Context-Based Synchronization of Concurrent Process Using Aspect-Oriented Programming

Ogheneovo, E. E., Asagba, P.O. and Ejiofor, C. I.
Department of Computer Science,
University of Port Harcourt, Port Harcourt, Nigeria.
edward_ogheneovo@yahoo.com, pasagba@yahoo.com, ejioforifeanyi@yahoo.com

Abstract

Resource access synchronization within concurrent processes or threads is implemented using various constructs such as semaphores, monitor locks, Mutex, etc. The algorithm supporting most of these structures works by keeping at bay all other concurrent processes or threads till the current process accessing the resource has successfully relinquished the resource. This works very well as race conditions are controlled and shared data state remains consistent. The problem with this approach is performance in terms of system response. When each thread has to wait for the other to finish accessing the resource before it can proceed, a long line waiting threads can easily build-up, which obviously translates to relatively slow system response. In this paper, we propose contextual synchronization model to avoid where applicable, the unnecessary build-up of threads waiting for access to the resource. This model describes different contexts within which a resource access can be executed. Each model is ascribed different priorities of which different policies were applied. The most important feature of this model is that the context representing plain resource access will not cause any race condition if all other threads are accessing from the same context. The result of our experiment shows that context-based synchronization performs better than Java given the same number of threads.

Keywords: Aspect-oriented programming, synchronization, resource, and concurrent process.

Introduction

In Java programs, synchronization is commonly referred to as the coordination of multiple threads in accessing shared program states. As concurrency becomes a common programming practice in the multi-core era, the designers of concurrent programs are faced with many choices of synchronization mechanisms such as the use of locks, atomic blocks (Ben-Ari et al., 2006), and more recently, software transactional memory (Miller, S. K. 2001; Miller, M. S. 2006, Miller et al., 2003). For their distinctive operational differences, clear functional trade-offs exist among these synchronization mechanisms. This is problematic for building general-purpose and reusable Java systems. In conventional approaches, synchronization mechanisms are "hardwired" to the application logic through the use of library APIs or specialized language constructs. At the same time, choosing the most appropriate mechanism is increasingly about how reusable systems are being integrated in diversified comparison contents.

Context-based synchronization does not refer to any specific synchronization architecture, but to a method of applying synchronization. It distinguishes between

three (3) major reasons for performing resources access synchronization:

- Data/resources retrieval
- Data/resource modification
- Priority based access synchronization.

In this paper, we proposed a contextual model for synchronizing concurrent process using Aspect-oriented programming (AOP) based on the kind of access an action is specified to perform. This is to ensure unnecessary build-up of threads waiting for access to the resource thereby avoiding a long link of waiting threads which will result in a relatively slow system response. The model describes different contexts within which a resources access can be executed. Each model is ascribed different priorities of which different policies are applied. The most important feature is that the context representing plain resource access will not cause any race condition if all other threads are accessing from the same context.

Concurrent Process, Multi-Threading Programming and AOP

Concurrent Process

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel (Ben-Ari, 2006). Current programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network. The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinated among access to resources that are shared among processes.

A number of different methods can be used to implement concurrent programs, such as implementing each computational process as an operating system process, or implementing the computational processes as a set of threads within a single operating system process. In some concurrent computing systems, communication between the concurrent components is hidden from the programmer (e.g., by using futures), while in others it must be handled explicitly. Explicit communication can be divided into two classes:

- **Shared memory communication:** Concurrent components communicate by altering the contents of shared memory locations. This style of concurrent programming usually requires the application of some form of locking (e.g., mutexes, semaphores, or monitors) too coordinate between threads.

- **Message passing communication:** Concurrent components communicate by exchanging messages. The exchange of messages may be carried out asynchronously, or may use a rendezvous style in which the sender blocks until the message is received. Asynchronous message passing may be reliable or unreliable. Message-passing concurrency tends to be far easier to reason with than shared-memory concurrency, and is typically considered a more robust form of concurrent programming. Shared memory and message passing concurrency have different performance characteristics, typically, the per-process memory overhead and task switching overhead is lower in a message passing itself is greater than for a procedure call. These differences are often overwhelmed by other performance factors [23].

Multi-Threading Programming

In a network environment, it is a common practice for resources to be

shared among multiple users. Modern operating systems are usually designed to process multiple jobs (programs) at the same time. This is often referred to as multi-tasking. Multi-tasking results in effective and simultaneous utilization of various system resources such as processors, disks, and printers. Thus multiple tasks can be executed concurrently [2]. Java as an object-oriented programming language supports multithreading. Threads are dispatch able unit of work. They are light-weight processes within a process. A process is a program in execution. It consists of a number of independent units known as threads. A process is the collection of one or more threads and associated system resources. However, while a process depends on the architectural constructs of an application, a thread is a coding construct that does not affect the architecture of an application. All threads within a process share the same state and same memory space, and can communicate with each other directly, since they share the same variables.

Threads are an inherent part of software products as a fundamental unit of CPU utilization as a basic building block of multithreaded systems [22]. The use of threads has evolved over the years from each program consisting of a single thread as the path of execution of it. Threads are objects in Java programming language. They can be created using two different mechanisms.

- Create a class that extends the standard thread class
- Create a class that implements the Standard Runnable interface

Thus a thread can be defined by extending the java.lang.Thread class or by implementing the java.lang.Runnable interface. The Java programming language uses a thread to do garbage collection in the background thereby saving programmers the trouble of managing memory. Graphical user interface (GUI) programs have a separate thread user interface events from the host operating environment.

The notion of multithreading is the expansion of the original application thread to multiple threads running in parallel handling multiple events and performing multiple tasks concurrently [9]; Akhter and Roberts, [1]. Multithreading brings a higher level of responsiveness to the user as a thread can run while other threads are on hold awaiting instruction [12]. Multithreaded programs extend the notion of multitasking by taking it one level lower. Individual programs will appear to do multiple tasks at the same time. Each task is usually called a thread. Programs that can run more than one thread at once are called multithreaded. Therefore, for multithreading to be beneficial, the runtime of each individual thread must be long compared to the time it would take to switch between them.

Kerns [10] highlight the benefits of multithreading to include:

- High speed
- Small size
- More efficient in communication
- Recourse sharing

Aspect-Oriented Programming (AOP)

Object-oriented analysis, design, and programming (OOADP) is an old paradigm in software development and it has been proven successful in both small and large projects. As a technology, it has gone through its childhood and is moving into a mature adult stage. Research by educational establishments as well as audits by companies have shown that using object-oriented programming (OOP) instead of functional-decomposition techniques has dramatically enhanced the state of software. The benefits of using

object-oriented technologies in all phases of software development process vary. These include:

- Reusability of components
- Modularity
- Less complex implementation
- Reduced cost of maintenance

Each of these benefits will have varied importance to developers. One of them, modularity, is a universal advancement over structured programming that leads to cleaner and better understood software.

As global digitization and the size of applications expand at an exponential rate, software engineering's complexities are also growing. One feature of this complexity is the repetition of functionality—such as security, memory management, resource sharing, and error and failure handling—through and application. To address this issue, software researchers, pioneered by Gregor Kickzales [11] developed the aspect-oriented paradigm at Xerox Palo Alto Research Centre (PARC). Aspect-oriented programming is concerned with the identification of concerns, reminiscent of modularization, that are found in various parts of a programming and the effective management and reuse of the associated code [21]. Breaking programs into modules is present in some form in most if not all programming languages [17] identified the benefits of modularization which include: the reduction of development time because of the divide and conquer approach, and increasing software flexibility and understanding

Aspect-oriented programming supports two fundamental goals:

- Allow for the separation of concerns as appropriate for a host language
- Provide a mechanism for the description of concerns that crosscut other components

AOP is not meant to replace OOP or other object-based technologies. Instead, it supports the separation of concerns, typically using classes, and provides a way to separation aspects from the components. Aspect-oriented programming enables the representation of a concern by an aspect which is semantically tangled or scattered. In this sense, AOP paradigm extends OOP paradigm and AspectJ extends Java [18].

Conventionally, when code is scattered in different fragments throughout a program, it is hard to see its structure and hard to get a good view of the apparent tangling of the code. It is hard to change such code efficiently and hard to find all the cases that have to be changed. In commonly employed code, there are elements that are secondary to the primary functionality of the code. These elements though non-primary are vital to the proper execution of the code. Furthermore, they may be so scattered throughout the system that they contribute to the complexity of the code. These elements are called aspects. Examples of aspects include: security, fault-tolerance and synchronization. Aspect-oriented programming tires to isolate aspects into separate modules so that the isolate aspects into separate modules so that the resultant client-code is more purpose-specific, more reusable, and less tangled. It accomplishes this by a process of interception—intercepting function calls and managing their execution.

AOP Concepts

The main concepts of AOP are:

- **Join:** A join point is a particular location in the flow of the program instructions (e.g., beginning or end of a method execution, field's read or write access). It is a well-defined point in the base program (component language) that can be identified by an aspect. Join points may include calls to a method, a

conditional check, a loop's beginning or an assignment [7].

- **Advice:** These are methods that are activated when precise join points are reached, i.e., the mechanism of weaving inserts in the initial code the advices calls either in a static method (done at compile-time) or in a dynamic method (done at execution time). Advic**e** **c**an execute before, or around the join point).

- **Aspects:** An aspect is a special module which allows the association between advices and join points by means of point-cuts. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways ([11]. Examples of aspects include: memory access patterns and synchronization of concurrent objects.

- **Point-cuts:** They are used to define a set of join points on which will have to activate an advice. A point-cut allows easy capturing of the execution context of join points. For example, in a method call, this context includes the target object, the arguments of the method and the reference of the returned object, as many information of most useful for the injection of mechanism of traces.

- **Cross-cutting concerns:** A concern is a particular goal, concept, or area of interest; it means that it is in substance semantic concern. From the structural point of view, a concern may appear in the source code (Kiczales et al., 1997; Forgáč and Kollár, 2007; and Popovici et al., [18]. Cross-cutting concerns are elements of software, which cannot be expressed in any functional unit of the programming language's abstraction. In object-oriented programming parlance, cross-cutting concerns are elements of an application which cannot be cleanly captured in a method or class and so has to be scattered across many classes and methods. Such concerns include: design patterns, synchronization policies, exception handling, error-checking or fault tolerance concerns, resource sharing, security issues, performance, etc.

- **Weaving:** Weaving is the process of composing different functional modules and aspects according to the specifications given in the aspects. Weaving results in the behaviour of those functional modules impacted by aspects being modified accordingly.

Methodology

Object-oriented based system model

The object-oriented based system model describes the conceptual framework of contextual synchronization functionalities which are viewed more or less as resources. Each of these resources have properties (exposed through get and set methods), or methods whose operations effect the state of the object in ways that would be compromising if invoked simultaneously from multiple threads. The framework classifies a specific number of contexts that define the different scenarios that can occur when considering asynchronous invocation or accessing of resource properties. These contexts are classified based on the overall effect they have on the state of the resource object. The analysis class diagram is as shown in figure 1 below.

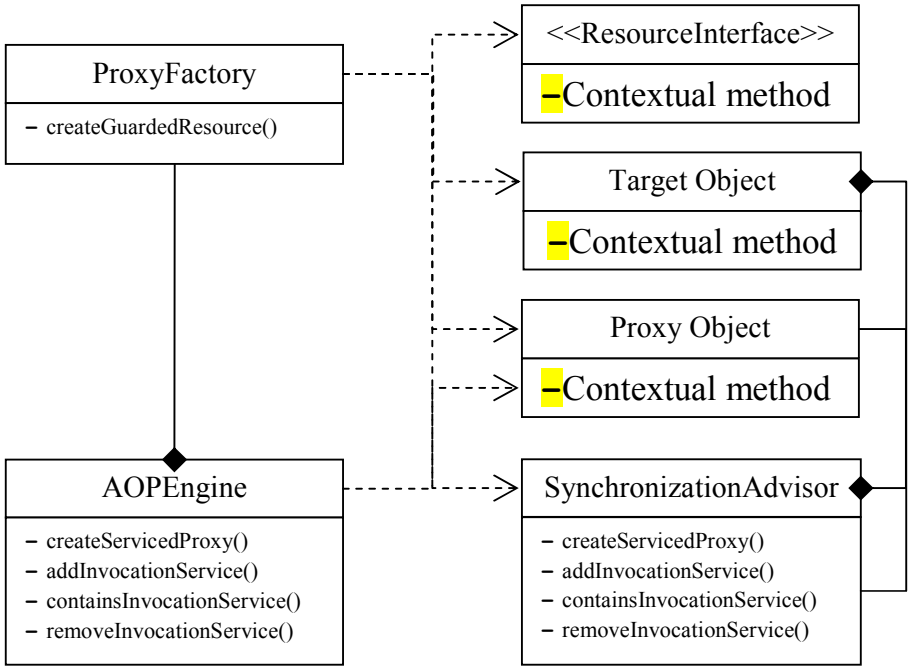

Fig. 1: Class diagram of the conceptual model

The analysis classes identified from the conceptual model are as follows:

- Proxy factory
- AOP engine
- Synchronization advisor
- Resource interface
- Resource object

Proxy Factory

The proxy factory class is responsible for creating proxies out of a combination of a resource interface, and the implementation resource object. It would be responsible for initializing the AOP engine and synchronization advisor with information about the proxy about to be created.

AOP Engine

The AOP engine is a class that encapsulate operations and logic necessary for advising target object methods, exception, etc. Naturally, the engine should be generic enough for objects to register as advisors to a target object.

Synchronization Advisor

This is an instance of an advice that the AOP engine dynamically applies to proxies. Since in the case of this paper, advice is limited to the synchronization advisor, advice flexibility/scalability will be kept to a minimum. The synchronization advisor implements the rules described in the **conceptualization phase.**

Resource Interface

This interface exposes the methods that are serviced by the synchronization have to be repeated in the resource interface.

Resource Object

This is any object that explicitly extends the resource interface. A resource generally represents any entity to which

synchronization is to be applied to. This is if the interface extends other interfaces whose functionality, exposed through its methods, aer to be serviced by the synchronization advisor, such methods

Architectural Design

Proxy generation

The proxy generation module is made of a single class. This class employs the facilities of the AOP Engine to create the proxy class that encapsulates the target resource as well as provide the contextual – synchronization services. The single method exposed statically by the class is: create guarded resource. This method accepts a single parameter, the resource object, polymorphically as a serializable interface (to support serialization). This logic within this method attempts to extract and store in an array, all interfaces, implemented by the argument object that is annotated with the "GuardedResource" annotation. Once this is done, the AOP ProxyFactory class is called upon to create the actual proxy out of these annotated interfaces. State-wise though, the ResourceAccessController processes two static properties (i) an instance of the actual object whose methods will be interpreted and serviced by the AOP synchronization service advisor.

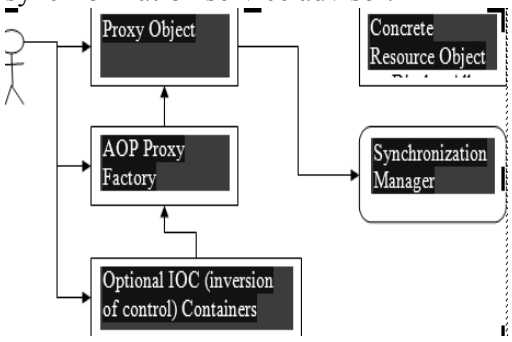

Fig. 2: **Architectural design overview**

AOP ProxyFactory and (ii) a hashable that caches GuardedResouce annotated. Interfaces to help speed up the process of proxy generation.

Structurally, the four separate modules that define the system are:

- Proxy generation
- AOP engine
- Synchronization advisor
- Client resource

AOP Engine

In the AOP engine, AOP is not an inherent part of the Java platform, but rather is implemented as a supporting technology. The flexibility and scalability of the Java language, renders it a fertile ground for extensions beyond even the imaginations and goals that its creators set for it. AOP is implemented in 1 of 2 ways in Java:

- **Interface proxy-ing:** this involve creating proxy classes that implement interfaces that are targets of AOP advising. This is made possible by the addition of the proxy generation framework to the Java default library.

- This framework allows that the generated proxy object will implement, as well as a class loader and an invocation handler. The invocation handler is the component that guarantees the concept of AOP can be implemented. The invocation handler is delegated method calls and is tasked to interpret these as it best chooses.

- **Bytecode Manipulation:** several third party frameworks have been developed that take the implementation a couple of steps farther: they directly manipulate the instructions within the byte-codes of classes that advice is to be applied to. This removes the added complexity of using proxy generation framework, and implementing invocation handlers, designing with interfaces when unnecessary, etc., so the framework users can concentrate on simply their solutions exactly as they want. This paper uses the proxy-ing implementation of AOP. This choice is made because the concepts

employed in this framework are within the bounds of the Java language, and thus fully supported.

Synchronization advisor

Synchronization advisor is actually a chained method service. It intercepts the method invocations, then inspects the context designation annotation on the methods and determines what synchronization strategy to use. The advisor utilizes counters, one per context. These counters signify active threads accessing the resources methods. This way, it can know when to exit a higher priority context and allow lower priority contexts execute.

Client resource

This module consists of interfaces written by the framework user. The interface must be annotated with a GuardedResource annotation. Next, contextual methods are annotated with ResourceAccessor, ResourceMutator, or Prioritized annotations. These annotations correspond to the contexts they are name after. In the case of Prioritized annotation, an argument is accepted; this argument specifies the specific priority level of the method. An enumeration is used to specify a priority from 0 through to 9, 9 being the highest priority. The annotation class exposes methods used to compare priority magnitude. These methods are utilized by the synchronization advisor to know when to enter a higher priority context, or to return to a lower priority one.

.

Experimental Results

On the development station, we used system with a Pentium Dual-core 2.10Ghz Processor, 2GB Ram, 64-bit System Architecture, and Windows 7 Operating System. We also deployed Java Platform version 1.6.0_20 using Java SE runtime build 1.6.0_20-b02. When we ran the program, we discovered that the execution time for our model is lower than the execution time for Java when we used the same number of threads. As seen in the figure, execution time for both Java and our framework increase gradually up to where the number of thread is five (5) and thereafter the execution time of Java increased at a faster rate when compared with that of our model. This is an indication that our framework performed better when context-based synchronization is used than when only Java is used.

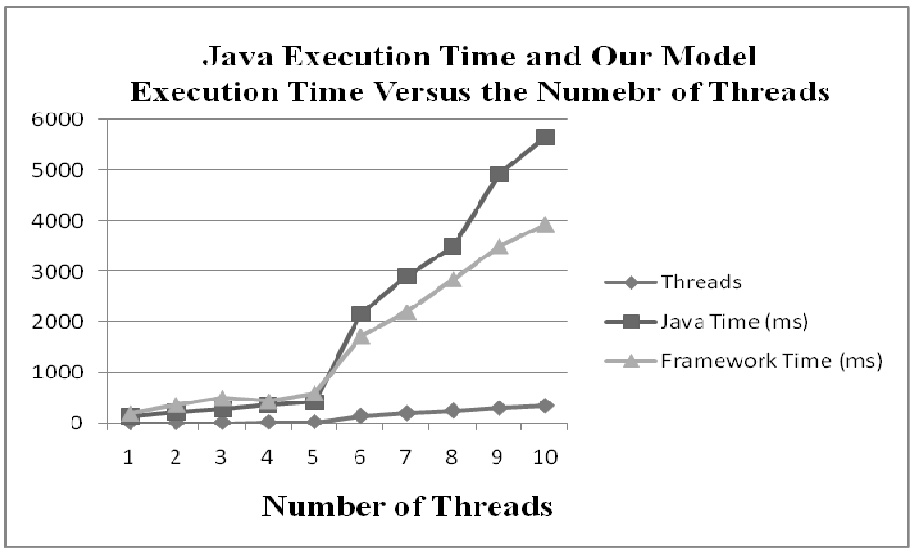

Fig. 3: A graph showing execution time versus the number of threads

Evaluation and Discussion of Result

A simulation was developed to test the validity of the theory that selective/contextual synchronization can gain performance over traditional Java greedy-synchronization style. The simulation was developed in a straight forward implementation of the framework made up of 5 different classes. These classes are:

- **Resource:** this is the interface representing the resource to which selective synchronization is to be applied. It exposes only 2 methods, *accessMethod* and *mutationMethod*, both representing the accessor and mutation contexts respectively.
- **ResourceObjectSync:** this is an implementation of the Resource interface. Its methods are both synchronized to represent the classic scenario for Java's greedy-synchronization style.
- **ResourceObject:** a simple implementation of the Resource interface. Its methods are not synchronized.
- **TestThread:** a thread created to execute the resources methods at random, but over an accessor method being called to that of mutator method. This ratio is chosen because of one of the inherent limitations of the framework with evenly distributed method calls from both contexts; the framework will perform worse than the Java synchronization mechanism due to the overhead of managing the state of the framework classes.
- **Main:** this is the entry into the simulation. It also doubles as a manager of the simulation. It exposes 3 static properties:
  - **CALL_COUNT:** this is a constant integer value specifying how many calls each of the threads created is allowed before exiting.
  - **THREAD_COUNT:** this is also a constant integer value, but specifying how many threads are to be created.
  - **T_COUNT:** an automatically modifiable integer value. It's starts off is equal to the THREAD_COUNT, but with each thread that exits, the value is reduced. When it hits zero (0), the main thread will proceed to record the execution time and exit.

Figure 4 below shows the Prioritized-Resource-Access method for synchronized thread.

```
Public Object
prioritizedResourceAccess(InvocationChai
nLink nextLink, ProxyInvocationContext
context, Priority p)
throws AbortInvocationException,
MethodInvocationException
  {
    GuardStateRecord gsr = null;
    try
    {
      gsr =
this.stateRecords.get(Thread.currentThrea
d());
      if(gsr!=null)
      {

if(gsr.state==GuardState.unsyncAccess)
this.decrementUnsync();
        else
if(gsr.state==GuardState.prioritizedAccess
)
this.decrementPrioritized((Priority)gsr.par
am);

this.stateRecords.remove(Thread.currentT
hread());
  //the reason i remove it is simple, if i
dont, each
@ResourceModifier/@Prioritized method
called hence forth from this method
        //will also decrement the
unsyncAccess counter,which will
obviously lead to erroneous counter
values.
      }

      synchronized(this)
```

```
        {
           try
           {
              this.incrementPrioritized(p);
//Enter Prioritized mode.

while((this.getHighestPriority()!=null &&
this.getHighestPriority().isGreaterThan(p))
||

this.unsynchronizedAccessCount.get()>0 )
this.wait();
              this.notifyAll();

              {

this.stateRecords.put(Thread.currentThrea
d(), new
GuardStateRecord(Thread.currentThread()
,GuardState.prioritizedAccess,p));
                 Object r =
nextLink.link(context);

this.stateRecords.remove(Thread.currentT
hread());

                 return r;
              }
           }
           catch(Exception e){ throw new
RuntimeException(e);}
           finally
           {
              this.decrementPrioritized(p);
//decrement the prioritizedAccessCount,
no matter the outcome...

              if(gsr!=null)
              {

if(gsr.state==GuardState.unsyncAccess)
this.incrementUnsync();
                 else
if(gsr.state==GuardState.prioritizedAccess
)
this.incrementPrioritized((Priority)gsr.para
m);

this.stateRecords.put(Thread.currentThrea
d(), gsr);
                 }
              }
           }
        }
        finally
        {
        }
     }
```

Fig. 4: Prioritized-Resource-Access method for synchronized thread

The class creates either a ResourceObjectSync or a proxy version of the Resource Object and feeds to each thread it creates. Each of these threads are then started and left to run. The threads on the other hand use a random number to generate values from 0-3 inclusive; 3 of these values are mapped to the accessor-method; the other 1 is mapped to the mutator-method. Figure 4 shows the ResourceObjectSync class for implementing the resource.

```
public class ResourceObjectSync
implements Resource
{

   public synchronized void
accessorMethod()
   {
      float x = new
Random().nextFloat()+1;
      for(int cnt=0;cnt<10;cnt++) x/=(new
Random().nextFloat()+1);
   }

   public synchronized void
mutatorMethod()
   {
      float x = new
Random().nextFloat()+1;
```

```
    for(int cnt=0;cnt<10;cnt++) x/=(new
Random().nextFloat()+1);
  }
```

Fig. 5: Resource object Synchronization class

Table 1 below shows the tabulate form of the result obtained after running the simulation a number of times, keeping the operation count constant but varying the number of threads. A graph of milliseconds against thread count is then plotted.

Table1: Simulation Results

Threads	Method Calls	Java Time (ms)	Framework Time (ms)
5	1000	136	200
10	1000	220	370
15	1000	280	500
20	1000	370	440
25	1000	420	600
150	1000	2162	1728
200	1000	2914	2203
250	1000	3500	2848
300	1000	4933	3512
350	1000	5661	3937

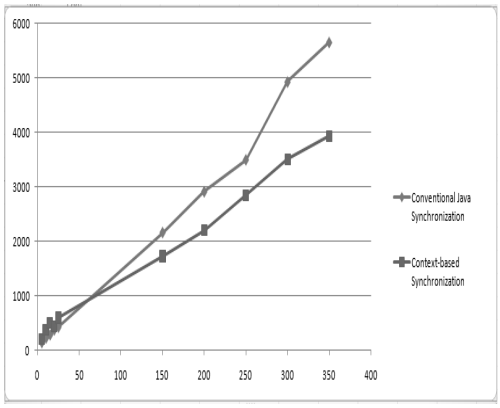

Fig. 6: A graph of thread count against milliseconds

It is obvious from figure 4 that the framework's performance starts lagging behind Java's synchronization mechanism. This is because with fewer threads, there are fewer races/contention for the synchronized resource. Thus waiting in line more is more efficient than selectively synchronizing the resource because of the overhead incurred by the selection process. On the other hand, it can be seen that when the threads increases greatly, the tables turn, and the contextual synchronization out-performs Java's implementation. This makes the contextual synchronization a candidate for server systems where great numbers of threads are spawned to service requests concurrently.

Conclusion

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel. Current programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network. In this paper, we have proposed contextual synchronization model as a solution to the problems inherent in sequential execution of programs or computational process. This is done to ensure a situation where each thread has to wait for the other to finish accessing the resource. The model proposed in this work describes different contexts within which a resource access can be executed.

References

[1] Akhter, S. and Roberts, j. (2006). Multi-Core Programming; Increasing Performance Through Software Multi-Threading, Intel Corporation.

[2] Arora, N. S., Blumofe, R. D., and Plaxton, G. C. (1998). Thread Scheduling for Multiprogrammed Multiprocessors. In Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 119-129.

[3] Böllert, K. (1999). On Weaving Aspects. In Proceedings of Aspect-Oriented Programming Workshop at ECOOP'99, Lisbon, Portugal, June 1999.

[4] Cardelli, L. and Gordon, A. D. (1999). Mobile Ambients, SIGACT: ACm Special Interest Group on Programming Languages, Communication of ACM, New York, NY, USA, pp. 4 - 16.

[5] Flanagan, C. and Qadeer, S. (2003). A Type and Effect System for Atomicity. In PLDI, New York, NY, USA, ACM, pp. 338 – 349.

[6] Flanagan, C. and Freund, S. N. (2004). Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In PODL, New York, NY, USA, ACM, pp. 256 267.

[7] Forgáč, M. and Kollár, J. (2007). Static and Dynamic Approaches to Weaving. In Proceedings of 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics Poprad, Slovakia, January 25-26, 2007.

[8] Herlihy, M., Luchango, V. and Moir, M. (2006). A Flexible Framework for Implementing Software Transaction Memory. In OOPSLA'06, New York, NY,USA, 2006, ACM, pp. 253 – 262.

[9] Hilderink, G. H., Broenink, J. F. and Bakkers, A. W. P. (1998). A New Java Model for Concurrent Programming of Real-Time Systems, Real-time Magazine, pp. 30-34.

[10] Kerns, T. (1998). The Advantages of Multithreading Applications, EE Evolution Engineering, pp. 76-78.

[11] Kickzales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. , and Irwin, J. (1997). Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming, ECOOP'97, Finland, Springer-Verlag LNCS 1241, June 1997.

[12] Merrin, K. (1993). Multithreading Support Grows Among Real-Time Operating Systems, Computer Design, pp. 77-78.

[13] Miller, M. S., Ka-Ping, Y. Shapiro, J. (2003). Capability Myths Demolished. Technical Report, pp. 28.

[14] Miller, S. K. (2001). Aspect-Oriented Programming Takes Aim at Software Complexity, Technology News.

[15] Miller, S. M. (2006). Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, pp. 45-50.

[16] Mordechai, B.-R. (2006). Principles of Concurrent and Distributed Programming (2nd ed., Addison-Wesley, pp. 2-8.

[17] Parnas, D. C. (1972). On the Criteria to be Used in Decomposing Systems into Modules, Communication of the ACM, Vol. 15, No. 5, pp. 1053-1058.

[18] Popovici, A., Gross, T. Alonso, G. (2002). Dynamic Weaving for Aspect-Oriented Programming. In Proceedings of 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, 2002, pp. 141 – 147.

[19] Popovici, A., Gross, T. Alonso, G. (2002). Just in Time Aspects: Efficient Dynamic Weaving for Java, In: 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, 2003, pp. 100-109.

[20] Shavit, N. and Touitou, D. (1995). Software Transactional Memory. In PODC, New York, NY, USA, ACM, pp. 204 – 213.

[21] Stamey, T., Saunders, B., and Blanchard, S. (2005). The Aspect-Oriented Web, Communication of ACM, SIGDOC'05, September 21 – 23, 2005, Coventry, United Kingdom.

[22] Thornley, J., Chandy, K. M. and Ishii, H. (1998). A System for Structured High-Performance Multithreaded Programming in Windows NT. In Proceedings of the 2nd Conference on USENIX Windows NT Symposium, Vol. 2.

[23] Zhang, C. (2009). FlexSync: An Aspect-Oriented Approach to Java Synchronization, IEEE Computer Society, Washington, DC, USA, pp. 375-385