IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661, ISBN: 2278-8727Volume 7, Issue 4 (Nov. - Dec. 2012), PP 46-52
www.iosrjournals.org

Performance analysis of N-computing device under various load conditions

Snigdha Srivastava[1], Saumya Srivastava[2] & Sneha Mani Tripathi[3]
[1, 2 & 3] *Department of computer science and Engineering, Institute of Technology and Management, GIDA, Gorakhpur, India*

Abstract: *In the present time, a personal computer has a very high processing power than required for a single user system. Hence, it can be effectively used as a multiuser system which serves several users concurrently. This can be achieved by N-computing. We will create a multiuser environment on a uniprocessor system; this can be achieved, when there is a separate kernel for each user in the same operating system. This concept is related to the concept of mapping between user level thread and kernel level thread, which is discussed in this paper. Further the system's performance under various load conditions will be analyzed. In other words we want to utilize the full processing power of a personal computer for number of users simultaneously and also provide better performance at less cost.*

Keywords: *Kernel, mapping, n-computing, thread, uniprocessor.*

I. Introduction

N-computing is technology that allows multiple users to share single computer simultaneously; this means that with n-computing we could have one ordinary desktop computer catering for people or more at the same time. The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within programs. It exhibit poor performance if the cost of creating and managing parallelism is high. Even a fine-grained program can achieve good performance if the cost of creating and managing parallelism is low. Threads are a lighter-weight abstraction; multiple threads share an address space and its resources, and communication can be accomplished through shared data. Kernel level threads are, effectively, processes that share code and data space, where user level threads are implemented at the application level. This research divides responsibility for thread management between the kernel and application. Multithreading has emerged as a leading paradigm for the development of applications with demanding performance requirement. When number of users increase, so does the number of processes and hence number of threads, then performance is a major issue. This can be achieved by better thread management by kernel and better mapping of kernel level thread to user level thread which is explained in this paper.

II. Thread Management By Kernel

A thread is a light-weight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. On a single processor, multithreading generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor (including multi-core system), the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task. Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The kernel of an operating system allows programmers to manipulate threads via the system call interface. Some implementations are called a kernel thread, whereas a lightweight process (LWP) is a specific type of kernel thread that shares the same state and information. Multithreading has emerged as a leading paradigm for development of application with demanding performance requirements (For real time systems, a dynamic uniprocessor scheduling algorithm has an O (n log n) Worst case complexity [1].) Generally, threads are located on shared data structure: a shared run queue for ready threads and shared communication structures for blocked thread. Access control to the shared resource is maintained through lock-based mechanism which ensures safe access to such critical section of core [2]. There are two commonly used thread models: kernel level threads and user level threads. Kernel level threads suffer from the cost of frequent user-kernel domain crossings kernel scheduling priorities. User level threads are not integrated with the kernel; blocking all threads whenever one thread is blocked. The Scheduler Activations model, proposed by Anderson et al., combines CPU allocation decisions with application control over thread

scheduling. It discusses the performance characteristics of an implementation of Scheduler Activations for a uniprocessor BSD system, and proposes an analytic model for determining the class of applications that benefit from its use [3].

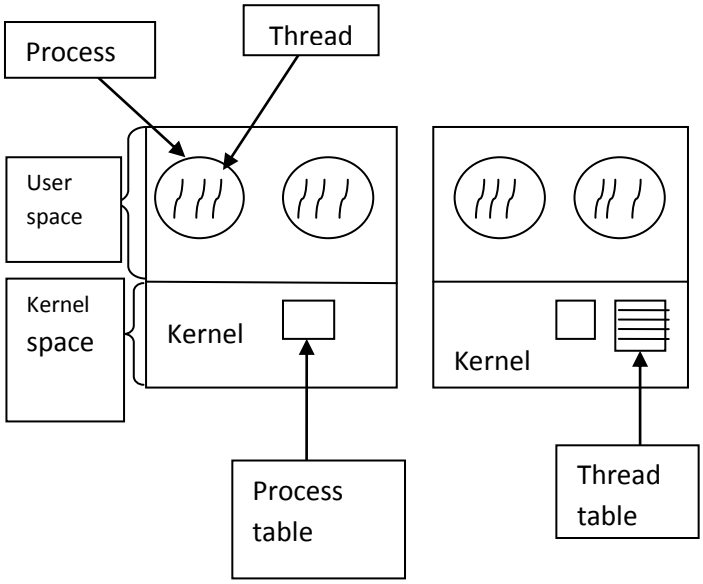

Figure 1: Thread and process

Kernel level threads share some of the disadvantages of processes. Switching between them is slow, taking an order of magnitude more time than a user level thread context switch. Also they are scheduled by the kernel; with no application control .this can be negatively affect performance. For example, if threads have different priorities and the priorities are not visible to the kernel, a low priority thread may be scheduled in place of a high priority one. User level threads systems control scheduling decisions, but because they are not integrated with the kernel, when one thread blocks (e.g. to perform I\O), all of the user level threads sharing the process are blocked. This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores or across a cluster of machines— because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks. The operating system kernel has complete control over the allocation of processors among address spaces including the ability to change the number of processors assigned to an application during its execution. To achieve this, the kernel notifies the address space thread scheduler of every kernel event affecting the address space, allowing the application to have complete knowledge of its scheduling state. . For fixed priority scheduling on uniprocessors, under certain conditions, the system's schedulability is maximized when the priorities are chosen in the inverse order of the task's deadlines [4]. The thread system in each address space notifies the kernel of the subset of user-level thread operations that can affect processor allocation decisions, preserving good performance for the majority of operations that do not need to be reflected to the kernel [5].

III. Thread Scheduling

Today's operating systems provide kernel threads for parallel applications and multi-threaded servers. Scheduling plays an important role with regard to efficiency and fairness — especially for distributed applications, multimedia processing and server processes. A multi-threaded application should be able to specify the scheduling strategy for its threads itself. In most modern operating systems the scheduling strategy is hard-coded into the kernel and cannot be changed by the user. There are a few user-level thread packages available, where the users can define the scheduling strategy. Yet user-level threads are not suitable for applications that interact with the operating system frequently, such as server processes or distributed applications. In this paper we present a concept that allows handling of kernel-thread scheduling from the user level using hierarchical schedulers. Each application can have one or more of its own schedulers, which can define the application-specific scheduling strategy. Thus, the programmer can implement his own scheduling strategy for his application or even for subsystems inside the application [6].

In most computer operating systems, the kernel is the central component. It is the bridge between the user and applications and the computer hardware. It also is the mechanism that allows the computer to handle multiple users and multiple tasks simultaneously. The types of kernels are the monolithic kernel, the microkernel, the hybrid kernel, the nanokernel and the exokernel. The kernel manages all of the computer's system resources. This includes long-term storage, the central processing unit (CPU), short-term memory and the input and output devices. When an application needs one of these resources, the kernel makes the resource available and completes the request. This handling of resources allows the operating systems to be both multi-user and multitasking. The operating system does not actually perform more than one task at a time. Instead, the kernel switches tasks at such a high speed that the computer appears to be performing multiple tasks. The kernel also is responsible for making sure that resources used by one user or process are not violated the request of another user or process. There two main types of kernels are the monolithic kernel and the microkernel. Monolithic kernels employ a supervisory method of resource management in which all of the operating system services are run in the same address space, called the kernel space. Some monolithic kernels can load and unload executable modules. This extends the operating system's capabilities while still maintaining a minimum amount of code running in the kernel space at any one time. Micro kernels run only the minimal amount of operating system services, such as memory management, thread management and inter-process communication in the kernel space. All other services, such as device drivers, user interfaces and file management, are run in the user space. The microkernel severely minimizes the amount of code that is running in the kernel mode.

Every thread has a thread priority assigned to it. Threads created within the common language runtime are initially assigned the priority of ThreadPriority.Normal. Threads created outside the runtime retain the priority they had before they entered the managed environment. You can get or set the priority of any thread with the Thread.Priority property. Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system. The details of the scheduling algorithm used to determine the order in which threads are executed varies with each operating system. Under some operating systems, the thread with the highest priority (of those threads that can be executed) is always scheduled to run first. If multiple threads with the same priority are all available, the scheduler cycles through the threads at that priority, giving each thread a fixed time slice in which to execute. As long as a thread with a higher priority is available to run, lower priority threads do not get to execute. When there are no more runnable threads at a given priority, the scheduler moves to the next lower priority and schedules the threads at that priority for execution. If a higher priority thread becomes runnable, the lower priority thread is preempted and the higher priority thread is allowed to execute once again. On top of all that, the operating system can also adjust thread priorities dynamically as an application's user interface is moved between foreground and background.

Efficient front end scheduling for simultaneous multithreading, where priority given to a thread can be overridden if it suffers from an excessive amount of incorrectly speculated instructions, is presented. Simulation results demonstrate that the proposed scheme not only significantly reduces the number of wrong-path instructions but also improves the instruction throughput. The overall performance of a simultaneous multithreading (SMT) processor depends on many factors including how threads are selected and the number of threads from which to fetch instructions .Further, how to allocate the limited fetch slots to the selected threads must be judiciously decided: if instructions fetched from a thread reside in the instruction window for too many cycles before they are issued ,they occupy entries of the window that could be used by instructions of other threads ,ultimately limiting the Instruction-Level Parallelism (ILP) and the Thread-Level Parallelism (TLP) which can be exploited [7]. As multithreaded applications become common, scheduling inside applications plays a very important role for efficiency and fairness. There are different categories of applications that especially profit from specific scheduling strategies:

1. Calculating applications with small working set. These applications often consist of many threads. Threads rarely block. Scheduling can be important to make the working set fit into the cache.
2. Calculating applications with large working set. These applications consist of many threads, which are often blocked because of page faults. Scheduling can be very helpful to minimize the working set and to prevent page thrashing.
3. Distributed user applications that communicate across a network, for example through message passing, virtual shared memory, or object invocation. These applications consist of many threads, which are often blocked in kernel while waiting for a response from other application parts. Application-specific scheduling on one node can help to minimize the idle time of other nodes.
4. Servers (daemon processes). Servers consist of many threads, which are often blocked (for example when waiting for the network). Scheduling is very important to guarantee fairness between different jobs and to gain maximum efficiency.
5. Multimedia applications with different real-time demands. In many cases correct scheduling decisions are only possible if the quality-of-service parameters of the application are available.

User-level scheduling with kernel threads allows the user to implement the optimal scheduling strategy for his application, for his modules or even between his applications. Especially applications that operate with a lot of system interaction, such as distributed applications, servers and applications with large working sets need to define their own scheduling strategies to achieve higher efficiency [6].

IV. Mapping Of Kernel Thread To User Thread

Threads can be supported either at user level or in the kernel. Neither approach has been fully satisfactory. User-level threads are managed by runtime library routines linked into each application so that thread management operations require no kernel intervention. The result can be excellent performance: in systems, the cost of user-level thread operations is within an order of magnitude of the cost of a procedure call. User-level threads are also flexible; they can be customized to the needs of the language or user without kernel modification. User-level threads execute within the context of traditional processes; indeed, user-level thread systems are typically built without any modifications to the underlying operating system kernel. The thread package views each process as a "virtual processor," and treats it as a physical processor executing under its control; each virtual processor runs user-level code that pulls threads off the ready list and runs them. In reality, though, these virtual processors are being multiplexed across real, physical processors by the underlying kernel. "Real world" operating system activity, such as multiprogramming, I/O, and page faults, distorts the equivalence between virtual and physical processors; in the presence of these factors, user-level threads built on top of traditional processes can exhibit poor performance or even incorrect behavior. Kernel is unaware of it. When one thread gets blocked, even though there are Runnable threads, they do not get a chance to run Kernel threads: Scheduled by kernel; in kernel's view, it is like a process only, but a lightweight process. Each such thread is a schedulable entity. User threads on top of kernel threads: Here kernel is aware of the threads that are running in the user space. So user level threads can make blocking calls and kernel can run other threads from the same process. Also, if a one user level thread blocks, then the kernel thread on which the user thread was running, is also blocked; and if that kernel thread was the only thread that was running on a processor, then the processor becomes unusable. If the user level thread is blocked, to have multiple kernel threads and context switch between them, then if all the kernel threads are blocked (because the corresponding user level thread blocks), then none will be running in the system. Instead of this, i.e., In this case, we can avoid the need to create multiple kernel threads right. The operating system kernel provides each user-level thread system with its own virtual multiprocessor, the abstraction of a dedicated physical machine except that the kernel may change the number of processors in that machine during the execution of the program [5]. There are several aspects to this abstraction:

•The kernel allocates processors to address spaces; the kernel has complete control over how many processors to give each address space's virtual multiprocessor.

•Each address space's user-level thread system has complete control over which threads to run on its allocated processors, as it would if the application were running on the bare physical machine.

• The kernel notifies the user-level thread system whenever the kernel changes the number of processors assigned to it; the kernel also notifies the thread system whenever a user-level thread blocks or wakes up in the kernel (e.g., on I/O or on a page fault). The kernel's role is to vector events to the appropriate thread scheduler, rather than to interpret these events on its own.

•The user-level thread system notifies the kernel when the application needs more or fewer processors. The kernel uses this information to allocate processors among address spaces. However, the user level notifies the kernel only on that subset of user-level thread operations that might affect processor allocation decisions. As a result, performance is not compromised; the majority of thread operations do not suffer the overhead of communication with the kernel.

• The application programmer sees no difference, except for performance, from programming directly with kernel threads.

There are two types of threads to be managed in a modern system: User threads and kernel threads. User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs. Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously. In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies:

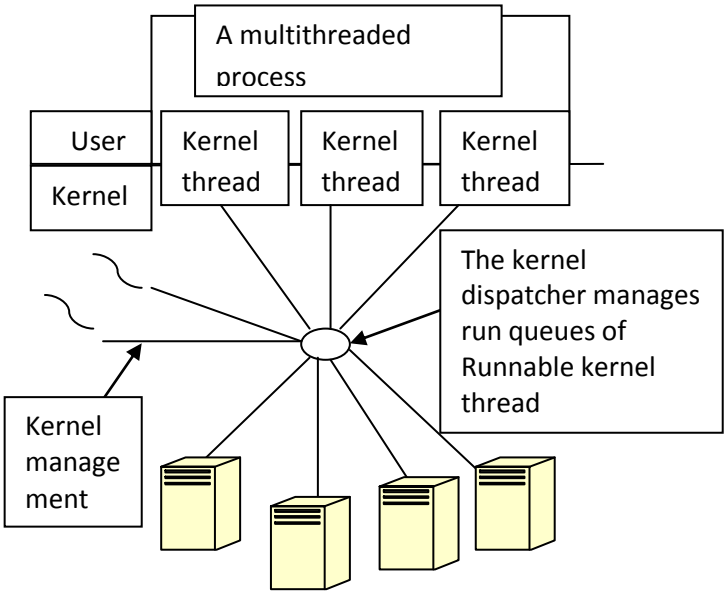

Figure 2: Mapping between user level thread and kernel level thread

- **Many-To-One Model**

In the many-to-one model, many user-level threads are all mapped onto a single kernel thread. Thread management is handled by the thread library in user space, which is very efficient. However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue. Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs. Green threads for Solaris and GNU Portable Threads implement the many-to-one model.

- **One-To-One Model**

The one-to-one model creates a separate kernel thread to handle each user thread. One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs. However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system. Most implementations of this model place a limit on how many threads can be created. Linux and Windows from 95 to XP implement the one-to-one model for threads.

- **Many-to-Many Model**

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models. Users have no restrictions on the number of threads created. Blocking kernel system calls do not block the entire process. Processes can be split across multiple processors. Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors. One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation. IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9.

Threads are the vehicle for concurrency in many approaches to parallel programming. Threads can be supported either by the operating system kernel or by user-level library code in the application address space, but neither approach has been fully satisfactory. The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within programs. Even a coarse-grained parallel program can exhibit poor performance if the cost of creating and managing parallelism is high. Even a fine-grained program can achieve good performance if the cost of creating and managing parallelism is low. One way to construct a parallel program is to share memory between a collection of traditional UNIX-like processes, each consisting of a single address space and a single sequential execution stream within that address space [5].

V. Multitasking On Uniprocessor

Multitasking is the ability of a computer to run more than one program, or task , at the same time. Multitasking contrasts with single-tasking, where one process must entirely finish before another can begin.

MS-DOS is primarily a single-tasking environment, while Windows 3.1 and Windows NT are both multi-tasking environments. On a single-processor multitasking system, multiple processes don't actually run at the same time since there's only one processor. Instead, the processor switches among the processes that are active at any given time. Because computers are so fast compared with people, however, it appears to the user as though the computer is executing all of the tasks at once. Multitasking also allows the computer to make good use of the time it would otherwise spend waiting for I/O devices and user input--that time can be used for some other task that doesn't need I/O at the moment

Within the category of multitasking, there are two major sub-categories: preemptive and non-preemptive (or cooperative). In non-preemptive multitasking , use of the processor is never taken from a task; rather, a task must voluntarily yield control of the processor before any other task can run. Windows 3.1 uses non-preemptive multitasking for Windows applications. Programs running under a non-preemptive operating system must be specially written to cooperate in multitasking by yielding control of the processor at frequent intervals. Programs that do not yield sufficiently often cause non-preemptive systems to stay "locked" in that program until it does yield. An example of failed non-preemptive multitasking is the inability to do anything else while printing a document in Microsoft Word for Windows 2.0a. This happens because Word does not give up control of the processor often enough while printing your document. The worst case of a program not yielding is when a program crashes. Sometimes, programs which crash in Windows 3.1 will crash the whole system simply because no other tasks can run until the crashed program yields. Preemptive multitasking differs from non-preemptive multitasking in that the operating system can take control of the processor without the task's cooperation. (A task can also give it up voluntarily, as in non-preemptive multitasking.) The process of a task having control taken from it is called preemption. Windows NT uses preemptive multitasking for all processes except 16-bit Windows 3.1 programs. As a result, a Window NT application cannot take over the processor in the same way that a Windows 3.1 application can. A preemptive operating system takes control of the processor from a task in two ways:

- When a task's time quantum (or time slice) runs out. Any given task is only given control for a set amount of time before the operating system interrupts it and schedules another task to run.
- When a task that has higher priority becomes ready to run. The currently running task loses control of the processor when a task with higher priority is ready to run regardless of whether it has time left in its quantum or not.

At any given time, a processor (CPU) is executing in a specific context. This context is made up of the contents of its registers and the memory (including stack, data, and code) that it is addressing. When the processor needs to switch to a different task, it must save its current context (so it can later restore the context and continue execution where it left off) and switch to the context of the new task. This process is called context switching . When Windows NT switches tasks, it saves all of the registers of the task it's leaving and re-loads the registers of the task to which it's switching. This process also enables the proper address space for the task to which Windows NT is switching. In a single processor multi-programming system, multiple processes are contained within memory (or its swapping space). Processes alternate between executing (Running), being able to be executed (Ready), waiting for some event to occur (Blocked), and swapped-out (Suspend). A significant goal is to keep the processor busy, by "feeding" it processes to execute, and always having at least one process able to execute. The key to keeping the processor busy is the activity of process scheduling, of which we can categories three main types:

- **Long-term scheduling:** the decisions to introduce new processes for execution, or re-execution.
- **Medium-term scheduling:** the decision to add to (grow) the processes that are fully or partially in memory.
- **Short-term scheduling:** the decisions as to which (Ready) process to execute next.

The performance of the various architectures is examined on a uniprocessor system. Three workloads are examined: no disk-I/O, moderate disk-I/O and heavy disk-I/O. These three workloads highlight the differences among the architectures. The design of real-time systems is usually approached by decomposing the system in a set of tasks that are scheduled on set of processing resources. Real time requirements impose constraints on the necessary computational resources. For these reasons, the scientific community has focused on establishing the conditions by which the task set is schedulable on a set of processing resources by some scheduling policy under given real time requirements. For fixed priority scheduling on uniprocessor, under certain conditions, the system's schedulability is maximized when the priorities are chosen in the inverse order of the task's deadlines, a similar condition has also been shown for dynamic scheduling on uniprocessor, where under certain conditions schedulability is maximized when higher priorities are given to the tasks with lower absolute deadlines. This scheduling policy is known as the earliest deadline first scheduling policy.

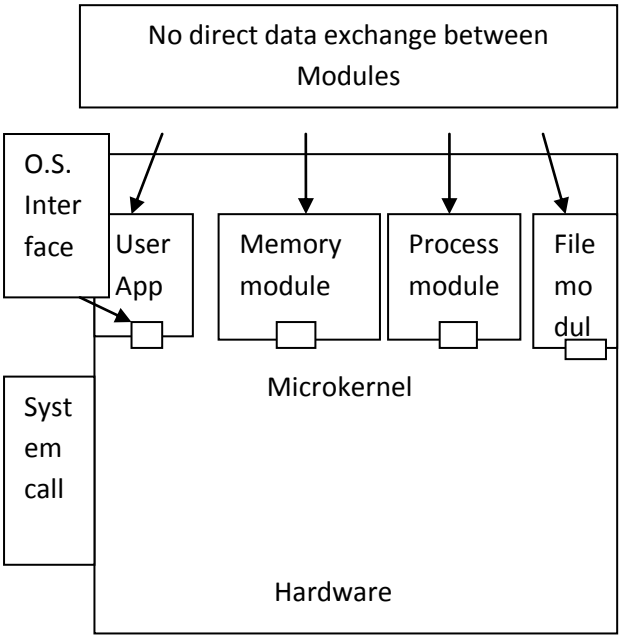

Figure 3: Uniprocessor environment

VI. Conclusion

We've all become accustomed to the PC model, which allows every user to have their own CPU, hard disk, and memory to run their applications. But personal computers have now become so powerful that most people can't possibly use all the processing power they purchase. N-Computing is a modern take on the time-honored concept where multiple users share the processing power of a single computer. This approach has several advantages over the traditional PC model, including lower overall costs, better energy efficiency, and simplified administration. Our proposed solution and further analysis would increase performance of any uniprocessor system. The n-computing solution works because today's PCs are so powerful that the vast majority of applications only use a small fraction of the computer's capacity. The proposed system will be more efficient and provide better performance at less cost and it also consumes less power.

References

Journal Papers:

[1] K. Schwan and H. Zhou [Georgia Institute of Technology], *"Dynamic scheduling of hard real-time tasks and real-time threads",* IEEE transactions on Software Engineering, Aug. 1992, *Volume 18, Issue 8*, p. 736-748.

[2] K.Debattista, K.Vella and J.Cordina, *"Wait-free cache-affinity thread scheduling",* IEEE Proc., Softw.-April 2003, *Volume 150, Issue 2*, p. 137-146.

[3] Christopher Small and Margo Seltzer, *"Scheduler Activations on BSD:Sharing Thread Management Between Kernel and Application"*, Harvard University.

[4] L. Mangeruca, A. Ferrari and A. L. Sangiovanni-Vincentelli, *"Uniprocessor scheduling under precedence constraints", RTAS'06, p. 157-166*, IEEE Computer Society, Washington DC, USA.

[5] Thomas E. Anderson, Brian N. Bershad and others**,** *"Schedular activations: effective kernel support for the user-level management of parallelism",* ACM Transactions on Computer Systems [New York, USA], Feb. 1992, *Volume 10, Issue 1*, p. 53-79.

[6] Thomas Riechman, Jurgen Kleinoder, *"User-Level Scheduling with Kernel Threads",* Department of Computer Science, University of Erlangen-Nurnberg, Germany.

[7] D. Kang and J.L.Gaudiot [University of California], *"Speculation- aware thread scheduling for simultaneous multithreading"*, Electron. Lett. 4 March 2004, *Volume 40, Issue 5*, p. 296-298.