New Trends in Software Methodologies, Tools and Techniques
H. Fujita et al. (Eds.)
IOS Press, 2014

doi:10.3233/978-1-61499-434-3-767

Redefining Game Engine Architecture through Concurrency

Ali MOHEBALI and Thiam Kian CHIEW[1]
Department of Software Engineering, Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur, Malaysia

Abstract. Over the past 30 years, software developers have been conveniently taking advantage of hardware performance increase, giving little consideration to internal architecture changes of the hardware like central processing unit. In the years to come, these hardware architectural changes will affect software architectures and can no longer be ignored. This is especially true for real-time applications, which tend to push the limits of hardware and take the most advantage of available resources. As a result, computer game applications which are inherently real-time and known for pushing computer hardware boundaries will not be immune. By studying the concepts of concurrency, multithreading and multi-core CPU technology, this paper redefines the existing linear architecture of game engines as a generic concurrent and multi-core friendly architecture. Major game engine modules and their inter-dependencies are identified in order to design the new architecture. A sample game was developed to evaluate the performance of the proposed architecture. The comparison of the test results provided in this paper indicates noticeable improvements in the concurrent architecture over the conventional linear approach.

Keywords. Concurrency, architecture, game engine, multi-core, performance

Introduction

Over the past 30 years, hardware architects have improved performance in three areas: clock speed, execution optimization, and cache; which are mainly focused on singular and linear execution. Although Moore's Law predicts exponential growth in hardware performance, clearly exponential growth cannot continue forever before hard physical limits are reached. The growth in single core designs' clock speeds (frequency) is at the expense of faster growth rate in power consumption [1].

Sutter [2] noted that the clock race is already over due to several physical issues including heat, high power consumption, and leakage problems. He concluded that the performance gains in future are going to be accomplished in fundamentally different ways, driven by hyper threading, multi-core, and cache. All these drivers indicate a singular outcome: software development will need to adopt concurrency and parallelism.

The birth of inexpensive parallel computers powered by multi-core in recent years has started a new era of concurrency in the history of software development. There are two main reasons to think about concurrency:

[1] Corresponding author: tkchiew@um.edu.my

- To improve responsiveness especially for real-time applications as responsiveness is a major quality attribute of such applications [3].
- To improve performance and scalability, which according Sutter and James [4], have not been widely investigated through parallelism and concurrency.

The two reasons collectively seem to be valid for game applications where responsiveness and performance are the major concerns and contribute massively to the game play experience of the end users. Nevertheless, games are inherently linear applications consisting of a series of cycles (or frames), making it not easily amendable to parallelization. Each game engine cycle includes a series of operations. Almost all the steps in game engine cycles, from retrieval of the end-user input to updating and processing AI and physics, and finally producing corresponding visual and audio outputs, need to be done in a linear fashion. This linear nature of game application processing makes it difficult to adapt to and utilize the performance advantage that concurrency could offered.

1. Game Engines and Concurrency

Game engines are complex software systems designed for the creation and development of video games and they behave much like an operating system (OS). A decent game engine cycle consists of input-output (IO), artificial intelligence (AI), physics, sound, scene and screen render processing. The tight coupling between the modules as depicted in Figure 1, makes it difficult to design a concurrent game engine architecture.

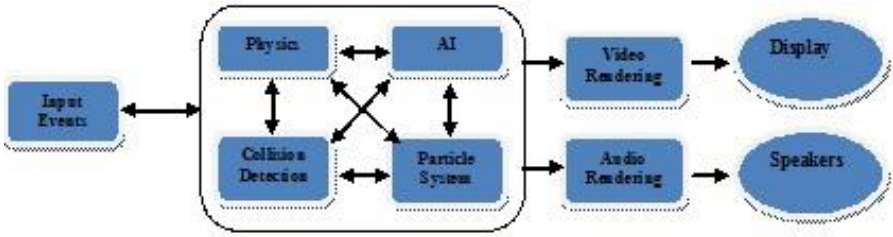

Figure 1. Interaction of modules in a game engine.

Thus far, game applications have been pushing the boundaries of hardware through adaptation or optimization to increase efficiency in usage of available resources [4], without taking full advantage of available CPU power. In order to continually push the limits of hardware, they have no choice but adopting concurrency to utilize all parallel cores [5].

A parallel game server architecture has been proposed by [6] to support concurrency for online multi-player games. The architecture, however, neglects synchronization by assuming network latencies already exist and games are full of approximations. The concurrent video-game design pattern proposed by [7] supports only concurrency within frames, leaving inter-frame concurrency for future improvement. The limitation has been overcome by [8] but the proposed design pattern, namely Sayl, does not consider categorization of tasks in a typical game and their interdependencies. Identification of the interdependencies is essential in

optimizing scheduling of tasks and thus concurrency, too. Researchers have also attempted to achieve concurrency through scripting [9], without an overall game engine. The overall limitation of these researches is that there is a lack of domain-specific (game) concurrent engine with necessary low level details, which the game developers seek. It is thus the intention of this paper to propose a domain specific generic concurrent game engine architecture that meets the needs of game developers.

2. Background

Even though the concept of concurrency has been around for a long time, it is still new to mainstream software developers. Concurrent development concept suffers from the shortage of reusable design patterns and concurrent framework definition [10]. In addition, the concurrent programming model is much more difficult to understand and reason, than it is for sequential and linear control flow.

The ideal scenario of a concurrent application is where all tasks run entirely independent of each other and produce distinct outputs. However, that rarely is the case, and leads to another major difficulty in adoption of concurrency: the degree of coupling between tasks and operations in an application. Tight coupling between tasks and operations makes designing concurrent applications challenging.

Another problem in concurrency is resource sharing and synchronization [5]. Synchronization of resources is traditionally done through locking techniques which introduce performance overheads. Additionally, poor design of resource sharing and locking will lead to data racing, and as a result data corruption, data inconsistency and dead-locks.

The traditional approach to resource sharing is the usage of lock primitives namely mutexes and semaphores. The two primitives prevent specific blocks of code to run concurrently, which if done otherwise, would lead to corruption of shared resources. If a thread attempts to acquire a lock that is held by another one, the thread will be suspended until the lock is released.

Improper design of concurrent architecture, however, can lead to deadlocks, a term used to indicate a set of concurrent threads waiting for resources owned by one another in a circular chain. Deadlock is a major concern in software concurrency as it occurs in run-time under specific circumstances and is very hard to debug. Apart from deadlocks, synchronization locks introduce other runtime issues such as performance slowdowns. These performance issues usually occur due to priority inversion and convoying [11]. Both priority inversion and convoying cause CPU power easily wasted, particularly if the resource sharing is not thought through and designed properly.

While synchronization methods are used to solve low level parallelism issues, they will not be able to resolve concurrency's high level and architectural problems. In order for an application to perform efficiently in a multi-core and parallel environment, it has to be defined and architected as a set of appropriately granulated concurrent tasks. Finding concurrent tasks within the right degree of granularity is a difficult task to which little attention has been paid during the history of parallel programming [10].

Parallelism can be defined through two criteria: data or task [12, 13]. Task and data parallelism seem to fit game engines quite well since they are usually composed of a set of well-defined modules with differing types of functionality. Although this might seem to be true, taking a deeper look into the structure of game engines suggests otherwise. Each module has a well-defined and distinguishable task to perform; but it

will produce data that other modules are dependent on and/or uses data produced by other modules. Thus, although the responsibilities are distinguishable, the data consumed by each module causes the modules to be dependent on one another, making high level task and data decomposition a challenging effort.

Nevertheless, it is possible to decompose each module for parallelism in its own local domain, based on its local tasks and data. Decomposing at higher levels is possible as well, but in much less granularities requiring careful examination of modules' responsibilities and their inter-dependencies.

Joselli et al. [14] claimed that a typical game loop can be divided into three general classes of tasks:

- Data acquisition tasks responsible for retrieving user commands.
- Data processing tasks responsible for updating the game state.
- Data presentation tasks responsible for presenting the results to the user.

Furthermore, based on runtime execution environment, the tasks are classified as either CPU task, GPU task, or both. Thus, they proposed an adaptive loop model which represents a game loop implementation architecture that uses both CPU and GPU to perform and complete game engine tasks.

This model, as depicted in Figure 2, applies the idea of automatic task distribution. The architecture uses a set of heuristic algorithms to study the existing CPU and GPU attributes (e.g. speed) on the system at runtime, in order to perform task distribution between the processing units efficiently.

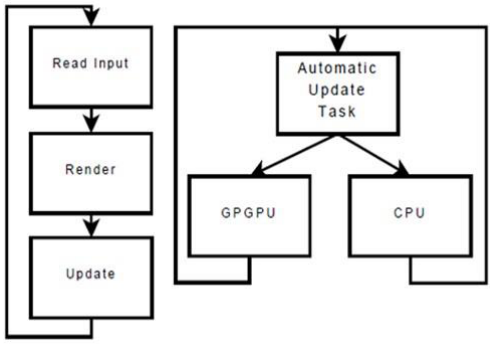

Figure 2. Adaptive Loop Model [7]

The adaptive loop model is based on this categorization and thus the main goal of the technique is the arrangement of the execution of tasks based on their category, in order to simulate parallelism. Even though the adaptive loop model defines a mathematical methodology to efficiently distribute and assign different tasks to different central and graphical processing unit, it does not define an appropriate generic methodology to categorize different tasks in different game engine modules. Furthermore, the technique is exceedingly hardware specific and might not be easily adapted to future hardware.

3. Gaming Modules and Inter-dependencies

Although game engines vary architecturally from one game to another, they all follow identical principal of internal module composition. A typical game engine usually consists of a rendering engine (renderer) for 2D or 3D graphics, physics engine (collision detection and collision response), sound module, scripting module, AI, networking and a scene graph.

In traditional game engines, where all engine tasks are run on a single thread, almost all the steps in the engine cycles from retrieval of the end-user input, to updating and processing AI and physics, and finally producing corresponding visual and audio outputs, are done in an ordered and linear fashion. Although the order of task execution might vary immensely from one engine to another, the overall principal is identical in all games as depicted in Figure 3.

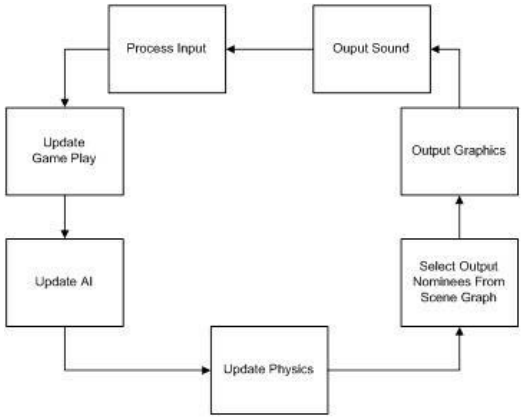

Figure 3. Typical game engine loop

A typical game engine loop starts with the engine retrieves the input from the end user. The game play and AI logic then respond to the input correspondingly, changing states of the game environment. Next, the physics module resolves physics states and collision detection. After that, the scene graph selects and nominates visual objects for presentation to the end end-user. Finally, the renderer updates the graphical screen while the audio plays the corresponding sound effects, collectively reflecting the game state. Although this loop works perfectly in a single threaded engine, it causes architectural difficulties when dealing with multi-threaded and concurrent environments. In the following sections, we will examine the relationships between these fundamental game engine modules with the intention of discovery and definition of the possible data and operational coupling between each of them.

3.1 Input Module

Input module is responsible for entrance of data or information which is fed into the game system and which activate/modify an engine process, eventually causing state or behavioral change in the game logic. In a single engine loop, the input module is the

first process to activate, where the output produced is usually used by the game play logic. The data produced by the module consists of input device changes.

Due to the fact that the input commands are results of end user actions, they can be generated anytime within the engine loop. Although multiple commands can be initiated by the user at different times within a single loop, the input module would evaluate the input changes only once in every loop. As the result, only the latest state of input devices is incorporated, through polling or event listening. In either case, the input states are cached by the input module. The cached data is then accessed by game play module.

Polling system is the preferred system on concurrent engines as the timing of input retrieval can be controlled and synchronized by the input module itself. Since the event listening method can happen anytime within the game loop, data synchronization methods such as mutex locks are needed, introducing complexity to the input system.

3.2 Game Play

Game play module defines the interactive aspect of a game through its rules and logics. Game play retrieves the input commands from the user and updates the states of the game environment accordingly based on the logics defined by the designers. Game play could be considered as the control center of a game engine where all required actions are initiated. Game play affects AI, physics, scene graph and sound modules.

3.3 AI

Game artificial intelligence (AI) refers to algorithms used in video game engines in order to simulate intelligence in the behavior of non-player elements, thus producing illusion of an intelligent interactive environment. In a typical game engine, the AI module operates on a set of AI properties assigned by the game logic to generate intelligent behavior within the game play environment. The AI module might modify or produce visual and auditory data reflecting the intelligent state changes. The visual modification is usually done through scene graph, physics and graphic module. AI module is indirectly dependent on input commands from the end user which are converted into AI parameters through game play module.

3.4 Physics Module

The physics module involves integration of the laws of physics into the game simulation giving the illusion of a more realistic game play environment. A typical physics module is composed of a physics system and a collision detection system. The physics system is responsible for updating the game play environment according to Newtonian physics rules, whereas the collision detection system resolves the interaction and overlapping of different physical objects. In either system, the output is used to update the positional and spatial information of game environment elements. The physics module operates internally on its own local data (physics states from last game engine frame) to produce new or modify the current physical elements states. The output of the physics module is then fed to the scene graph module to reflect the game play environment's physical state.

3.5 Scene Graph

A scene graph module is responsible for operating and maintaining the data structure that systematizes the spatial and logical representation of the game graphical scene. Scene graph module provides the means for rapid scene spatial data query, such as ray tracing and finding visual objects contained within a certain area of the game play environment. These services particularly help the graphics module to build a list of graphical objects that are visible to the end user in order to optimize and minimize rendering operations. The scene graph is also used in AI decision makings in order to generate response and define interactions of an AI agent in relation to the surrounding environment.

3.6 Graphics

The graphics module in a game engine is responsible for generating visual output using a graphics hardware configured on a computer system. A graphics module might operate on 2D or 3D or on both. In any systems, whether 3D or 2D, the visual elements are defined through a visual shape and shading parameters. The shape defines the spatial and volumetric properties of the element while shading defines the coloring and lighting (darkness levels) parameters. Both shape and shading parameters of an element can be modified through the game play to produce animation effects. Graphics module generally does not modify the states of any other module while its data is mostly modified by the game play and AI modules. In order to perform rendering, the graphics module retrieves the spatial and visual data within the game environment through the scene graph module.

3.7 Audio

Audio module is responsible for producing auditory outputs based on game play and AI logic as well as spatial positioning of the game play elements. The audio module merely receives commands of audio data playback and does not modify the states of any other engine module.

3.8 Inter-dependencies between Modules

The examination of individual modules shows that some modules produce and modify most of the data while others only use the manipulated data. The game play and AI modules are the major data manipulators while graphics and audio mostly consume the generated data to produce outputs to the end user. Figure 4 depicts the overall inter-module coupling in a game engine. The direction of an arrow defines dependency on a module.

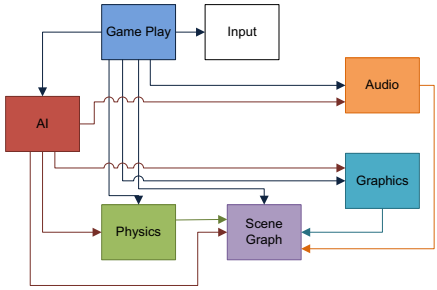

Figure 4. Game engine modules inter-dependency

It can be concluded that scene graph data is the most manipulated/used data while the game play and AI are the heaviest data manipulator/users. As a result, the scene graph data access is likely to be the most congested area of a concurrent game engine and thus requires much optimized synchronization algorithm to avoid performance bottlenecks. On the other hand, input data is only used by game play module and it can be easily synchronize through simple locking algorithms. Game play and AI are the only modules to modify physics, audio and graphics and thus it seems to be a good decision to keep them running under the same thread rather than concurrently, to remove the need of physics, audio and graphics data synchronization.

3.9 Modules' Internal Data/Operation Parallel Decomposition

Apart from module concurrency, a good concurrent game engine must also provide support for internal module task parallelism in order to generate fine grains of concurrent operation threads.

Local task parallelism within the input module can be introduced through separation of input data retrieval based on different input devices. For example, if the game logic requires the state of keyboard and mouse peripherals to be evaluated, the input module can issue two completely separate and independent tasks for the update process: one to retrieve keyboard state, another to update mouse state. Due to the fact that the data for one device is entirely independent of another, the input module scenario makes for an ideal case of local task concurrency.

The game play and AI are very game specific. This is mainly because the game design defines the logic incorporated in the game. For this reason, it is not trivial to define a generic methodology or approach for internal data/operation decomposition of these modules. As a rule of thumb, any operation within the game logic that operates on a separate set of data is a good candidate for local task parallelism. For example, in a real-time strategy game, the AI logic running for two different AI agents (e.g. troops) operating on distant locations on the game terrain can run as parallel tasks.

In general, physics, scene graph, graphics and audio modules are good candidates of data parallelism. Most of the physics engines in recent days split the physical environment into multiple physical islands to minimize the calculations needed to resolve collisions and object interactions. Thus, it is possible for the physics module to operate on the data for each of these islands in a parallel manner.

Scene graphs are inherently designed based on separation of data space. For example, an octree scene graph is a tree data structure used to partition a 3D space by subdividing it into eight octants in a recursive manner. Searching through an octree can therefore be defined as a set of concurrent tasks, each searching through a different partition of the tree and thus operating on a separate set of data.

On the other hand, for graphic modules, any kind of graphical animation can be parallelized through data decomposition. For example, in a typical particle engine, each set of particles cloud is updated separately through the manipulation of a separate set of data. Each of these sets of data has no dependency on the other, and thus they can be manipulated in a parallel manner at the same time.

4. The Architecture

4.1 Data/Operation Separation Model

Based on the examination of a typical game engine and its modules, the following points are crucial in terms of efficiency for a concurrent game engine:

- In order for a game engine to operate concurrently with minimum performance overhead, each module needs to operate locally in its own domain, limited to a minimum shared data access and as little interaction with other modules as possible.
- The engine must support module sub task parallelism, making it possible to have finer grains of operation concurrency and thus higher processing unit task allocation efficiency.

To accomplish the goals, we designed a data/operation separation model. This model is designed based on the concept of data server redundancy, where the engine modules will maintain a local copy of the shared data rather than access common data. This technique removes the cost of using lock methods and thus reduces performance overhead.

To maintain consistency between the local data maintained by different modules, a state manager needs to be implemented. The change to shared data needs to be reported to the state manager, which in return will inform all the systems interested in the data change. It is also possible for multiple modules to modify the shared data at the same time. Rules are defined to determine (synchronize) the correct value after the changes.

This data/operation separation model requires implementation of a centralized scheduler which holds the master clock and is set at a pre-determined frequency. The responsibility of the scheduler is to submit underlying engine modules for execution, through the task manager for every clock tick. The scheduler will wait for all modules to complete execution according to the preset clock tick duration.

The task manager handles scheduling of a system's tasks within its thread pool. The thread pool creates one thread per processor to get the best possible multi-way scaling to processors and prevents over subscription, which in turn avoids unnecessary task switching within the OS. The task manager receives the list of tasks from the scheduler. Only one primary task is defined per system, although each primary task is allowed to generate as many sub-tasks as it needs to operate on its local data. Figure 5 illustrates the overall engine loop explained.

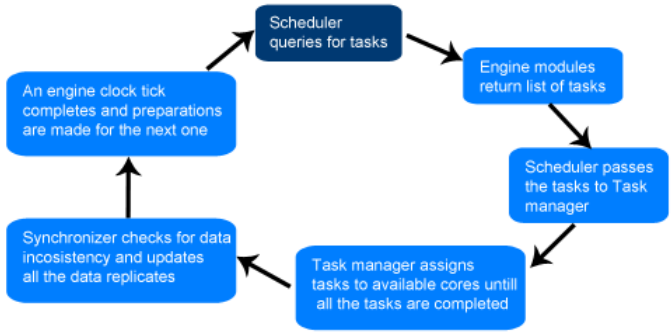

Figure 5. Loop of the engine

4.2 Layered Architecture

The proposed concurrent engine architecture consists of three layers as depicted in Figure 6:

- Framework layer
- Kernel layer
- Engine system layer

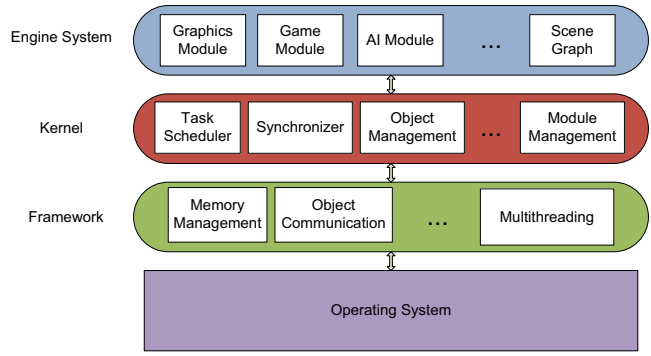

Figure 6. Layers of the concurrent engine architecture

The framework layer is the lowest layer and is responsible for providing OS level functionality such as thread creation, file operation, etc. The framework layer also defines memory management and object messaging used within the engine system. Generally, the framework defines the skeleton of engine architecture to increase portability, extensibility and maintainability of the system.

The second layer, engine kernel, defines the generic engine operations such as object management, module management, data synchronization and task management. Besides providing generic concurrent engine management functionality, the kernel defines interfaces of engine objects and modules that can be implemented to extend engines functionality.

The last layer, engine system, contains the actual engine functionality and game logic that is implemented on top of the kernel. Unlike the typical game engine examined earlier, the engine system layer does not define any specific set of modules and the kernel interfaces are designed so that any types of module, regardless of its functionality and purpose, can be created to extend the engine's functionality while easily integrating into the engine's concurrent environment.

5. Quantitative Validation

A concurrent and a traditional game program with exact features were implemented to assess the performance and accuracy of the proposed engine. As there are significant variations on the game genres, it is essentially expensive to test the implementation of the proposed architecture across different sort of games. Thus, to assure inclusion of most game type scenarios, the sample game plays were designed to include the most common as well as the most computationally expensive components in different game genres as follows:

- 3D physics engine to simulate real world physics;
- 3D renderer for rendering visual feedback;
- 3D scene graph for management of scene objects; and
- Game service managing user input and game logic.

For the traditional game program, all modules are run in a linear fashion under single core implementation. In contrast, under multi-core implementation for the concurrent game program, each engine layer service is queried for available tasks for the next frame. In particular:

- 3D physics generates two or more tasks in each frame for processing collision detection and physics simulation.
- Game service generates only one task querying for input from user and updating game logic.
- Scene graph generates four tasks: one task for generating three scene object lists and three tasks for processing the three scene object list simultaneously.
- Renderer generates one task for rendering the previously generated render list.

In the concurrent game program, the tasks are run in parallel on multiple CPU cores. Once all tasks for a single frame are performed, the engine synchronizes all duplicate data for consistency.

The games include a huge enclosed environment consisting of nine stacks of 125 boxes each. These stacks consist of five rows, five columns and five levels. The user can move the camera around using the mouse and the keyboard, as well as throw heavy metal balls at the box stacks. Figure 7 depicts a screenshot of the physical environment within the programs.

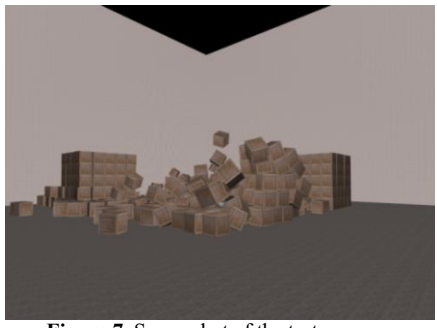

Figure 7. Screenshot of the test programs

A set of tests were performed to analyze the performance of each program on CPUs with different architectures. The performance for each test was measured using frames per second (FPS). A higher FPS indicates less time spent for execution of code in one frame of game program, denoting a more efficient execution. Each test result was sampled for the duration of one minute of game play, with the player moving around and shooting metal balls randomly. As the end-user interacted with the game environment while the test progressed, more objects were added to the scene. As a result, more collision calculation was required and more CPU resources were required to complete the additional calculations, leading to noticeable FPS drops toward the completion of the testing.

FRAPS, a real time video capture and benchmarking application (http://www.fraps.com) was used to measure the FPS of the game at runtime. Instrumentations were run on a set of CPUs. Each CPU was carefully selected to represent a different family of architecture: from single core designs to recent multi-core schemes. The list of CPU models used is shown in Table 1. The AMD model was selected as the candidate for typical single core architecture, whereas the rest of the models exhibit multi-core capabilities and represent different generations of multicore hardware architecture from 2006 to 2010. To reduce random factors in the measurements, the same graphic card and memory configuration were adopted for the tests.

Table 1. CPU models used in benchmarking

Family	Specific Model	Core Speed	Multi-Core	Released
AMD 1.8	Athlon XP 2500+	1.833GHz	No (1)	2003
Intel Core 2	E6600	2.4GHz	Yes (2)	2006
Intel Core 2 DUO	T5250	1.6GHz	Yes (2)	2007
Intel Core 2 DUO	P7550	2.26GHz	Yes (2)	2009
Intel Core I3	i3-2100	3.2GHz	Yes (2)	2010

For the first step of benchmarking, the traditional game program was run on the selected CPUs. Figure 8a depicts the results of the tests showing frame count changes as time progresses. Although AMD CPU was the oldest hardware in the set, its performance was not the slowest. The AMD performed well at first, but the frame rate dropped slowly as more objects in the scene were manipulated. Core 2 DUO 1.66GHz, being the slowest CPU started average and ended in a very low frame rate. The two middle range processing units, Core 2 DUO 2.26GHz and Core 2 2.4GHz, produced equal sampling results as the execution speed per core for both CPUs is quite similar.

Although Core 2 DUO 2.26GHz clock speed is slightly slower than Core 2 2.4GHz, it performed better because of its newer architecture. Core I3 is the latest among the architectures and performed very well. It was only at the end of the test that frame rate on Core I3 started to have minor drop. In conclusion, all CPUs performed according to the clock speed: the faster the core speed, the higher the FPS.

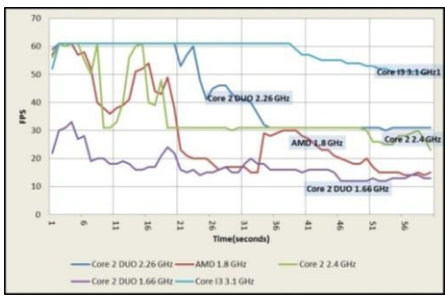

Figure 8a. Results on traditional engine

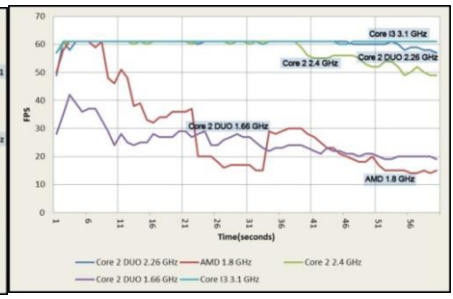

Figure 8b. Results on concurrent engine

For the second step of benchmarking, test results were sampled on the same set of CPUs using the proposed concurrent engine. The results are illustrated in Figure 8b. The AMD CPU was the only one in the set with one core. For this reason, FPS samplings on the AMD CPU for both traditional and concurrent game programs were very close. The sampling results on the slowest CPU, Core 2 DUO 1.66GHz, showed improvement as both cores on the CPU were utilized. In this case, FPS for Core 2 DUO 1.66GHz was increased by almost 10 frames for each sample. As for the higher end CPUs, the improvement was obvious. The latest architecture Core I3 showed almost no frame drop during the one minute sampling period. The other two CPUs, Core 2 DUO 2.26GHz and Core 2 2.4GHz, the frame drop was very minor.

The maximum frame rate for most of CPUs with higher speed showed no significant change in either benchmarking step. This is due to the fact that at the start of the simulation, not much pressure was put on the CPUs by the program. The only exception is the Core 2 DUO 1.66GHz. In this case, the CPU showed improvement of 10 units on the maximum frame rate. This is a good indication that the new architecture improves frame rate on low end multi-core architecture by removing calculation bounding on a single core and utilizing all cores available.

On the other hand, the minimum frame rate for all the CPUs with multicore design has improved, with the change being more significant on older architectures. For Core 2 2.4GHz CPU, the minimum frame rate has almost doubled. Overall, the average frame rate for all multi-core CPUs showed improvement with the new proposed concurrent architecture.

It can be noticed that the only single core CPU, AMD, showed no apparent difference in frame rate in either implementation. Thus, it shows that the proposed concurrent architecture does not affect performance on older single core CPUs while increasing frame rate on newer multi-core CPUs. Table 2 summarizes the improvement by showing the average FPS measured from the benchmarking tests.

Table 2. Average FPS

CPU	Traditional Program	Concurrent Program	Improvement (%)
AMD 1.8	29.55	29.70	5.1
Intel Core 2 2.4GHz	36.02	58.05	61.2

Intel Core 2 DUO 1.6GHz	17.13	25.38	48.2
Intel Core 2 DUO 2.26GHz	44.20	60.33	36.5
Intel Core I3 3.2GHz	58.1	60.88	9.9

6. Discussion and Conclusion

As compared to other software applications, concurrency is still new to game development. Many game developers avoid parallelism due to difficulties related to synchronization and its effect on runtime performance. One of the difficulties in adopting concurrency is that game engines are inherently linear applications. All the steps from consumption of inputs, to process of data until the production of output are inter-related, and to some extent, need to be done in a correct order.

This research takes the initiative to review the existing concurrent software techniques that take advantage of the multithreading and synchronization concepts, such as lock free programming, to propose a generic, flexible, and scalable concurrent game engine architecture. This study has highlighted the need of undertaking the challenge to carefully examine the interrelations of game engine modules at low levels. The aim was to discover sections and operations appropriate for concurrency, while at the same time, maintaining and assuring the required sequence of consumption, process and production of data in a concurrent game engine cycle. Benchmarking and industry experts' assessment were adopted to validate and verify the proposed concurrent game engine architecture. Both of them have shown encouraging results.

As opposed to Adaptive Loop Model, a popular concurrent game architecture which merely addresses engine task concurrency, the architecture proposed in this research defines an appropriate generic methodology to concurrently handle task management as well as data sharing and synchronization. The architecture presented here does not require major change in the existing linear game architectures. The traditional engine step model has been preserved. Engine tasks are still performed within the engine step time frames with exception that the new architecture provides task concurrency on the same frame utilizing all the CPU cores available on an executing hardware.

Another strong point of the proposed architecture is its compatibility with single core processor architectures. The provided design avoids usage of traditional data synchronization techniques (e.g. locks), which have proven to be resource intensive and introduce overheads. This was achieved through application of lock-free synchronization algorithms and data redundancy, which avoid overheads (introduced by older techniques) on single core architecture. For this reason, the number of cores available is transparent to the proposed architecture. The architectural design perceives available CPU resource as a computational unit with variable quantity of cores. In other words, from the proposed architecture point of view, older generation CPUs are just another multi-core processor, with only one core available. At the same time, the architecture does not limit the power of newer generation CPUs. The more cores available, the more tasks can be performed synchronously, leading to improved performance results.

Due to the lockless synchronization virtue of the proposed architecture, real-time creation of new objects by one service which are requested by another synchronous service is unlikely during the task performance phase. As its current state of design, the proposed architecture will only permit such operation during data synchronization

phase. This can be seen as a major drawback for the games where the game play environment data are synchronously generated and destroyed seamlessly in a streaming manner. Thus, the streaming data loaded asynchronously might not be available until the next upcoming frame. As a result, frequent unnecessary downtime of game logic execution might be introduced, leading to underutilization of available CPU power.

In addition, the proposed architecture has not been tested on platforms such as XBOX 360 and PS3, which are known to be the leading game consoles with superior processing power through specialized modern multi-core hardware architecture platforms and might need to be adjusted based on specific concurrency techniques available on each platform. Smartphones and tablets are other major platforms for game application that have gained popularity in recent years. Although improved greatly from the early predecessors, they are still very limited compared to personal computers and game consoles. As a result, handheld devices might lack some of the lockless techniques introduced in this research.

The limitations suggested opportunities for improvement and future work. For example, discovery of an efficient and lock-free technique for generation of shared objects between two different synchronous modules during task performance phase would be desired. It would also be good to implement and test the proposed architecture on platforms like game consoles and handheld devices. Different synchronization algorithms or techniques which are boosted for each specific game platform might be required, too. Meanwhile, as there are many game genres exists in the market and each game type imposes different styles of game play implementation, it is very important for the proposed architecture to be tested across various game types.

References

[1] G. Blake, R. G. Dreslinski, & T. Mudge, A survey of multicore processors, *IEEE Signal Processing Magazine* **26(6)** (2009), 26-37

[2] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal,* **30(3)** (2005). Retrieved 25 February 2013 from http://www.gotw.ca/publications/concurrency-ddj.htm

[3] M. Joselli, E. Clua, A. Montenegro, A. Conci, & P. Pagliosa, A new physics engine with automatic process distribution between CPU-GPU. In *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games* (2008), 149-156.

[4] H. Sutter, & L. James, Software and concurrency revolution. *Queue-Multiprocessors,* **3(7)** (2005), 54-62.

[5] V. Pankratius, C. Schaefer, A. Jannesari, & W.F. Tichy, Software engineering for multicore systems: An experience report. *In Proceedings of the 1st International Workshop on Multicore Software Engineering* (2008), 53-60.

[6] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, C. Griwodz, A demonstration of a lockless, relaxed atomicity state parallel game server (LEARS), In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games (NetGames)* (2011), 1-3.

[7] M. J. Best, A. Fedorova1, R. Dickie, A. Tagliasacchi, A. Couture-Beil, C. Mustard, S. Mottishaw, A. Brown, Z. F. Huang, X. Xu, N. Ghazali, & A. Brownsword, Searching for concurrent design patterns in video games. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, (2009) 912-923.

[8] W. AlBahnassi, S. P. Mudur, D. Goswami, A design pattern for parallel programming of games, In *Proceedings of the 14th International Conference on High Performance Computing and Communication* (2012), 1007-1014.

[9] J. Kehoe, & J. Morris, A concurrency model for game scripting, In *Proceedings of the 12th International Conference on Intelligent Games and Simulation* (2011), 10-15.

[10] T. Mattson, & M. Wrinn, Parallel programming: Can we PLEASE get it right this time? In *Proceedings of the 45th Annual Design Automation Conference* (2008), 7-11.

[11] M. Herlihy & J.E. Moss, Transactional memory: Architecture support for lock-free data structures, *ACM SIGARCH Computer Architecture* **21(2)** (1993), 289-300.
[12] Miller, A. The task graph pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (2010).
[13] Manolescu. D-A. A data flow pattern language. In *Proceedings of the 4th Pattern Languages of Programming* (1997).
[14] M. Joselli, M. Zamith, E. Clua, A. Montenegro, R. Leal-Toledo, A. Conci, P. Pagliosa, L. Valente, & B. Feijó, An adaptative game loop architecture with automatic distribution of tasks between CPU and GPU. *Computers in Entertainment* **7(4)** (2009). DOI:10.1145/1658866.1658869