© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

STUDY OF MULTITHREADING ABILITY OF CENTRAL PROCESSING UNIT

[1]Sawati, Research Scholar, Department of CSE, IIET (Jind), jain008swati@gmail.com
[2]Mrs. Nikita, Assistant Professor & HOD, Department of CSE, IIET (Jind), nikitasagar@gmail.com

Abstract: In computer architecture, multithreading is ability of a central processing unit (CPU) or a single core within a multi-core processor to implement multiple processes or threads concurrently, appropriately supported by operating system. it plea differs from multiprocessing, as within multithreading processes & threads have to share resources of a single or multiple cores: computing units, CPU caches, & translation lookaside buffer (TLB). Multiprocessing systems include multiple complete processing, multithreading targets to increase utilization of a single core by using thread-level as well as instruction stage parallelism. As two approaches are complementary, they are sometimes combined within systems within multiple multithreading CPUs & within CPUs within multiple multithreading cores. A scheduler might aim at one of several goals, for example, maximizing throughput, minimizing response time, or minimizing latency, maximizing fairness. In practice, these goals often conflict, thus a scheduler would implement a acceptable agreement. Preference is given to any one of concerns mentioned above, depending upon user's needs & objectives. Objective of research is to enhance efficiency of scheduling dependent task using enhanced multithreading.

Keywords: TLP, Response Time, Latency, throughput, multithreading, Scheduling

[1]Introduction

The multithreading paradigm had been become more favoured as efforts to further use instruction-level parallelism have stalled since late 1990s. it allowed concept of throughput computing to re-emerge from more generalized field of transaction processing; despite fact that it is very difficult to additionally speed up a single thread or one program, most computer systems are literally multitasking among multiple threads or programs. Thus, methods that improve throughput of all tasks result within overall performance gains.

Types of multithreading

Block multithreading

The simplest type of multithreading happens when one thread executes until it is blocked by an event that normally would create a big latency stall. Such a stall might be a cache miss that had been to access off-chip memory, that might take hundreds of CPU cycles for data to return. Instead of waiting for stall to resolve, a threaded processor will switch execution to another thread that was ready to run. Only when data for previous thread had arrived, would previous thread be placed back on list of ready-to-run threads.

For example:

1. Cycle *i*: instruction *j* from thread *A* is issued.
2. Cycle *i* + 1: instruction *j* + 1 from thread *A* is issued.
3. Cycle *i* + 2: instruction *j* + 2 from thread *A* is issued, that is a load instruction that misses within all caches.
4. Cycle *i* + 3: thread scheduler invoked, switches to thread *B*.
5. Cycle *i* + 4: instruction *k* from thread *B* is issued.
6. Cycle *i* + 5: instruction *k* + 1 from thread *B* is issued.

Conceptually, it is similar to cooperative multi-tasking used within real-time operating systems, within which tasks voluntarily give up implementation time when they need to wait upon some type of event. it type of multithreading is known as block, cooperative or coarse-grained multithreading.The target of multithreading hardware support to allow quick switching between a blocked thread & another thread ready to run. To achieve it goal, hardware cost is to replicate program visible registers, as well as some processor control registers (such as program counter).change efficiently between active threads, Such as to quickly switch between two threads, register hardware needs to be represented twice.

Additional hardware support for multithreading allows thread switching to be done within one CPU

© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

cycle, bringing performance enhancements. Also, Some extra hardware allows each thread to behave as if it were executing alone & not sharing any hardware resources within other threads, minimizing amount of software changes needed within application & operating system to support multithreading.

Many families of microcontrollers & embedded processors have multiple register banks to enable quick context switching for interrupts. Such schemes could be considered a type of block multithreading among user program thread & interrupt threads.

Interleaved multithreading

The purpose of interleaved multithreading is to exclude all data dependency stalls from execution pipeline. Since one thread is comparatively independent from other threads, there is very few chance of one instruction within one pipelining stage needing an output from an older instruction within pipeline. Conceptually, it is similar to preemptive multitasking used within operating systems; an analogy would be that time slice given to each active thread is one CPU cycle.

For example:

1. Cycle *i* + 1: an instruction from thread *B* is issued.
2. Cycle *i* + 2: an instruction from thread *C* is issued.

This kind of multithreading was first called barrel processing, within which staves of a barrel represent pipeline stages & their executing threads.Fine-grained, preemptive,Interleaved or time-sliced multithreading are more modern terminology.

In addition to hardware costs discussed within block type of multithreading, interleaved multithreading had been an further cost of each pipeline stage tracking thread ID of instruction it is processing. Also, since there are more threads being executed simultaneously within pipeline, shared resources such as caches & TLBs need to be larger to avoid thrashing between different threads.

Simultaneous multithreading

The most advanced kind of multithreading applies to superscalar processors. In contrast a normal superscalar processor generates multiple instructions from a single thread each CPU cycle, within simultaneous multithreading (SMT) a superscalar processor could issue instructions from multiple threads each CPU cycle. identifying that any single thread had been a limited number of instruction-level parallelism, it type of multithreading tries to use parallelism available beyond multiple threads to decrease waste associated within unexploited issue slots.

For example:

1. Cycle *i*: instructions *j*&*j* + 1 from thread *A*& instruction *k* from thread *B* are simultaneously issued.
2. Cycle *i* + 1: instruction *j* + 2 from thread *A*, instruction *k* + 1 from thread *B*, & instruction *m* from thread *C* are all simultaneously issued.
3. Cycle *i* + 2: instruction *j* + 3 from thread *A*& instructions *m* + 1 &*m* + 2 from thread *C* are all simultaneously issued.

To distinguish other types of multithreading from SMT, term "temporal multithreading" is used to denote when instructions from only one thread could be issued at a time.

In addition to hardware costs discussed for interleaved multithreading, SMT had been additional cost of each pipeline stage tracking thread ID of each instruction being processed. Again, shared resources such as caches & TLBs have to be sized for large number of active threads being processed.

Implementations include DEC (later Compaq) EV8 (not completed), IBM POWER5, Intel Hyper-Threading, Sun Microsystems UltraSPARC T2, CRAY XMT, &MIPS MT.

Implementation specifics

A major area of research is thread scheduler that must quickly choose among list of ready-to-run threads to implement next as well as maintain ready-to-run & stalled thread lists. An important subtopic is different thread priority schemes that could be used by scheduler. thread scheduler might be implemented totally within software, totally within hardware, or as a hardware/software combination.

Another area of research is what type of events would cause a thread switch: cache misses, inter-thread communication, DMA completion, etc.

If multithreading scheme replicates all of software-visible state, including privileged control registers & TLBs, then it provides virtual machines to be created for each thread. it allows each thread to run its own operating system on same processor. On other hand, if only user-mode state is saved, then less hardware is required, that would allow more threads to be active at one time for same die area or cost.

[2] LITERATURE REVIEW

***Yeh-Ching Chung wrote on* "Performance Analysisand Applications of A Compile-Time Optimization Approach for List Scheduling**

© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

Algorithms on Distributed Memory Multiprocessors"

They have reportedacompile-time optimization approach, *bottom-up top-down duplication heuristic* (BTDH), to static scheduling of *directed+cyclic graphs* (DAGS) on *distributed memory multiprocessors* (DMMs). In it paper, they discuss applications of BTDH for *list scheddhg algorithms* (LSAs). There are two ways to use BTDH for LSAs.BTDHcan be used within aLSAto form a new *scheduling* algorithm (LSA/BTDH). It could be usedas apure *opti*mization algorithm for a LSA (LSA-BTDH). We have used BTDH within two already knownLSAs, *highest level first within estimated time* (HLFET) &*earlier taskfirst* (ETF) heuristics. We have calculate extensive simulation to study performance of BTDH for LSAs. Three parameters, ratio of average communication cost to average computation cost *(CCR)* of a DAG, *graph parallelism* (GP) of a DAG& number(PN) of a DMM,aresimulated. From sunulation, they have following conclusions. Given a DAG, GP of DAG could accurately predict number of processors to be used such that a good scheduling length & a good resource utilization (or efficiency) could be achieved .

Ishfaq Ahmad1 & Yu-Kwong Kwok2 wrote on "On Parallelizing Multiprocessor Scheduling Problem"

Beingheuristics for scheduling a node & edge weighted directed task graph to multiple processors could produce satisfactory solutions but incur high time complexities that tend to exacerbate within more realistic environments within relaxed assumptions. Accordingly, these heuristics do not scale well & cannot handle problems of normal sizes. A natural approach to reducing complexity while aiming for a similar or potentially better solution is to parallelize scheduling algorithm. it could be done by partitioning task graphs & concurrently generating partial schedules for partitioned parts that are then concatenated to obtain final schedule. problem, however, is nontrivial as there existsdependencies among nodes of a task graph that must be preserved for generating a valid schedule. Moreover, time clock for scheduling is global for all processors (that are executing parallel scheduling algorithm), making inherent parallelism invisible. In it paper, they introduce a parallel algorithm that is guided by a systematic partitioning of task graph to perform scheduling using multiple processors. algorithm schedules both tasks & messages, & is suitable for graphs within arbitrary computation & communication costs, & is applicable to systems within arbitrary network topologies using homogeneous or heterogeneous processors. We have executed algorithm on Intel Paragon & compared it within three closely attached algorithms. experimental results indicate that our algorithm yields higher quality solutions while using an order of magnitude less scheduling times. algorithm also exhibits an interesting trade-off between solution quality & speedup while scaling well within problem size.

Maruf Ahmed, Sharif M. H. Chowdhury wrote on List Heuristic Scheduling Algorithms for Distributed Memory Systems within Improved Time Complexity

They present a compile time list heuristic scheduling algorithm known as*Low Cost Critical Path algorithm (LCCP)* for distributed memory systems. LCCP had been low scheduling cost for both homogeneous & heterogeneous systems. In some recent papers list heuristic scheduling algorithms keep their scheduling rate low by using a fixed size heap & a FIFO, where heap always keeps fixed number of tasks & excess tasks are inserted within FIFO. When heap had been empty spaces, tasks are inserted within it from FIFO. Best known list scheduling algorithm depends on it strategy requires two heap restoration operations, one after extraction & another after insertion. Our LCCP algorithm enhances on it by using only one such operation for both extraction & insertion, that within theory reduces scheduling cost without compromising scheduling performance. In our experiment they compare LCCP within other well known list scheduling algorithms & it shows that LCCP is fastest among all.

Wayne F. Boyer wrote on "Non-evolutionary algorithm for scheduling dependent tasks within distributed heterogeneous computing environments"

The Problem of finding thebest matching & scheduling of interdependent tasks within distributed heterogeneous computing (DHC) environments is generally known to be an NP-hard problem. In a DHC system, task execution time is dependent on machine to which it is assigned & task precedence constraints are represented by a directed acyclic graph. Recent research within evolutionary techniques had been shown that genetic algorithms usually obtain more efficient schedules that other known algorithms.

We propose a non-evolutionary random scheduling (RS) algorithm for efficient matching & scheduling of inter-dependent tasks within a DHC system. RS is

© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

a succession of randomized task orderings & a heuristic mapping from task order to schedule. Randomized task ordering is effectively a topological sort where outcome might be any possible task order for which task precedent constraints are maintained. A detailed comparison to existing evolutionary techniques (GA & PSGA) shows proposed algorithm is less complex than evolutionary techniques, computes schedules within less time, & requires less memory & fewer tuning parameters. Simulation results show that average schedules produced by RS are approximately as efficient as PSGA schedules for all cases studied & clearly more efficient than PSGA for certain cases.

[3]RESEARCH METHODOLOGY

In computing, **scheduling** is method by which work specified by some means is assigned to resources that complete work. work might be virtual computation elements such as threads, processes or data flows, that are within turn scheduled onto hardware resources such as processors, network links or expansion cards.

A scheduler is what carries out scheduling activity. Schedulers are often implemented so they keep all computer resources busy (as within load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, & an intrinsic part of execution model of a computer system; concept of scheduling makes it possible to have computer multitasking within a single central processing unit (CPU).

A scheduler might aim at one of several goals, for example, maximizing *throughput* (total amount of work completed per time unit), minimizing *response time* (time from work becoming enabled until first point it begins execution on resources), or minimizing *latency* (the time between work becoming enabled & its subsequent completion), maximizing *fairness* (equal CPU time to each process, or more generally appropriate times according to priority & workload of each process). In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler would implement a suitable compromise. Preference is given to any one of concerns mentioned above, depending upon user's needs & objectives.

In real-time environments, such as embedded systems for automatic control within industry (for example robotics), scheduler also must ensure that processes could meet deadlines; it is crucial for keeping system stable. Scheduled tasks could also be distributed to remote devices across a network & managed through an administrative back end.

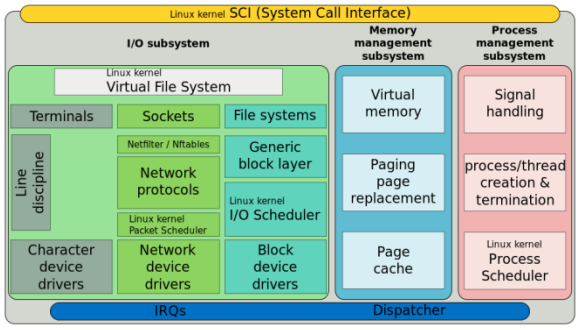
Scheduling disciplines are algorithms used for distributing resources among parties that simultaneously & asynchronously request them. Scheduling disciplines are used within routers (to handle packet traffic) as well as within operating systems (to share CPU time among both threads & processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation & to ensure fairness amongst parties utilizing resources. Scheduling deals within problem of deciding that of outstanding requests is to be allocated resources. There are several different scheduling algorithms. In it section, we introduce several of them.

In packet-switched computer networks & other statistical multiplexing, notion of a **scheduling algorithm** is used as an alternative to first-come first-served queuing of data packets.

The simplest best-effort scheduling algorithms are round-robin, fair queuing (a max-min fair scheduling algorithm), proportionally fair scheduling & maximum throughput. If differentiated or guaranteed quality of service is offered, as opposed to best-effort communication, weighted fair queuing might be utilized.

In advanced packet radio wireless networks such as HSDPA (High-Speed Downlink Packet Access) 3.5G cellular system, **channel-dependent scheduling** might be used to take advantage of channel state information. If channel conditions are favourable, throughput & system spectral efficiency might be increased. In even more advanced systems such as LTE, scheduling is combined by channel-dependent packet-by-packet dynamic channel allocation, or by assigning OFDMA multi-carriers or other frequency-domain equalization components to users that best could utilize them.



© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

Fig 1 System Call interface

Highly simplified structure of Linux kernel: *process scheduler*, *I/O scheduler*, *packet scheduler*& other subsystems

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, & cooperative for processes & threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, & cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, & Server)	Yes	Multilevel feedback queue

[4] CHALLENGES WITHIN RESEARCH

Multiple threads could interfere within each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved but could be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement within its HyperThreading technology,[1] while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run within parallel. On other hand, hand-tuned assembly language programs using MMX or Altivec extensions & performing data pre-fetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading & could indeed see degraded performance due to contention for shared resources.

From software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs & operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel software techniques used for computer multitasking of computer programs. Thread scheduling is also a major problem within multithreading.

[5] SCOPE OF RESEARCH

If a thread gets a lot of cache misses, other threads could continue taking advantage of unused computing resources, that might lead to faster overall execution as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all computing resources of CPU (because instructions depend on each other's result), running another thread might prevent those resources from becoming idle. If several threads work on same set of data, they could actually share their cache, leading to better cache usage or synchronization on its values.

© INTERNATIONAL JOURNAL FOR RESEARCH PUBLICATION & SEMINAR
ISSN: 2278-6848 | Volume: 07 Issue: 05 | July – September 2016
Paper is available at www.jrps.in | Email : info@jrps.in

REFERENCE

1. *Remzi H. Arpaci-Dusseau; Andrea C. Arpaci-Dusseau (January 4, 2015). "Chapter 7: Scheduling: Introduction, Section 7.6: A New Metric: Response Time". Operating Systems: Three Easy Pieces (PDF). p. 6. Retrieved February 2, 2015.*
2. *Paul Krzyzanowski (2014-02-19). "Process Scheduling: Who gets to run next?". cs.rutgers.edu. Retrieved 2015-01-11.*
3. *Abraham Silberschatz, Peter Baer Galvin & Greg Gagne (2013). Operating System Concepts **9**. John Wiley & Sons,Inc. ISBN 978-1-118-06333-0.*
4. Here is C-code for FCFS
5. Early Windows at Wayback Machine
6. *Sriram Krishnan. "A Tale of Two Schedulers Windows NT & Windows CE".*
7. Inside Windows Vista Kernel: Part 1, Microsoft Technet
8. *"Vista Kernel Improvements".*
9. *"Technical Note TN2028 - Threading Architectures".*
10. *"Mach Scheduling & Thread Interfaces".*
11. http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6
12. *Molnár, Ingo (2007-04-13). "[patch] Modular Scheduler Core & Completely Fair Scheduler [CFS]". linux-kernel (Mailing list).*