to implement and has less computational overhead.

This paper presents a new policy WSCLOCK that combines the operational advantages of WS with the simplicity and efficiency of CLOCK. We describe the data structures, replacement algorithm and load control used to implement WSCLOCK. We introduce a simple new *Loading Task/Running Task (LT/RT)* load control mechanism to control competiton for access to auxiliary memory; although *LT/RT* is a general control for all memory management policies, it is shown to be particularly appropriate for WSCLOCK. Finally, we describe the use of a realistic simulation model to demonstrate the effectiveness of both the *LT/RT* control and of WSCLOCK.

Our primary focus in this paper is virtual memory management methods in interactive systems. Interactive systems are characterized by large numbers of tasks which make numerous small processing requests; in such systems, the amount of memory used by all tasks can be many times the size of main memory and, thus, is where one finds the greatest advantage of virtual memory. Compared to batch systems, interactive systems activate and deactivate tasks very frequently and perform the basic memory management functions more often; these systems benefit the most from algorithmic simplicity and efficiency. Although the methods presented in this paper may be most effective in interactive systems, they do not appear to have any disadvantages when used in batch systems.

This work was supported by the Department of Energy, Contract DE-AC03-76SF00515.

[1] Author's current address: Tandem Computers Inc., 19333 Vallco Parkway, Cupertino, CA 95014

General Model of a Virtual Memory Computer System

This section is a brief summary of a virtual memory computer system model presented in [CARR81]. The model is designed specifically to compare scheduling, memory management, and load control policies on conventional large-scale computers. The model incorporates both an accurate representation of program behavior based on measured program reference strings, and a general, but detailed, model of a virtual memory operating system.

Task Model

A task is modeled by a *virtual address space* $P = \{ p_i \mid i = 1,2,...,m\}$ of *pages* and a *reference string* $\{r_t \mid t = 1,2,...,T\}$. Each *reference* r_t is an ordered pair (p,d), where $p \in P$ and d is Boolean variable which is *true* if the reference changes or *dirties* the page. The *virtual time VT* of a task is the number of references that have been completed for that task. Tasks make *I/O requests* at stochastically-distributed intervals of virtual time.

Configuration Model

The computer system configuration model contains (1) a *central processor*, (2) a *main memory* of *M page frames*, and (3) a collection of *I/O devices*. The central processor is implicity defined as capable of executing one reference for one task in each unit of virtual time. Associated with each frame are the *use-bit*, set when the frame is referenced, and the *dirty-bit*, set by each dirty reference. I/O devices are modeled as simple servers with independent and indentically distributed service times.

One or more of the I/O devices is designated as an *auxiliary memory*, which contains a copy of every task page. Although the model permits both task and paging I/O requests to access the same device, the studies presented in this paper assume that paging devices are separated from task I/O devices.

Operating System Model

The operating system model has three main components: the *scheduler*, the *memory manager*, and the *load control*.

Each task occupies one of two scheduler queues (the ready queue and the active queue) or is dormant. The scheduler orders the ready queue and assigns a time-slice to each ready task in order to balance objectives of response time, processor utilization, throughput and externally specified priorities. The model uses a multi-level load-balancing queue, described in [CARR81], which can be parameterized to operate as any of the simpler queueing disciplines, such as first-in-first-out or round-robin. Tasks in the active queue are selected for execution in round-robin order for short time quanta to approximate a processor sharing discipline. A task remains in the active queue until (1) the time-slice is exhausted, (2) the task is deactivated by the load control, or (3) the task completes or becomes dormant.

The memory manager allocates main memory frames to each task, and requests paging I/O operations to copy pages between main and auxiliary memory. At any given time, each task address space P is partitioned into a *resident set* R, of pages which occupy main memory frames, and a *missing set* R'. (R' denotes the complement of the set R.) If the processor executes a reference (p,d) and $p \notin R$, a *page fault* blocks the task until the missing page is made resident by copying it from auxiliary memory. The model assumes *demand-paging*: a page p is loaded only after a page fault occurs for p. A $p \in R$ is *clean* whenever it is copied from main memory to auxiliary memory, or vice-versa; p is *dirty* following any reference which sets the frame dirty-bit.

To achieve optimal performance in a virtual memory computer, the operating system seeks to maximize the number of active tasks without inducing thrashing. Thrashing occurs when so many tasks are active that the sum of their memory needs exceeds the size of main memory and; memory becomes overcommitted. The load control monitors the commitment of main memory (either directly or indirectly) and when memory appears to be undercommitted, load control may move tasks from the ready queue to the active queue; when memory appears to be overcommitted, the load control moves tasks from the active queue to the ready queue.

Working Set Policy

The reader should be familiar with the basic concepts of the WS policy (see [DENN68] or [DENN70]). We limit our discussion to the relevant details of its implementation.

Working Set Determination

The WS policy defines the working set W to be all $p \in P$ referenced in the previous θ units of the task's virtual time. Implementation of WS load control and replacement requires some timely mechnism to determine each task's W. If a task's page table contains only $p \in W$, then the arrival of a new $p \in W$ is signalled by a page fault. To detect the departure of a page from W is more difficult. Typically, we require (1) a task's virtual time VT, (2) the last reference time $LR(p)$ for each page, and (3) a procedure to detect pages for which $VT - LR(p) \geq \theta$.

Task VT is easily obtained by summing the time intervals that the task has executed. Pure WS assumes some mechanism that can update $LR(p)$ automatically in parallel with program execution, but practical implementations use either the page frame use-bit or special hardware to approximate $LR(p)$. To use the frame use-bit, each $p \in R$ of a given task is examined by software at various times (e.g., at faults or at fixed intervals). The use-bit is tested and cleared, and if the page was recently referenced then $LR(p)$ is set to the task's current VT. To detect $p \in W$ and for which $VT - LR(p) \geq \theta$ implies a *WS-scan* of each $p \in P$ to find each $p \in W$. (To scan only $p \in W$ for a given task would require the maintenance of an additional data structure.) The WS-scan can incorporate the use-bit test to approximate $LR(p)$ with little extra cost.

With the hardware support for WS on the Maniac II (see [MORR72]) each page frame has an associated counter that approximates the virtual-time-since-last-reference $VT-LR(p)$ directly. The counter is cleared whenever the page is referenced and the processor automatically increments the counter of each page of the task every .25 msec. that the task executes. Since the Maniac II has only 64 page frames, the processor time is minimal, but with the larger memories on modern machines, the time required to update thousands of counters might be excessive. The Maniac II implementation still requires a WS-scan to remove the pages for which $VT-LR(p) \geq \theta$. In essence, the Maniac II scheme is equivalent to the ordinary use-bit method, except that the scanning is scheduled and performed without the overhead associated with a system interrupt.

Load Control and Replacement Algorithm

When a task is active, $W = |W|$ frames are committed to the task. The total memory commitment W_{active} is the sum of the W of all active tasks. The WS load control will activate a ready task unless the W of the first ready task exceeds $M - W_{active}$.

A set A of available frames is replenished whenever a task is deactivated or when a WS-scan removes resident pages from an active task's W. The replacement algorithm simply chooses some frame in A. If A is empty, then the load control selects a task to deactivate and that task's resident pages are placed in A.

Page Writing and Reclamation

When a dirty page is placed in A, it must be cleaned before it is replaced. A simple approach is to couple the writing of a dirty page and the reading of the page that replaces it; this method blocks a faulting task for the time of two paging I/Os instead of one. Another approach is to replace only the clean pages in A and to clean the dirty pages in A when there are no outstanding page read requests for a device; if all pages in A are dirty, then cleaning operations will naturally have precedence over page reads.

Although a page in A is eligible for replacement, it may not be replaced for some time. If a task references a page in A, the system can avoid the delay and a page-in I/O operation if it can *reclaim* the page in A. This requires a special procedure to search A each time a page fault occurs.

CLOCK Policy

CLOCK is a simple approximation of the global LRU replacement algorithm [CORB68]. All main memory page frames are ordered in a fixed circular list as illustrated in Figure 1. A pointer or "hand" always points to the last frame replaced. When a frame is needed to hold a missing page, the pointer is advanced "clockwise", scanning frames in circular order. The use-bit is tested and cleared: if the bit was set, the frame is recently-used and is not replaced; if the bit was clear, the frame is not-recently-used and is replaceable if the page is clean. If a replaceable page is dirty, then it is scheduled for cleaning and is not replaced. When the CLOCK-scan locates a clean and not-recently-used page, the algorithm halts, leaving the pointer pointing to the chosen frame to mark the starting point for the next scan. Note that a page is never removed from R until it is actually replaced.

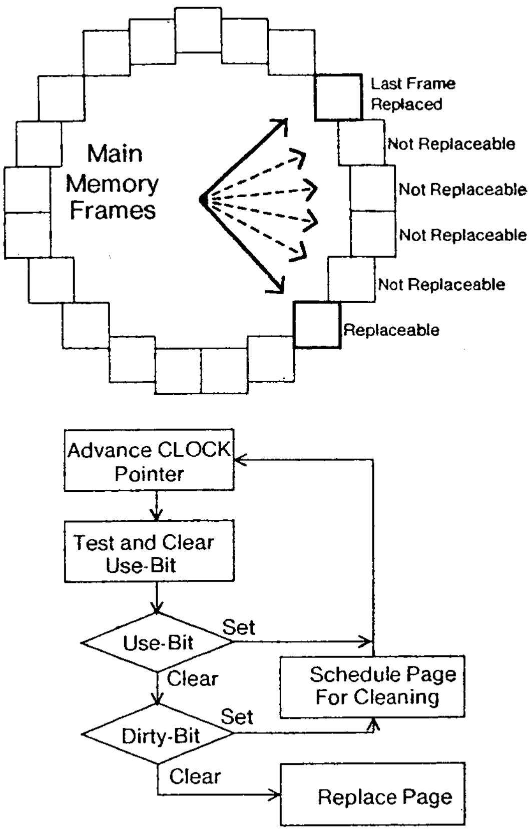

Figure 1. CLOCK Replacement Algorithm

Global policies, such as CLOCK, allow all active tasks to compete for main memory allocation. There is no mechanism to determine a task's memory needs independently of the other tasks. Thus, instead of a load control based on explicit estimates of the main memory committment, global policies typically require an adaptive feedback control mechanism. For example, the control may monitor the page fault rate (or auxiliary memory traffic) and adjust the multiprogramming level if the rate is too high or too low.

Comparing WS and CLOCK

The WS and CLOCK policies can be compared at many levels. At the implementation level, WS appears to be more complex than CLOCK. In particular, WS requires:

(1) scheduling and executing the WS-scan procedure,

(2) memory to store $LR(p)$ for every page of every task, and

(3) algorithms to maintain the set A of available pages, including a method to reclaim pages from A.

The need to store $LR(p)$ effectively doubles the size of the page tables. In a system where the total of all task virtual memory is many times the size of main memory, minimization of page tables is an important consideration.

The implementation of the CLOCK replacement algorithm is simpler and consumes less memory, but CLOCK requires an adaptive feedback load control mechanism that is heuristic and, compared to the WS load control, may be more difficult to tune.

At the policy level, WS appears to have the advantages of good task isolation, and a predictive load control. Under a global policy, a task's resident set depends on how actively it references its current locality relative to the other active tasks. Mathematical models of global replacement (see [SMIT80]) show that some tasks can monopolize main memory and force other tasks to execute slowly and inefficiently; global algorithms can also lead to thrashing [DENN70]. WS isolates tasks from each other and guarantees each active task can acquire sufficient main memory to hold its working set.

At the performance evaluation level, no conclusive comparisons of WS and CLOCK have been performed. Analytical models are too weak to characterize the differences between local and global memory management policies in general, or the WS and CLOCK policies in particular. Empirical and simulation studies have not addressed the problem with sufficient completeness. This work makes no claim that either WS or CLOCK is more effective than the other; it is entirely likely that a well-implemented version of either policy will have approximately the same performance if the overhead of computing the policy is eliminated. This widely-held conjecture is supported by studies in [CARR81]. The purpose of this paper is to present a policy which is as effective as both WS and CLOCK and avoids many of the implementation difficulties of each.

WSCLOCK

The WSCLOCK policy combines the best features of WS and CLOCK. It retains the thrashing-preventative load control and task isolation properties of WS, but it eliminates:

(1) the WS-scan,

(2) the space for $LR(p)$ for each task page,

(3) the available frame set A, and

(4) the page reclamation procedure.

The WSCLOCK replacement algorithm uses the simple mechanism found in CLOCK but does not require an adaptive feedback load control. WSCLOCK is simpler than either WS or CLOCK.

Data Structures

Main memory frames are arranged in a fixed circular CLOCK-like list. The CLOCK pointer identifies the last frame replaced in the previous CLOCK-scan. Instead of an $LR(p)$ for all $p \in P$, $LR(p)$ is defined only for the resident pages, $p \in R$, in a storage cell associated with each page frame.

When a page fault occurs, a page read request is placed on a *paging queue*. When an auxiliary memory device is available, a request for that device is removed and processed; at that time, the replacement algorithm is invoked to obtain a frame containing a clean replaceable page to hold the incoming page.

Replacement Algorithm

The WSCLOCK replacement algorithm uses the CLOCK scanning method to apply the WS replacement rule as shown in Figure 2.

To examine a frame, WSCLOCK tests and clears the frame use-bit. If the bit was set, $LR(p)$ is set to the owning task's VT. Otherwise, if $VT - LR(p) \geq \theta$ then the page is removed from W. A page p is replaceable if either (1) $p \notin W$, or (2) the owning task is not active. If a replaceable page is dirty, then it is scheduled for cleaning and not replaced. When WSCLOCK finds a clean replaceable page, it halts, and leaves the pointer at the chosen frame.

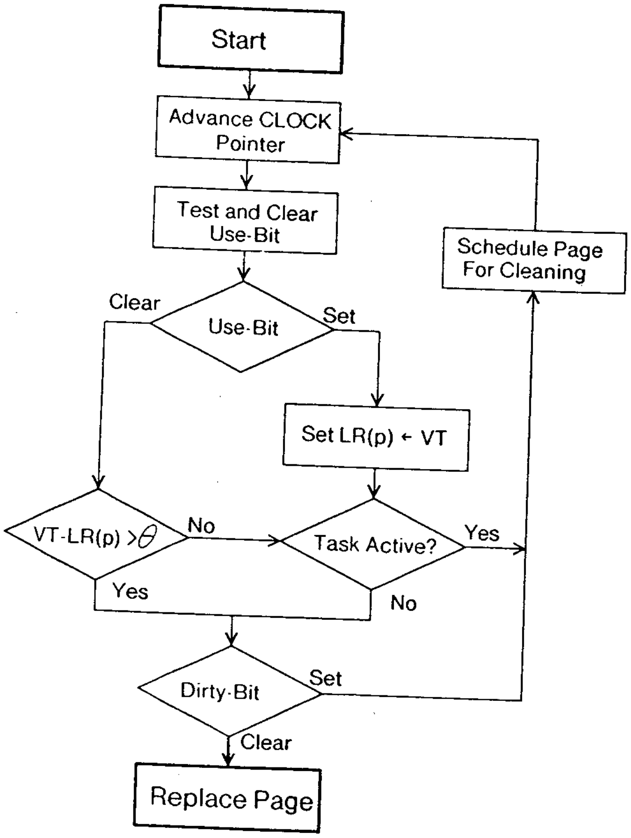

Figure 2. WSCLOCK Replacement Algorithm

WSCLOCK eliminates the available page set A because it simply searches for and finds a replaceable page when one is needed. Page reclamation is eliminated because a page is not removed from a task's R until it is selected for replacement. Pages to be cleaned are placed on the paging queue with the read requests. The queue can be ordered by time of request (FIFO) or by placing reads before writes.

In general, it is unnecessary to remove a page being cleaned from R. The dirty-bit is cleared when the I/O to write the page is initiated; if the page is updated subsequently (even during the I/O) the dirty-bit is reset and the page will be cleaned again before it is replaced. After a page is cleaned, it will be replaced on the next circuit of the CLOCK pointer unless it is referenced (and reenters W). Alternatively, a list of recently-cleaned pages can be maintained; the replacement algorithm takes a page from this list in preference to performing the CLOCK-scan.

Load Control

Since WSCLOCK scans only $p \in$ R, it does not approximate W if W contains some $p \notin$ R. WSCLOCK approximates only the *resident working set* RW = R $\cap$ W. WSCLOCK load control uses the same rules as WS load control, but using $RW = |\mathrm{RW}|$ as the memory commitment of each task. RW_{active} is defined to be the sum of the RW of the active tasks. When a task is deactivated, all of its pages are eligible for replacement and, thus, RW of a ready task is the value of RW when that task was deactivated.

WSCLOCK detects overcommitment when the CLOCK-scan fails to find a clean replaceable page in a full circuit of the frames. If there are any page cleaning requests on the paging queue, these are processed to produce a clean replaceable page. Otherwise, there are no replaceable pages, either clean or dirty, and some task must be deactivated to relieve overcommitment.

LT/RT Control

Introduction

When a task is activated, it usually enters a *loading* phase in which it has few $p \in$ R and must load missing pages to execute efficiently. When the task has loaded a sufficient resident set, it enters a *running* phase in which few page faults occur and the task executes efficiently. Typically, the virtual time of the loading phase is small, while the real time of the loading phase is disproportionately large because of the paging I/O delays.

If activations occur frequently, many loading tasks may contend for access to auxiliary memory. If we assume that each loading task has an equal opportunity to access auxiliary memory, the loading time is proportional to the number of concurrent loading tasks. Any increase in the number of loading tasks will increase the duration of each task's loading phase and, thus, will also increase the probability that the remaining running tasks will complete their time slices and be displaced by even more loading tasks. This tendency for an undesirable situation to become even worse is reminiscent of thrashing, but arises from an overcommitment of auxiliary memory rather than main memory. Note that this phenomenon may cause the auxiliary memory to be extremely busy even though main memory is undercommitted; this illustrates a weakness of load controls based auxiliary memory traffic, such as the 50% rule [DENN76].

To avoid periods of low processor utilization that occur when the active queue contains only loading tasks, we have devised a simple control that reduces the mean time that an active task remains in the loading phase and ensures a more consistent balance of loading and running tasks.

Implementation

The *loading task/running task* (*LT/RT*) control discriminates the two phases of task processing and limits the number of concurrent loading tasks. The primary *LT/RT* parameter is L, the maximum number of concurrently loading tasks. Typically, L will be determined empirically, but the optimal value is close to the number of paging devices which can be accessed simultaneously. In complex systems, it may be necessary to consider that paging requests may not be balanced across a set of paging devices.

The discrimination of loading tasks and running tasks is by a simple heuristic: a task is loading until it has executed for τ units of virtual time or has requested an I/O operation. We claim that this heuristic is robust if τ is chosen to be moderately larger than the loading phase of most tasks. Suppose that a particular task stops loading after τ' units of virtual time, where $\tau' \leq \tau$; although *LT/RT* will prevent a new activation for an additional $\tau - \tau'$ time units, the actual delay will be minimal because the pages required by the task are resident and the task will execute without page faults. A task that makes an I/O request is considered to be running because an I/O-bound task might prevent new task activations for an unreasonably long time.

LT/RT has three related effects. First, it reduces the mean delay between the time that main memory becomes available to activate a task and the time that the task enters the productive running phase. Second, main memory is used more effectively, since fewer page frames are committed to unproductive loading tasks. Finally, *LT/RT* improves processor utilization because it maintains a more consistent balance of loading tasks and running tasks.

A further improvement with *LT/RT* is possible: task deactivations for time-slice completion should be delayed until the memory made available by the deactivation can be used effectively. Thus, if L loading tasks are active, no time-slice deactivations should be processed. Tasks that have completed their time-slices should be deactivated only when there are fewer than L loading tasks *and* there is insufficient uncommitted memory to activate the first ready task. This policy might further ensures a good balance of loading and running tasks, but has not been incorporated in this study.

With *LT/RT* we can refine the paging queue strategy by processing page reads for running tasks before reads for loading tasks. This strategy should improve processor utilization directly by giving preferential service to those tasks that are executing efficiently. If the load control is operating properly, running tasks should have relatively few page faults. Furthermore, this strategy provides an additional load control for global policies: if memory becomes overcommitted, all tasks will begin to page fault; the paging queue strategy will process page faults for a subset of the active tasks and, in effect, lower the multiprogramming level until the running tasks cease to fault.

Note that the *LT/RT* control is independent of the WS and CLOCK load control mechanisms; *LT/RT* prevents the activation of too many loading tasks even when they will not overcommit main memory. Furthermore, when a task is first executed, the size of its locality is unknown until it has completed the loading phase. Thus, the *LT/RT* control can aid both WS and CLOCK load control by delaying the activation of additional ready tasks until the memory needs of the recently activated tasks can be measured. In [CARR81], we claim that *LT/RT* is a viable, if slightly suboptimal, load control for CLOCK in the absence of any other load control.

We recognize that a task may encounter additional loading phases if it has more than one locality and transitions among them. If such transitions were predictable or if their onset and duration could be reliably estimated, then the *LT/RT* control could also be applied to tasks in these transition loading phases. In this study, however, we elect to apply *LT/RT* only to the highly predictable loading phase that occurs when a task is activated.

Operation of WSCLOCK with *LT/RT* Control

Assume, for the moment, that the CLOCK-scan estimates the $LR(p)$ for each $p \in \mathrm{R}$ with reasonable accuracy. Then, the main dissimilarity between WS and WSCLOCK lies in the difference between W and RW and its use in the WS load control. When a task is activated, W (or RW) frames of memory are *committed* to the task. Memory frames are *allocated* only as the task executes and demands them by referencing them. If, at activation, a $p \in \mathrm{W} \cap \mathrm{R}'$ is not referenced for θ, the periodic WS-scan of WS will remove p from W, while WSCLOCK lacks a mechanism to perform this operation. WSCLOCK can remove inactive pages only if they are resident. Fortunately, the following result limits the inaccuracy of WSCLOCK to the first θ after each activation.

Claim: If a task has executed for at least θ units of virtual time since activation, **W** and **RW** will be identical.

Proof: (1) If $p \in \mathrm{W}$ then it has been referenced (and made resident) in the last θ. By the WS replacement rule, p can not have been replaced. Thus, $p \in \mathrm{W} \Rightarrow p \in \mathrm{R}$ and, thus, $p \in \mathrm{RW}$. (2) If $p \notin \mathrm{W}$ then $p \notin \mathrm{R} \cap \mathrm{W} = \mathrm{RW}$. By (1) and (2), $\mathrm{W} = \mathrm{RW}$.

The largest discrepency between W and RW occurs immediately after activation and decreases rapidly during the loading phase. If *LT/RT* discriminates loading tasks and running tasks properly, W and RW will be nearly equal when the loading phase ends. Since $\mathrm{RW} \subseteq \mathrm{W}$ during the loading phase, WSCLOCK underestimates the amount of memory that a loading task may require. This implies that WSCLOCK has a tendency to make additional activations and overcommit memory during this phase. Thus, the *LT/RT* control not only prevents overcommitment of auxiliary memory, but also prevents overcommitment of main memory by WSCLOCK.

Experimental Studies

Methodology

WS and WSCLOCK are compared using a discrete-event computer system simulation model. Due to space limitations, we provide only a summary description of the model; complete details, including a validation of the model, are found in [CARR81]. The major aspects of the model are:

- The workload model is a sequence of tasks chosen randomly from a set of prototype task models.
- Each task model is obtained by tracing a real program to produce a deterministic reference string and a count of I/O requests. The measurements are used to create the IRIM model of program behavior (see below) and a stochastic I/O request model. I/O requests are generated at exponentially distributed intervals with a mean interval equal to that of the measured program.
- The configuration model includes an explicit representation of each frame of main memory (including use- and dirty-bits) and of each page of virtual memory. I/O devices are modeled stochastically.
- The operating system model is a generalized, but highly detailed representation of a real operating system. Task scheduling, allocation of main memory, assignment of virtual memory pages to page frames, load control, and I/O request scheduling are all performed explicitly.
- Task execution is based on processing of the reference string, detecting page faults when a referenced page is missing, and setting of the use- and dirty-bits. Virtual time is calculated exactly as the number of references successfully completed.

The simulation does not model the effort required to execute the operating system (i.e., overhead). The purpose of the study is to compare the basic effectiveness of the WS and WSCLOCK. For example, there may be different methods of implementing the WS-scan or page reclamation algorithms, with different amounts of overhead for each implementation; we desire to minimize the possibility that observed differences in performance are due to extraneous implementation details.

The Inter-Reference Interval Model

The Inter-Reference Interval Model (IRIM) is a deterministic, trace-driven model of program behavior. It converts the program reference string to a more compact IRIM string, but it retains the identity of each task virtual memory page. Thus, it is particularly useful for the studies presented here because it permits the detailed simulation of page and frame management under alternative memory management policies. Due to space limitations, we provide a summary description of the IRIM. A full description of the IRIM, including (1) a formal definition, (2) methods to generate the IRIM string, (3) the use of IRIM strings to model program behavior in a multiprogram system model, (4) the validation of the IRIM and (5) measurements of simulation efficiency with the IRIM can be found in [CARR81].

At each moment of virtual time, the IRIM sorts each task page into one of three categories or states:

IDLE – in a interval of ω or more references in which no reference to the page occurs.

CLEAN – not IDLE (i.e., being referenced at least once every ω references) and is in a interval of ω or references in which no page updates occur.

DIRTY – not CLEAN or IDLE, i.e., is being updated at least once every ω references.

The IRIM parameter ω is analagous to the WS parameter θ except that the IRIM state transitions from CLEAN or DIRTY to IDLE occur at the beginning of any ω interval in which the page is not referenced (similarly for the transition from DIRTY to CLEAN). Thus the IRIM can be more closely compared to VMIN [PRIE76] which incorporates predictive information that a page will be unreferenced for some future interval.

The IRIM models program behavior both accurately and efficiently under many memory management policies, including the purely theoretical WS, practical approximate WS policies (including WSCLOCK), and global policies such as CLOCK or global LRU. The IRIM is highly suitable for simulating lookahead policies such as VMIN. The IRIM is not appropriate for simulation of policies, such as FIFO and RAND, that make little or no effort to detect program locality.

The IRIM is validated by comparing full system simulations using both ordinary reference strings and IRIM strings. Validation tests in [CARR81] showed extremely close agreement (less than 1% error) in both the long-term and the short-term behavior of the system.

Simulation using ordinary reference strings is extremely expensive, running from 10 to 40 times slower than real time. With $\omega = 5000$, the IRIM reduces the length of the program reference string by a factor of approximately 600 and the simulation runs about 10 times faster than real time. The IRIM is essential in making the studies described in this paper practical.

Model Configurations and Workloads

The study used three model system configurations that vary the relative main memory size and auxiliary memory access time.

Table 1 – System Configurations

	Configuration		
	A	B	C
Main Memory Frames	200	250	350
Auxiliary Memory Access Time (min-max)	20-40	30-50	50-80

The page size is 1024 (32-bit) words. Auxiliary memory access times are in units of 1000 references, and are uniformly distributed between the minimum and maximum values. We assumed a processor capable of executing 4×10^6 references per second (equivalent to a 2–2.5 MIPS processor). Configuration B is a typical system with a 1 megabyte main memory and a mean auxiliary memory access time of 10 msec. Configuration A has less memory and a faster auxiliary memory, while Configuration C is memory-rich but has a slower auxiliary memory. These configurations were selected because they all resulted in a performance utilization of about 60% and seem to be realistic. To compare memory management policies we desire a balanced system that is neither processor-bound nor paging-bound.

Task models are derived from measurements of 8 commonly used programs such as the Fortran and Pascal compilers, text-formatting and sorting utilities, etc., generating about 5,000,000 memory references for each. Each simulation is run until a total of 50 tasks complete, which is a simulation of over 270,000,000 memory references. To eliminate the largest source of variation between simulation runs, the sequence of tasks is identical in each run. Since the typical multiprogramming level is 5, this run length ensures that many combinations of the 8 programs are processed concurrently at different times. Although the simulation begins in an empty memory state and encounters many faults in the initial stages, measurements show that the startup transient, which lasts for less than 20,000,000 references, has a negligible effect on the results.

Model Policies

For each configuration, the system is simulated using three memory management policies:

1. Pure WS. – The theoretical WS policy is simulated precisely. $LR(p)$ is recorded each time a page p is referenced. A page is removed from W when $VT - LR(p) = \theta$. The simulator implements the WS load control described above.

   We note that this model differs from analytical models in which a page that is removed from W is also removed from R. We assume that the page remains in R until it is replaced. A reference to a page in R – W simply places that page in W and does not require an I/O transfer. Models without this capability will significantly underestimate system performance.

2. Pure WS with *LT/RT*. – Pure WS is modified to control the number of simultaneously loading tasks.

3. WSCLOCK. – The new policy described above uses (1) the CLOCK replacement algorithm modified to implement a WS replacement rule, (2) the *LT/RT* control, and (3) the WS load control based on the resident working set.

In addition to the basic WS tuning parameter θ, the parameters and policies described below were studied and tuned to their optimal values for the particular memory management policy. In each case, the optimal choice was the same for all three policies.

1. Task deactivation policy. When overcommitment is detected the load control chooses a task to deactivate. We considered six different deactivation policies and discovered that optimal performance is achieved by deactivating one of:

   (1) the task with the smallest resident set,

   (2) the last task activated, or

   (3) the task with the largest remaining time-slice.

   Poorer performance results if

   (1) the faulting task,

   (2) the task with the largest resident set, or

   (3) a random task

   is deactivated. In the studies described below, we used the policy of deactivating the last task activated.

2. *LT/RT* parameters. With one paging device, L=1 gives best performance. With two paging devices (and balanced requests to each), L=2 gives slightly better performance than L=1 (and much better than L>2). Performance improves steadily as τ is increased from 0 to 15,000 references. Between 15,000 and 100,000 there is little change, which supports the claim for robustness. This study used $\tau = 15{,}000$.

3. Paging Queue Order. Scheduling page reads before writes is clearly better than FIFO. Scheduling page reads for running tasks before reads for loading tasks also improves performance by a small factor. This study used the latter policy.

4. Free Page Pool. Denning suggests the use of a parameter K_0 that is the desired minimum number of uncommitted pages [DENN80]. If W is the working set size of the first ready task, it is activated only if $W + K_0 \leq M - W_{\text{active}}$. For the workloads and configurations studied, $K_0 = 0$ achieved maximum performance and is used in this study.

Measurements

Each combination of configuration and memory management policy was simulated for a range of values of the WS parameter θ. The basic measure of performance is processor utilization, which is the ratio of successfully executed references (i.e., virtual time) to total simulated real time. The results of these simulations are displayed in Figures 3, 4, and 5.

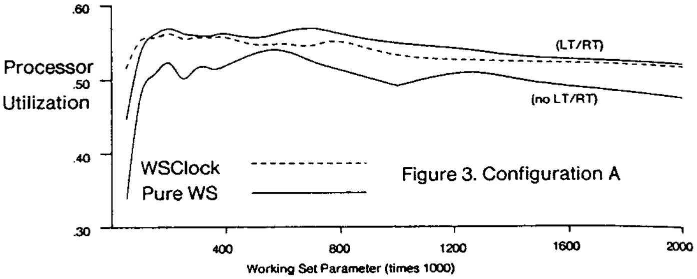

Figure 3. Configuration A

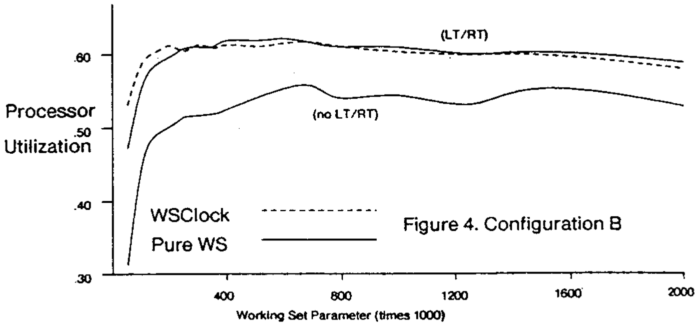

Figure 4. Configuration B

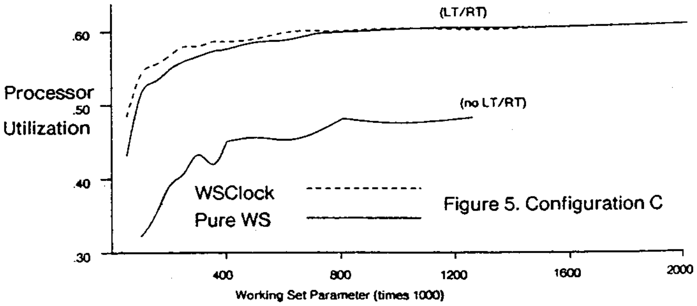

Figure 5. Configuration C

The usefulness of the *LT/RT* control for pure WS is evident, increasing processor utilization by 5 to 20%. The greatest improvement is for the large main memory/slow auxiliary memory configuration C. With this configuration, main memory is often underutilized because tasks cannot be loaded rapidly enough. *LT/RT* prevents overcommitment of auxiliary memory by loading tasks, and it increases performance by maintaining a orderly flow of running tasks that can be executed efficiently. With the current technological trends, the size of main memory is increasing faster than the speed of auxiliary memory; thus, the studies show that *LT/RT* is becoming more useful.

The performance of pure WS (with *LT/RT*) and WSCLOCK are very similar. On Configurations B and C, they are practically identical. On Configuration A (small main memory/fast auxiliary memory) WS outperforms WSCLOCK by a small margin. Table 2 gives the significant performance measures for the peak performance for each configuration/policy combination.

Table 2 – Peak Performance

	Configuration					
	A		B		C	
	WS	WSCLOCK	WS	WSCLOCK	WS	WSCLOCK
θ (x1000)	200	200	600	700	2000	2000
Utilization	.572	.566	.621	.616	.613	.613
Mean MPL	4.86	4.39	5.25	5.01	5.53	5.48
Load Control Deactivations	137	76	77	46	11	11
Page Faults	7840	6300	5437	4980	3970	3940
Pointer Travel/ Replacement		12.6		13.2		12.4
WS-Scans	3136	0	1280	0	454	0

The difference between peak performance ranges from 0 to 0.8%. The close agreement between WS and WSCLOCK extends to the value of θ that optimizes performance for each configuration. At comparable values of θ, WS had higher levels of multiprogramming, higher page faults rates, and larger numbers of deactivations.

Under WSCLOCK, the mean number of frames examined before finding a replaceable page is nearly constant, even though the configurations have a ratio of memory sizes of almost 2:1. Compared to the cost of performing a paging I/O, the WSCLOCK cost of examining an average of 13 frames for each page fault is insignificant. WSCLOCK reduces overhead by eliminating the WS-scan operations and, for some reason that is not at all clear at this time, by reducing the number of page faults.

Conclusions

WSCLOCK, a new algorithm for virtual memory management, has been presented. WSCLOCK employs the key notions of the working set policy: task isolation and a local replacement strategy, combined with the simpler implementation technique used in the global algorithm CLOCK. The resulting algorithm has performance properties comparable to WS.

The extensive and realistic simulation of the WS and WSCLOCK algorithms demonstrate that WSCLOCK presents a practical algorithm for implementing the working set concepts. We conclude that its use in a real operating system has significant advantages over existing implementations of either WS or global memory management algorithms.

Bibliography

[CARR81] R.W. Carr, *Virtual Memory Management*, Ph.D. Dissertation, Stanford University, August 1981 [Available from Computation Research Group, Stanford Linear Accelerator Center, Stanford CA]

[CORB68] F.J. CORBATO, *A Paging Experiment with the Multics System*, MIT Project MAC Report MAC-M-384, May 1968

[DENN68] P.J. DENNING, "The Working Set Model for Program Behavior", *Commun. ACM* 11, 5 (May 1968), 323-333

[DENN70] P.J. DENNING, "Virtual Memory", *Comput. Surv.* 2, 3 (Sept. 1970), 153-189

[DENN72] P.J. DENNING; and S.C. SCHWARTZ, "Properties of the Working Set Model", *Commun. ACM* 15, 3 (March 1972), 191-198

[DENN76] P.J. DENNING; K.C. KAHN; J. LEROUDIER, D. POTIER and R. SURI, "Optimal Multiprogramming", *Acta Informatica* 7, 2 (1976), 197-216

[DENN30] P.J. DENNING, "Working Sets Past and Present", *IEEE Trans. on Software Engineering* 6, 1 (Jan. 1980), 64-84

[EAST76] M.C. EASTON; and P.A. FRANASZEK, *Use Bit Scanning in Replacement Decisions*, RC-6192, IBM Research, Yorktown Heights, N.Y., Sept. 1976

[MARS79] W. MARSHALL; and C.T. NUTE, "Analytic Modeling of 'Working-Set Like' Replacement Algorithms, *Conf. on Simulation, Measurement, and Modeling of Computer Systems*, 1979

[MORR72] J.B. MORRIS, "Demand Paging Through Utilization of Working Sets on the MANIAC II", *Commun. ACM* 15, 10 (Oct. 1972), 867-872

[PRIE73] B. PRIEVE, *A Page Partition Replacement Algorithm*, Ph. D. Thesis, Univ. of California, Berkeley, Dec. 1973

[PRIE76] B. PRIEVE; and R.S. FABRY, "VMIN – An Optimal Variable-Space Page Replacement Algorithm", *Commun. ACM* 19, 5 (May 1976), 295-297

[RODR73] J. RODRIGUEZ-ROSELL, "Empirical Working Set Behavior", *Commun. ACM* 16, 9 (Sept. 1973), 556-560

[SMIT76] A.J. SMITH, "A Modified Working Set Paging Algorithm", *IEEE Trans. on Computers* 10, (July 1980), 593-601

[SMIT80] A.J. SMITH, "Multiprogramming and Memory Contention", *Software – Practice and Experience* 25, 9 (Sept. 1976), 907-914