Analysis of multi-threading and cache memory latency masking on processor performance using thread synchronization technique

Akhigbe-mudu Thursday Ehis[1]

[1]Department of Computer Science/ Information Technology, African Institute of Science Administration and Commercial Studies Lome, Lome, Republic of Togo

Correspondence: Akhigbe-mudu Thursday Ehis, Department of Computer Science/ Information Technology, African Institute of Science Administration and Commercial Studies Lome, Lome, Republic of Togo. E-mail: akhigbe-mudut@iaec-university.tg

Received: July 21, 2023

Accepted: September 25, 2023

DOI: 10.14295/bjs.v3i1.458

URL: https://doi.org/10.14295/bjs.v3i1.458

Abstract

Multithreading is a process in which a single processor executes multiple threads concurrently. This enables the processor to divide tasks into separate threads and run them simultaneously, thereby increasing the utilization of available system resources and enhancing performance. When multiple threads share an object and one or more of them modify it, unpredictable outcomes may occur. Threads that exhibit poor locality of memory reference, such as database applications, often experience delays while waiting for a response from the memory hierarchy. This observation suggests how to better manage pipeline contention. To assess the impact of memory latency on processor performance, a dual-core MT machine with four thread contexts per core is utilized. These specific benchmarks are chosen to allow the workload to include programs with both favorable and unfavorable cache locality. To eliminate the issue of wasting the wake-up signals, this work proposes an approach that involves storing all the wake-up calls. It asserts the wake-up calls to the consumer and the producer can store the wake-up call in a variable. An assigned value in working system (or kernel) storage that each process can check is a semaphore. Semaphore is a variable that reads, and update operations automatically in bit mode. It cannot be actualized in client mode since a race condition may persistently develop when two or more processors endeavor to induce to the variable at the same time. This study includes code to measure the time taken to execute both functions and plot the graph. It should be noted that sending multiple requests to a website simultaneously could trigger a flag, ultimately blocking access to the data. This necessitates some computation on the collected statistics. The execution time is reduced to one third when using threads compared to executing the functions sequentially. This exemplifies the power of multithreading.

Keywords: concurrency, multithreading, mutex, semaphore, synchronization.

Análise de mascaramento de latência de memória cache e *multi-threading* no desempenho do processador usando técnica de sincronização de *thread*

Resumo

Multithreading é um processo no qual um único processador executa vários threads simultaneamente. Isso permite que o processador divida tarefas em threads separados e as execute simultaneamente, aumentando assim a utilização dos recursos disponíveis do sistema e melhorando o desempenho. Quando vários threads compartilham um objeto e um ou mais deles o modificam, podem ocorrer resultados imprevisíveis. Threads que apresentam baixa localização de referência de memória, como aplicativos de banco de dados, geralmente sofrem atrasos enquanto aguardam uma resposta da hierarquia de memória. Esta observação sugere como gerenciar melhor a contenção do pipeline. Para avaliar o impacto da latência da memória no desempenho do processador, é utilizada uma máquina MT *dual-core* com quatro contextos de thread por núcleo. Esses benchmarks específicos são escolhidos para permitir que a carga de trabalho inclua programas com localidade de cache favorável e desfavorável. Para eliminar o problema do desperdício dos sinais de despertar, este trabalho propõe uma abordagem que envolve o armazenamento de todas as chamadas de despertar. Ele afirma as chamadas de despertar para o consumidor e o produtor pode armazenar a chamada de despertar em uma variável. Um valor atribuído no armazenamento do sistema funcional (ou *kernel*) que cada processo pode

verificar é um semáforo. Semáforo é uma variável que lê e atualiza operações automaticamente no modo *bit*. Ele não pode ser atualizado no modo cliente, pois uma condição de corrida pode se desenvolver persistentemente quando dois ou mais processadores tentam induzir a variável ao mesmo tempo. Este estudo inclui código para medir o tempo necessário para executar ambas as funções e traçar o gráfico. Deve-se observar que o envio simultâneo de várias solicitações a um site pode acionar um sinalizador, bloqueando, em última instância, o acesso aos dados. Isso requer alguns cálculos nas estatísticas coletadas. O tempo de execução é reduzido para um terço ao usar *threads* em comparação com a execução sequencial das funções. Isso exemplifica o poder do *multithreading*.

Palavras-chave: simultaneidade, *multithreading*, *mutex*, semáforo, sincronização.

1. Introduction

Human carries out a wide range of operations simultaneously – or concurrency, as it will be referred to in this study. Breathing, blood circulation, digestion, thinking, and walking, for instance, can all happen concurrently. All the senses, vision, touch, smell, taste, and hearing can be used at the same time. Computers also have the ability to perform operations concurrently. It is typical for personal computers to compile a program, send a file to a printer, and receive electronic mail messages over a network simultaneously.

Operating systems on single processor computers create the illusion of concurrent execution by rapidly switching between activities, but on these computers, only one instruction can execute at a time. Most programming languages do not allow for concurrent activities. Instead, control statements provide sequential control, allowing one action to be performed at a time, with execution moving on to the next action after the previous one has finished (Hassanein et al., 2020). Java allows for concurrency through the language and APIs (Carvalho et al., 2023).

It specifies that an application should contain separate threads of execution, where each thread has its own method call stack and program counter, while also sharing application-wide resources such as memory. This capability is known as Multithreading (Adam, 2022). A thread is a unit of control within a process. When a thread runs, it executes a function in the program. The process associated with a running program begins with one running thread, known as the main thread, which executes the "main" function of the program. In a multithreaded program, the main thread creates other threads that execute other functions.

These other threads can also create additional threads, and so on. Threads are created using constructs provided by the programming language or the functions provided by an application-programming interface (API). Each thread has its own stack of activation records and its own copy of the CPU registers, including the stack pointer and the program counter, which collectively describe the state of the thread's execution (Eslamimehr et al., 2018).

1.1 Multithreaded Servers

Multithreaded Server: A server with multiple threads is referred to as a Multithreaded Server (Vayadande et al., 2022). When a client sends a request, a thread is created which allows the user to interact with the server. Several threads need to be created to handle multiple requests from multiple clients simultaneously (Figure 1).

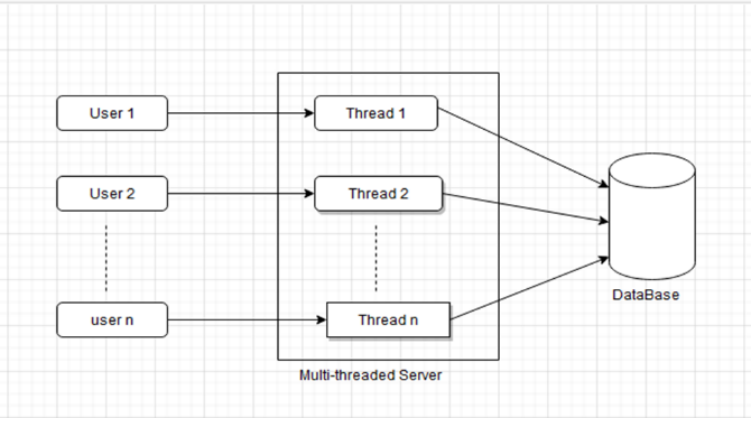

Figure 1. Multithreaded Server. Source: Adam (2022).

Multithreaded (MT) CPUs are designed to conceal the impact of memory latency by running multiple instruction streams in parallel (Cai et al., 2022). An MT CPU has multiple thread contexts and interleaves the execution of instructions from different threads. As a result, if one thread is delayed by a memory access, other threads can continue to make progress (Rafi et al., 2022). However, the threads in a multithreaded process share the data, code, resources, and address space of their process. Thread usage in a program significantly reduces the overhead involved in creating and managing threads, as well as sharing per-process state information. Since thread creation has lower overhead, it is focused on single-process multithreaded programs (Cheikh et al., 2020).

The operating system must determine how to allocate the central processing units (CPUs) among the processes and threads in the system. In some systems, the operating system chooses a process to run, and the selected process chooses which of its threads will execute. Alternatively, the operating system schedules the threads directly. At any given moment, multiple processes, each containing one or more threads, may be executing. For example, some threads may be waiting for an I/O request to complete.

The scheduling policy determines which of the ready threads is chosen for execution. Generally, each ready thread is given a time slice (referred to as a quantum) of the CPU. If a thread decides to wait for something, it voluntarily gives up the CPU. Otherwise, when a hardware timer determines that a running thread's quantum has ended, an interrupt occurs, and the thread is preempted to allow another ready thread to run. If there are multiple CPUs, multiple threads can execute simultaneously. On a computer with a single CPU, threads appear to execute concurrently, even though they actually take turns running and may not receive equal time.

Therefore, some threads may appear to run at a faster rate than others do. The scheduling policy may also consider a thread's priority and the type of processing it performs, giving certain threads priority over others. It is assumed that the scheduling policy is fair, meaning that every ready thread eventually gets a chance to execute. The correctness of a concurrent program should not depend on the threads being scheduled in a specific order. Switching the CPU from one process or thread to another, known as a context switch, requires saving the state of the old process or thread and loading the state of the new one (Muthukrishnan et al., 2023). Since there may be several hundred context switches per second, context switches can potentially introduce significant

1.2 Thread States

A state diagram, occasionally referred to as a state machine diagram, is a form of behavioral diagram in the Unified Modeling Language (UML) that displays changes between different entities. A state machine diagram represents the actions of a solitary entity, detailing the series of occurrences that an entity experiences throughout its existence in reaction to occurrences. At any given moment, a thread is considered to be in one of numerous thread conditions - depicted in the UML state diagram in Figure 2.

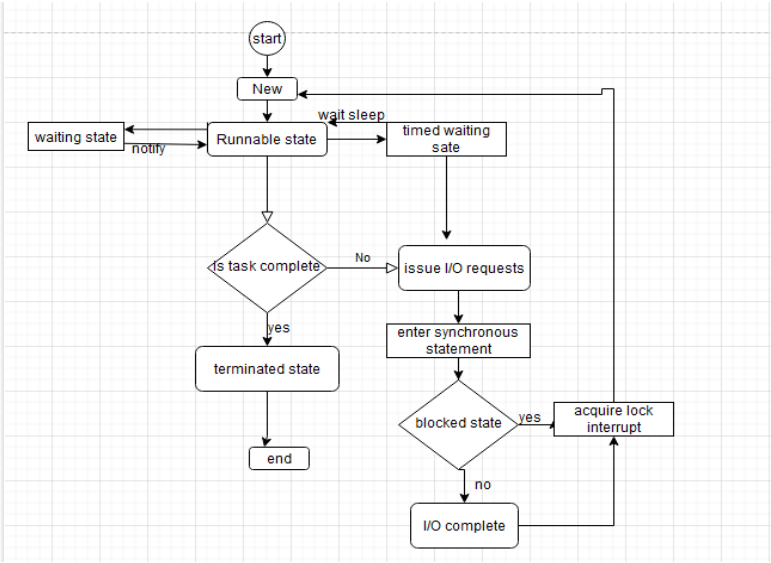

Figure 2. Life Cycle of a Thread. Source: Hague et al. (2022).

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

New State
Runnable State
Blocked State
Waiting State
Timed Waiting State
Terminated State

The state diagram of fig 2 above shows the different states in which the verification sub-system or class exist for a particular system

1.2.1 New and Runnable States

A fresh thread initiates its life cycle in the novel condition. It stays in this condition until the program commences the thread, which positions it in the ready state. A thread in the ready state is regarded as carrying out its duty.

1.2.2 Waiting State

Occasionally, a running thread shifts to the dormant state as it awaits for another thread to accomplish a task. A dormant thread shifts back to the running state solely when another thread alerts it to resume execution.

1.2.3 Timed Waiting State

A runnable thread that can be executed can enter the timed waiting state for a specified period. It returns to the executable state when that time ends or when the event it is waiting for occurs. Threads in timed waiting and waiting states cannot use a processor, even if one is available. A thread that can be executed can transition to the timed waiting state if it specifies a wait interval when waiting for another thread to complete a task. This thread returns to the executable state when another thread or when the timed interval ends - whichever happens first, notifies it. Another way to put a thread in the timed waiting state is to make a thread that can be executed sleep. A sleeping thread remains in the timed waiting state for a specified period (known as the sleeping interval), after which it returns to the executable state. Threads sleep when they temporarily have no work to do.

1.2.4 Blocked State

A runnable thread moves to the obstructed state when it tries to carry out a task that cannot be promptly finished and it needs to temporarily pause until that task is done. For instance, when a thread sends a request for input/output, the operating system obstructs the thread from running until that I/O request is completed. At that moment, the obstructed thread shifts to the running state, allowing it to continue execution**.**

1.2.5 Terminated State

A runnable thread enters the finished state (sometimes called the inactive state) when it successfully completes its task or otherwise stops, perhaps due to some error. Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A stop pseudo-state is represented as a cross. In unified markup language (UML) of the state diagram of figure 1, the finished state is followed by the UML, final state (the bull's eye symbol) to indicate the end of the state transitions.

Latency concealing: Provide each processor with useful work to do as it awaits the completion of memory access requests. Latency concealing provides the possibility of communications to be fully overlapped with computation, leading to high efficiency and hardware utilization. Multithreading may be a practical mechanism for latency. Multithreading is a useful mechanism for reducing latency. A multithreaded computation typically begins with a

sequential thread, then some supervisory overhead to set up (schedule) various independent threads, then computation and communication (remote accesses) for individual threads, and finally a synchronization step to stop the threads before starting the next unit (Fig 3).

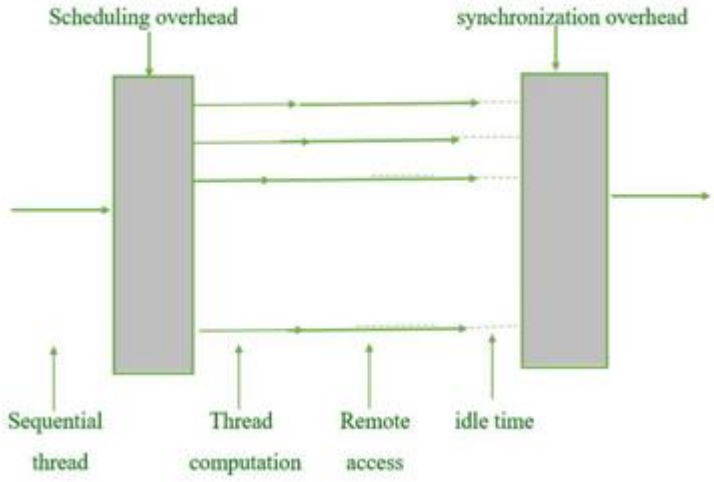

Figure 3. The Concept of Multithreaded Parallel Computation. Source: Author (2024).

2. Statement of the Problem

Though the human mind can perform functions simultaneously, individuals struggle to switch between parallel streams of thoughts. Occasionally, a running thread transitions to a waiting state while it awaits another thread to carry out a task. A waiting thread only returns to the running state when it is notified by another thread to continue execution. Another method to put a thread in the timed waiting state is to make a running thread go to sleep. A sleeping thread remains in the timed waiting state for a specified period (known as a sleep interval), after which it goes back to the running state. Threads sleep when they temporarily have no work to do.

For instance, a word processor might have a thread that periodically creates a backup (i.e., writes a duplicate) of the current document to the disk for recovery purposes. If the thread did not sleep between consecutive backups, it would need to have a loop that continuously checks whether it should write a copy of the document to the disk. This loop would consume CPU time without accomplishing any productive work, thereby decreasing system performance (Abellan et al., 2015).

Moreover, when multiple threads share an object and one or more of them modify it, unpredictable outcomes may arise. For instance, if one thread is in the process of updating a shared object and another thread also attempts to update it, it is unclear which thread's update will take effect. In such cases, the behavior of the program cannot be relied upon.

3. Related Works

In recent years, some augmented reality tasks have higher requirements for real-time performance when processing data. So, the traffic of mobile data continues to grow. It is not difficult to find that data requested by users is highly repetitive, which will lead to a large amount of redundant data transmission. In recent years, the caching problem has attracted the attention of researchers as a method to solve the delay problem (Vayadande et al., 2022). Cache is a new strategy to improve the performance and the service quality of mobile edge networks.

It includes offloading tasks to the mobile edge cloud and storing computation results in the local storage located at the edge of network. This technology avoids redundant and repetitive processing of the same task, thereby simplifying the offloading process and improving the utilization of network resources (Wu et al., 2022). These studies serve as basis for knowledge development, create guidelines for policy and practice, provide evidence of an effect, and engender new ideas and directives for this particular study. It provides a solid basis for understanding how concurrent applications

synchronize access to shared memory, the concepts are important to understand, even if an application does not use these tools explicitly.

As a new method to alleviate the unprecedented network traffic, mobile edge caching has been widely used in the wired internet, and it has proved that it can reduce delay and energy consumption (Jin et al., 2018). To date, many research works have focused on optimizing caching methods to solve the delay and energy consumption problems in computation offloading. In Kumar et al. (2023), the author considered the horizontal cooperation between mobile edge nodes for joint caching and proposes a new transformation method to solve the problem of edge caching and improve cache hit rate of the network.

In Ma et al. (2020) and Pramudita et al. (2020) authors designed a heterogeneous collaborative edge cache framework by jointly optimizing node selection and cache replacement in mobile networks. The joint optimization problem is expressed as a Markov Decision Process (MDP), and Deep Q Network (DQN) is used to solve the problem, which alleviates the offloading traffic load. These studies provide substantial practical and theoretical contributions, and help to identify the scope of this study.

In Chen et al. (2021), an approach, which is based on the users' interests, has been presented where media objects are organized logically according to their interests over the network. This approach assumes a peer, which has a media object that other peers are interested in, might have other media objects that they are also interested in. In the case of a cache miss, the request is forwarded to a peer that has the same interests (Cai et al., 2022). This section of the literature review examines factors influencing deadlock and analytical modeling, the experiments described in this section do not attempt to manipulate and test the factors that influence conformity, they help to understand the results obtained and consider the implication of the findings.

Thus, the overhead and network traffic is reduced. In Cheikh et al. (2021) and Dai et al. (2020), new approaches based on machine learning techniques have been proposed to improve the performance of traditional cache replacement policies such as least recently used (LRU), global distribution system (GDS), and least frequently used (LFU). However, these approaches have some disadvantages such as the additional computational overhead, which is needed for the target outputs preparation in the training phase when looking for future requests. Back Propagation Neural Network (BPNN) has been applied in Neural Network Proxy Cache Replacement (NNPCR) (Candelario et al., 2023) and NNPCR-2 (Fu et al., 2020) for making cache replacement decisions. However, BPNN performance is influenced by the optimal selection of the network topology and its parameters. These findings provide background context to the statistical analysis and contribute further to understanding deadlock effects in system performance.

Murthy & Rani (2022) and Fu et al. (2022) describe a model for multi-processor IPC based on cache miss rates (Pramudita et al., 2020. Their model of cache delays for single-threaded workloads uses a similar approach. The fundamental distinction of this work is that the model is a multithreaded processor that hides individual threads' memory latencies – this has not been addressed in Murthy & Rani (2022) study or elsewhere. This section provides context for the research identifies gaps in existing literature and ensures novelty.

Several recent studies have been performed on facemask detection during COVID-19, and those are presented in the literature. Deep learning-based approaches were developed by researchers to study the issue of facemask detection (Majumdar et al., 2021; Murthy; Rani, 2022). Address the issue of masked face recognition; it was developed as a reliable solution based on occlusion removal, as well as deep learning-based features (Kochol, 2023). Alex Net and VGG16 convolutional neural network designs are used as transfer learning for the development of new models (Madhi et al., 2018). However, all these studies focus on specific optimization or characterizations and stop short of providing a quantitative framework. They serve as basis for knowledge development and serve for future research.

Technology has been developed that prevents the spread of viruses and uses deep learning technology to ensure that people are wearing facemasks correctly. The Celeb dataset was used to develop a model to automatically remove mask objects from the face and synthesize the corrupt regions while maintaining the original face structure (Ma et al., 2022). A multi- threading strategy with VGG-16 and triplet loss Face Net dependent on the masked faced recognition approach is proposed, which is built on Mobile Net and Haar-cascade to detect facemasks (Raul et al., 2023). The embedding unmasking model (EUM) method was designed, which aimed to improve upon existing facial recognition models.

The self-restrained triplet (SRT) technique was utilized, which allowed EUM to produce embedding corresponding to the associated characters of unmasked faces (Srijone et al., 2021). The margin cosine loss (MFCosface) masked faced recognition algorithm, which is dependent on a wide margin cosine loss design, was proposed for detecting and identifying criminals. An attention-aware mechanism was improved by incorporating important facial features that

helps in recognition (Almutairi et al., 2023). This paragraph summarizes details of the paper's methodology and data that are relevant to this study. This has help to find the "sweet spot" for what this study has to focus on.

An attention-based component using the convolutional block attention module (CBAM) model was designed, which depends on the highlighted area around the eyes (Chen et al., 2021). The near infrared to visible (NIR-VIS) masked faced recognition problem was analyzed in terms of the training approach and data model (Yang et al., 2021). A method called heterogeneous semi-siamese training (HSST) was designed, which attempts to maximize the joint information between face representations using semi-siamese networks. The literature here introduces the background and defines the gaps this work aims to fill. It also helps to summarize details of the paper's methodology and data that are relevant to this study.

4. Methodology

4.1 Multi-threaded Issues (Thread Safe)

Multiple threads can execute the same block of code simultaneously or in parallel. It can be executed in a "Thread Safe" or "Not Thread Safe" manner. This research identifies the issues related to thread safety and explores methods to prevent them.

4.2 Thread Security

Thread security refers to the ability of multiple threads to access the same resources without causing unpredictable outcomes such as race conditions or deadlocks. For instance, when implementing a Singleton pattern for initializing a database connection, the idea is to create a database connection object only once and utilize that object whenever there is a need to interact with the database.

```
1.    //type global
2.  Type singleton map {string}string
3.
4.   var {
5.          Instance singleton
6.  }
7.
8.  Func new class( )   singleton {
9.
10.        If instance = = nil {
11.
12.          Instance=make{singleton} //  <...not thread safe
13.  }
14.
15.    Return instance
16.  }
```

A critical section is any piece of code that has the potential of being executed concurrently by more than one thread and sharing data or resources used by the application. The above code worked but it has a race condition at line 10. Assuming multiple threads run simultaneously on line 10, threads "compete" to read the instance variable, it becomes nil at that time and then multiple threads initialize connection with the database, which might consume all the connection pool of the database. Output in a race condition is difficult to predict and inconsistent.

4.3 Mutual Exclusions (thread Safe using Mutex)

There are several methods for avoiding race conditions to achieve thread safety. The first method is Locking or Mutex. Mutex, as the name implies Mutual exclusion, only one thread has exclusive access permission and blocks others from

accessing the resource. It allows all the threads to be able to use the resource but only one process is allowed to use it at a time.

```
1.  var lock- & sync. Mutex { }
2.
3.  // type global
4.  Type singleton map[string]string
5.
6.   var {
7.   Instance singleton
8.  }
9.
10.  func new class ( ) singleton {
11.
12.          Lock .lock ( )
13.     Defer lock.  Unlock ( )
14.
15.     If instance = = nil {
16.
17.        Instance = make (singleton)  //<... thread safe
18.  }
19.
20.   Return instance
21   }
```

For instance, Thread A executes in line 12, it attempts and successfully obtains a lock, then it proceeds to line 17 and generates a singleton database connection object. Meanwhile, when Thread B executes on line 12, it must wait to obtain the lock since Thread A is holding it. After Thread A returns in line 20, it releases the obtained lock in line 13 (defer keyword will move the execution of the statement to the very end inside a function). Until then, Thread B can successfully obtain the lock in line 12 and verify if the instance variable is nil, since it has already been assigned by Thread A, it won't initialize the singleton object again. Then it releases the obtained lock as depicted in (Figure 4).

4.4. Methodology 2

4.4.1 Creating and executing threads

The recommended way to develop multithreaded Java applications is by utilizing the Runnable interface from the java package. A Runnable object represents a task that can be executed simultaneously with other tasks. The runnable interface defines the sole method Run, which contains the code that specifies the task that a runnable object should carry out. When a thread executing a Runnable is initialized, the thread invokes the Run method of the Runnable objects, which runs in the newly created thread (Kumar et al., 2022).

4.4.2 Runnable and the Thread Class

```
1.  //fig.2: print //print task .java
2.  // Print task class sleeps for a random time from 0 to 5 seconds
3.  Import java.util.Random;
4.
5.  Public class printTask implemets Runnable
```

```
6.  {
7.  Private final int sleep time; // random sleep time for thread
8.  Private final string taskname; //name of task
9.  Private final static random generator = new random ( );
10.
11. Public printTask (spring name)
12. {
13. Task name = name; // set task name
14.
15. // pick random sleep time between 0 and 5 seconds
16. Sleep time = generator. nextInt (5000); // milliseconds
17. }  //  end printTask constructor
18.
19. // method run contains the code that a thread will execute
20. Public void run( )
21. {
22. Try // put thread to sleep for sleepTime amount of time
23. {
24. System .out.print{"%s going to sleep for %d milliseconds.\n"
25.         Taskname, sleep};
26.         Thread.sleep{sleeptime}; // put thread to sleep
27.        } // end try
28.      Catch { interruptedExecution exception}
29. {
30.      System.out.printf( "%s done sleeping\n", tasktime};
31.              Terminated prematurely due to interrrption"};
32. } // end catch
33.
34.      //print task name
35.      System.out.printf{"%s done sleeping\n" , taskname };
36.      }  //end method run
37. } // end class printTask
```

Utilize the runnable interface (line 5), so that multiple print tasks can execute concurrently. The variable sleep time (line 7) holds a random integer value from 0 to 5 seconds generated in the constructor of the print task (line 16). Each thread running a print task sleeps for the time specified by the sleep time, then prints its task's name and a message indicating that it has finished sleeping.

A print task is executed when a thread calls the run method of the print task. Lines 24-25 display a message indicating the name of the currently executing task and that the task is going to sleep for sleep time milliseconds. Line 26 calls the sleep method of the Thread class to put the thread in a timed waiting state for the specified amount of time. During this time, the thread relinquishes the processor, allowing another thread to execute. When the thread wakes up, it reenters the runnable state.

When the print task is assigned to a processor again, line 35 prints a message indicating that the task has finished sleeping, and then the run method terminates. Note that the catch block at lines 28-32 is necessary because the sleep method may throw a checked exception if the sleeping thread is interrupted (Almutairi et al., 2023).

5. Experimental Evaluation

The structure was a large-scale, cache-consistent, multi-processor system. It comprises of multiple multiprocessor clusters connected through a scalable, low delay interconnection network. Physical memory was spread out among the processing nodes in different clusters. The dispersed memory formed a worldwide address space. For each memory block, the distant directory kept record of remote nodes caching it. When a write occurred, point-to-point messages were sent to invalidate remote copies of the block.

Acknowledgement messages were used to notify the originating node when an invalidation was completed. Multi-threading requires that the processor be designed to handle multiple contexts simultaneously on a context-switching basis. Figure 4 below specified the typical architecture environment using multiple-context processors (P) and memory (M) as shown below. The dispersed memory forms a worldwide address space. Four machine parameters are defined to analyze the performance of this network: Latency (L), number of thread (N), context switching overhead (C) and the interval between switches (R). Two levels of local cache were used per processing nodes. Loads and write were separated with the use of write-buffers for implementing weaker memory consistency models (Figure 4).

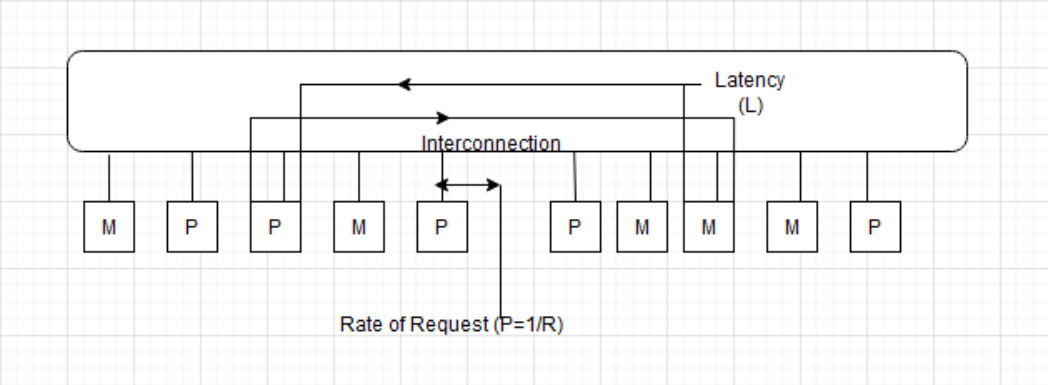

Figure 4. Architecture Environment. Source: Siqueia & Kreutz (2018).

The distributed memories form a global address space. Four machine parameters are specified below to analyze the performance of this network. The multithreading processor will suspend the current context and switch to another, so after a fixed number of cycles it will be busy again doing useful work, even if the remote reference is lost. Only when the whole context hangs do the processor go idle (Cai et al., 2022).

The goal here is to maximize the fraction of the time the processor is busy. Processor efficiency is a performance metric, given by,

$$Efficiency = \frac{busy}{(busy + switching + idle)} \qquad 1$$

The busy, switching, and idle cases represent time, measured over a substantial period. The basic idea behind multi-threaded machine is to interleave execution of multiple contexts to greatly reduce idle value, but not to increase the level of conversion too much. The instruction queue unit has a cache, which stores certain instructions according to the instruction specified by the program counter. The minimum required buffer size is:

$$B = N*C \text{ words} \qquad 2$$

Where N is the number of thread slots and C is the number of cycles required to access the instruction cache.

5.1 Processor efficiencies

The single-threaded processor executes the context until the remote reference is given (R cycle), then waits until the reference completes (L cycle). There is no context switching and obviously no switching overhead. Let us model this behavior as an alternate innovation process with an R+L cycle. R and L correspond to the duration of a cycle during which the processor is busy and idle, respectively. The efficiency of a thread server is given by:

$$E_1 = \frac{R}{R+L} = \frac{1}{1+L/R} \qquad 3$$

The equation shows performance degradation of such a processor in a parallel system with large memory latency (Wu et al., 2022).

Saturation region: In this saturated region, the processor operates with maximum utilization. The cycle of the renewal process in this case is R+C, and the efficiency is simply:

$$\boldsymbol{E_{sat} = \frac{R}{R+C} = \frac{1}{1+C/R}} \qquad 4$$

That is what to say. Saturation efficiency does not depend on the delay and also does not change as the number of contexts increases.

Liner - When the number of contexts is lower than the saturation point, there may be no contexts after the context switch, so the processor will go through cycles of inactivity. The time it takes to switch to the ready context, execute the context until the remote reference is emitted, and handle the reference using R+C+L. Assume N is below the saturation point, whereas all other contexts have a variation in the processor. Efficiency is given by:

$$\boldsymbol{E_{lin} = \frac{NR}{R+C+L}} \qquad 5$$

i.e.… the efficiency increases linearly with the number of contexts until the saturation point is reached and beyond that remains constant (Candelario et al., 2023).

```
Import time
Import mat plot lib.pyplot as plt
# measure the time taken by the normal method function
Start time = time.times ( )
Contents_normal = download _ all _ urls (urls)
End _time = time. Time ( )
Time_ normal = end_time – start _ time
# print the time taken by the normal function
Print ('time take by the normal method :{:2f} seconds', format ( time _ normal)
#measure the time taken by the multithreading function
Start time = time . time ( )
Contents _ Multithreading = download _ all _ urls _ multithreading (urls)
End _ time = time . tme( )
Time _ multithreading = end _ time – start _ time
#print the time taken by the multithreading function
```

```
Print ('time taken by multithreading method: {:.2f} seconds', format (time _ multithreading)
# set plot size
Plt . figure (figsize = (8,6))
# create a bar chart
Labels = ['normal method', 'multithreading method']
Times = [' time_method',time _ multithreading']
Plt. Bar (labels, times)
#add annotations to the chart
Plt, title ('time taken to download contents of all urls')
Plt. X label ('method')
Plt. Y label ('time (seconds'))
Plt. Y lim (top = max (times+5)
    for i, v in enumerate (times)'
plt.text (I,v+2, (:.2f), s.format (v), ha = 'center', va = 1(bottom')
# display the chart
Plt.show ( )
```

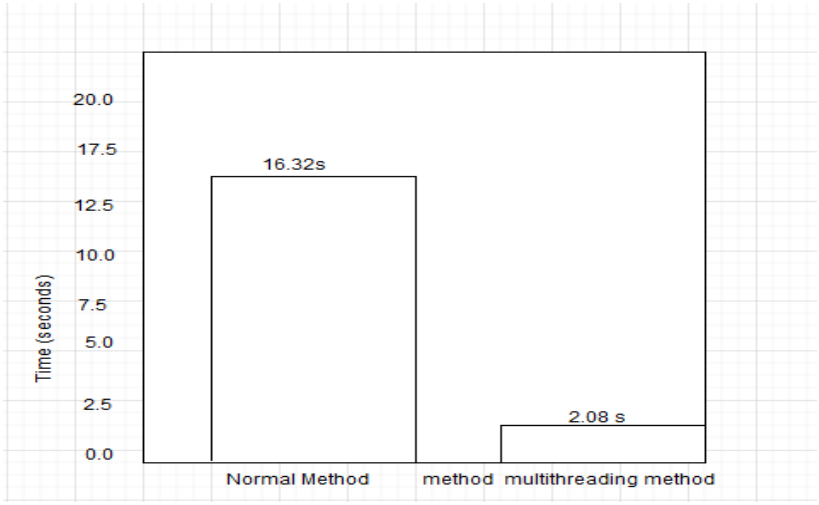

Figure 5. Time taken to download content of all URL. Source: Yang et al. (2021).

6. Result and Discussion

This study experimentally evaluated the performance of a multithreaded architecture using a network model consisting of clusters of multiprocessors (P) and physical memory (M) distributed among nodes, processed in different clusters, as shown in (Figure 4). Distributed memory forms a global address space. Confirmation messages were used to notify the root node when the deactivation was complete. Two levels of local cache were used for the processing nodes. Load and write have been separated with write cache to implement weaker memory consistency models. In multiprocessors using the ownership-based caching association protocol, if a line of cache blocks needs to be modified, direct prefetching with ownership can significantly reduce write latency and secure network traffic to get ownership.

Caching helps to hide latency by increasing the cache hit rate for read operations (Tang et al., 2021). When a cache failure occurs, cycles are wasted. The benefit of using a distributed cache is to reduce buffer misses. In the test screen, it shows that despite the lock contention, the processors have progressed at approximately their relative reserve speed. Thus, a clear advantage of using multithreading is demonstrated in the test.

To support this point, Figure 5 shows the time distribution between frames (methods) exposed by the X server to a client thread using Unix domain sockets, since server x runs at much lower throughput against its client, several

frameworks (methods) have been pointed out. For multi-threaded trains, higher latency was not observed because the server operates with the same customer-provided reservation. This further strengthens the lemma that multithreading can speed up performance through parallelism, a program using entirely two CPUs can run in almost half the time.

7. Conclusion

The motivation of this study was to improve the performance of several important modern applications that use a lot of memory that are known to cause frequent CPU crashes. The lack of practical experience with multithreaded architectures suggests that we don't have a complete understanding of how these processors work and the best way to design them. Having more hardware context increases the possibility of latency hiding. On the other hand, not having enough buffers can make memory latency so high that multithreading won't be able to fix it. Threads that exhibit poor memory location, such as database applications, are often blocked while waiting for a response from the memory hierarchy:

These threads are memory intensive. This observation suggests a better way of handling pipeline conflicts. To evaluate the effect of memory latency on processor performance, a dual-core MT machine with four thread contexts per core was used. The workload includes nine benchmarks from the 2000 CPU suite. These specific benchmarks were chosen to allow the workload to include programs with both good and bad cache locality. Two copies of each benchmark were run, resulting in a workload of 18 threads (each benchmark is a single-threaded process running in its own thread). Runtime-oriented simulations and benchmarks are run in the Solaris™ 9 operating environment. Solaris™ 9 does not include special support for Chip Multithreading (CMT) processors.

From an operating system perspective, a quad-core multithreaded machine emulates the same as a regular eight-way multiprocessor. The Solaris™ scheduler assigns threads to the hardware context as if it were assigning them to processors on a multiprocessor system, selecting eight threads at a time to schedule them according to the corrected modeling method change. When modeling a multithreaded environment, statistics are collected when multiple threads in the kernel are active, but the input request-modeling engine is a CPU service request when only one thread is active in the CPU.

This requires calculations on the collected statistics. Execution time is reduced by a third when using threads compared to executing functions sequentially. That's the power of multithreading. Multithreading is useful for I/O-bound processes, such as reading files from a network or a database, because each thread can simultaneously execute an I/O-bound process. Note that using multithreading for CPU-bound processes can slow performance since competing resources ensure only one thread can run at a time, and the overhead is incurred in dealing with multiple threads.

8. Acknowledgement

I would like to express my sincere gratitude to my supervisor, Professor Akinwale A.T., Federal University of Agriculture Abeokuta, Nigeria, for his guidance and support throughout the research process. His expertise and insights are invaluable in shaping the research and helping to overcome challenges.

I am also grateful to IAEC- University for providing me with resources and support to complete this study.

Finally, I would like to thank my family and friends for their encouragement and support throughout the research process. Without their support, I would not have been able to complete this research study.

9. Author Contributions

The author confirms sole responsibility for the following: study conception and design, data collection, analysis and interpretation of results, and manuscript preparation.

10. Conflicts of Interest

No conflict of interest.

11. Ethics Approval

Not applicable.

12. References

Abellán, J. L., Fernández, J., & Acacio, M. E. (2015). Efficient Hardware-Supported Synchronization Mechanisms for Manycores. *Handbook on Data Centers*, 753-803. https://doi.org/10.1007/978-1-4939-2092-1_26

Adam, G. K. (2022). Co-Design of Multicore Hardware and Multithreaded Software for Thread Performance Assessment on an FPGA. *Computers*, 11(5), 76. https://doi.org/10.3390/computers11050076

Almutairi, S. Z., Mohamed, E. A., & El-Sousy, F. F. (2023). A Novel Adaptive Manta-Ray Foraging Optimization for Stochastic ORPD Considering Uncertainties of Wind Power and Load Demand. *Mathematics*, 11(11), 2591. https://doi.org/10.3390/math11112591

Cai, C., Wang, L., & Ying, S. (2022). Symmetric diffeomorphic image registration with multi-label segmentation masks. *Mathematics*, 10(11), 1946. https://doi.org/10.3390/math10111946

Candelario, G., Cordero, A., Torregrosa, J. R., & Vassileva, M. P. (2023). Solving Nonlinear Transcendental Equations by Iterative Methods with Conformable Derivatives: A General Approach. *Mathematics*, 11(11), 2568. https://doi.org/10.3390/math11112568

Carvalho, T., Pinto, J. B., & Cardoso, J. M. P. (2023). A DSL-based runtime adaptivity framework for Java. *SoftwareX*, 23, 101496. https://doi.org/10.1016/j.softx.2023.101496

Cheikh, A., Sordillo, S., Mastrandrea, A., Menichelli, F., & Olivieri, M. (2020). Efficient mathematical accelerator design coupled with an interleaved multi-threading RISC-V microprocessor. *In:* Applications in Electronics Pervading Industry, Environment and Society: APPLEPIES 2019 7 (pp. 529-539). Springer International Publishing. https://doi.org/10.1007/978-3-030-37277-4_62

Chen, X., Diaz-Pinto, A., Ravikumar, N., & Frangi, A. F. (2021). Deep learning in medical image registration. *Progress in Biomedical Engineering*, 3(1), 012003. https://doi.org/10.1088/2516-1091/abd37c

Dai, A., Zhou, H., Tian, Y., Zhang, Y., & Lu, T. (2020). Image registration algorithm based on manifold regularization with thin-plate spline model. In Neural Computing for Advanced Applications: First International Conference, NCAA 2020, Shenzhen, China, July 3–5, 2020, *Proceedings 1* (pp. 320-331). Springer Singapore.

Eslamimehr, M., Lesani, M., & Edwards, G. (2018). Efficient detection and validation of atomicity violations in concurrent programs. *Journal of Systems and Software*, 137, 618-635. https://doi.org/10.1016/j.jss.2017.06.001

Fu, Z., Chu, S. C., Watada, J., Hu, C. C., & Pan, J. S. (2022). Software and hardware co-design and implementation of intelligent optimization algorithms. *Applied Soft Computing*, 129, 109639. https://doi.org/10.1016/j.asoc.2020.109639

Hassanein, A., El-Abd, M., Damaj, I., & Rehman, H. U. (2020). Parallel hardware implementation of the brain storm optimization algorithm using FPGAs. *Microprocessors and microsystems*, 74, 103005. https://doi.org/10.1016/j.micpro.2020.103005

Jin, Q., Xu, Z., & Cai, W. (2021). An improved whale optimization algorithm with random evolution and special reinforcement dual-operation strategy collaboration. *Symmetry*, 13(2), 238. https://doi.org/10.3390/sym13020238

Kochol, M. (2023). Linear Algebraic Relations among Cardinalities of Sets of Matroid Functions. *Mathematics*, 11(11), 2570. https://doi.org/10.3390/math11112570

Kumar, B. A., & Bansal, M. (2023). Face mask detection on photo and real-time video images using Caffe-MobileNetV2 transfer learning. *Applied Sciences*, 13(2), 935. https://doi.org/10.3390/app13020935

Vayadande, K., Shaikh, R., Narnaware, T., Rothe, S., Bhavar, N., & Deshmukh, S. (2022). Designing Web Crawler Based on Multi-threaded Approach For Authentication of Web Links on Internet. *In:* 2022 6th International Conference on Electronics, Communication and Aerospace Technology (pp. 1469-1473). IEEE. https://doi.org/10.1109/ICECA55336.2022.10009614

Ma, X., Wu, S., Pobee, E., Mei, X., Zhang, H., Jiang, B., & Chan, W. K. (2020). Regiontrack: A trace-based sound and complete checker to debug transactional atomicity violations and non-serializable traces. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1), 1-49. https://doi.org/10.1145/3412377

Ma, X., Wu, S., Pobee, E., Mei, X., Zhang, H., Jiang, B., & Chan, W. K. (2020). Regiontrack: A trace-based sound and complete checker to debug transactional atomicity violations and non-serializable traces. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1), 1-49. https://doi.org/10.1145/3412377

Ma, X., Ashraf, I., & Chan, W. K. (2022). Davida: A Decentralization Approach to Localizing Transaction Sequences for Debugging Transactional Atomicity Violations. *IEEE Transactions on Reliability*, 72(2), 808-826. https://doi.org/10.1109/TR.2022.3176680

Majumdar, S., Chatterjee, N., Das, P. P., & Chakrabarti, A. (2021). A mathematical framework for design discovery from multi-threaded applications using neural sequence solvers. *Innovations in Systems and Software Engineering*, 17(3), 289-307. https://doi.org/10.1007/s11334-021-00393-8

Moragues, R., Aparicio, J., & Esteve, M. (2023). Ranking the Importance of Variables in a Nonparametric Frontier Analysis Using Unsupervised Machine Learning Techniques. *Mathematics*, 11(11), 2590. https://doi.org/10.3390/math11112590

Murthy, P. V. R., & Rani, N. (2022). Testing Multi-Threaded Programs by Transformation to Hoare's CSP. *In:* 2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon) (pp. 1-7). IEEE. https://doi.org/10.1109/QRS54544.2021.00022

Muthukrishnan, H., Lustig, D., Villa, O., Wenisch, T., & Nellans, D. (2023). FinePack: Transparently Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems. *In:* 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) (pp. 516-529). IEEE. https://doi.org/10.1109/HPCA56546.2023.10070949

Pramudita, Y. D., Anamisa, D. R., Putro, S. S., & Rahmawanto, M. A. (2020). Extraction System Web Content Sports New Based On Web Crawler Multi Thread. *In:* Journal of Physics: Conference Series (Vol. 1569, No. 2, p. 022077). IOP Publishing. https://doi.org/10.1088/1742-6596/1569/2/022077

Rafi, M. E. H., Williams, K., & Qasem, A. (2022). Raptor: Mitigating CPU-GPU False Sharing Under Unified Memory Systems. *In:* 2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC) (pp. 1-8). IEEE. https://doi.org/10.1109/IGSC55832.2022.9969376. 978-1-6654-6550-2

Siqueira, H., & Kreutz, M. (2018). A simultaneous multithreading processor architecture with predictable timing behavior. *In:* 2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 62-66). IEEE. https://doi.org/10.1109/SBESC.2018.00018

Tang, H., Li, C., Zhang, Y., & Luo, Y. (2021). Optimal multilevel media stream caching in cloud-edge environment. *The Journal of Supercomputing*, 77, 10357-10376. https://doi.org/10.1007/s11227-021-03683-x

Vijaykumar, N., Olgun, A., Kanellopoulos, K., Bostanci, F. N., Hassan, H., Lotfi, M., ... & Mutlu, O. (2022). MetaSys: A practical open-source metadata management system to implement and evaluate cross-layer optimizations. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(2), 1-29. https://doi.org/10.1145/3505250

Wu, J., Fan, C., Li, G., Xu, Z., Jin, Z., & Zheng, Y. (2022). A New Heuristic Computation Offloading Method Based on Cache-Assisted Model. *Wireless Communications and Mobile Computing*, 2022, 3501329. https://doi.org/10.1155/2022/3501329

Yang, C., Ashraf, I., Ma, X., Zhang, H., & Chan, W. K. (2021). OPE: Transforming Programs with Clean and Precise Separation of Tested Intraprocedural Program Paths with Path Profiling. *In:* 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) (pp. 279-290). IEEE. https://doi.org/10.1109/QRS54544.2021.00039

Funding

Not applicable.

Institutional Review Board Statement

Not applicable.

Informed Consent Statement

Not applicable.

Copyrights