# A Design of Pipelined Architecture for on-the-Fly Processing of Big Data Streams

Usamah Algemili
Department of Computer Science
The George Washington University
Washington, DC 20052, USA

Simon Berkovich
Department of Computer Science
The George Washington University
Washington, DC 20052, USA

*Abstract*—**Conventional processing infrastructures have been challenged by huge demand of stream-based applications. The industry responded by introducing traditional stream processing engines along-with emerged technologies. The ongoing paradigm embraces parallel computing as the most-suitable proposition. Pipelining and Parallelism have been intensively studied in recent years, yet parallel programming on multiprocessor architectures stands as one of the biggest challenges to the software industry. Parallel computing relies on parallel programs that may encounter internal memory constrains. In addition, parallel computing needs special skillset of programming as well as software conversions. This paper presents reconfigurable pipelined architecture. The design is especially aimed at Big Data clustering, and it adopts Symmetric multiprocessing (SMP) along with crossbar switch and forced interrupt. The main goal of this promising architecture is to efficiently process big data streams on-the-fly, while it can process sequential programs on parallel-pipelined model. The system overpasses internal memory constrains of multicore architectures by applying forced interrupts and crossbar switching. It reduces complexity, data dependency, high-latency, and cost overhead of parallel computing.**

***Keywords—Big Data; Clustering; Computer Architecture; Parallel Processing; Pipeline Design; Variable Lengths; Symmetric Multiprocessing; Crossbar Switch; Forced Interrupt***

## I. Introduction

Conventional computing has been thoroughly challenged by the emerging situation of Big Data. Big Data is the problem of managing huge amount of unstructured data. The complexity of Big Data calls for new form of software clustering and hardware organization. At the beginning of this centenary, studies reported enormous growth of information that exceeded Moore's Law [1]. Big Data introduces unconventional pressure on time and memory performance. Consequently, new computation models are significantly required to cope up with Big Data situation. Researchers introduced "on-the-fly" clusterization of amorphous data. On-the-fly processing deals with a continuous stream of data, and it must maintain certain throughput of information flow. In this pattern, hardware design should not tolerate any postponement of oncoming stream. Multicore pipelined architecture provides a simple yet effective solution to the on-the-fly computation by transferring the operating states from core to core down the pipeline [2]. This pipelining device requires practically the same sequential programs that are currently used based on single processor system. Pipeline computing offers very effective solution for big data streams. It increases the throughput considerably when processing intensive streams of data. Pipelined architectures consist of sequence of processing elements where the output of one processor is the input of the next one. "By pipelining, processing may proceed concurrently with input and output, and consequently overall execution time is minimized. Pipelining plus multiprocessing at each stage of a pipeline should lead to the best-possible performance "[3].

This paper investigates the previous work on multicore processing and parallel computing architectures. It discusses stream processing requirements, followed by general outlook over the current limitations of parallel systems. This paper suggests a hardware model that is especially intended to process Big Data clustering on-the-fly, while this model can process sequential programs using parallel-pipelined multicore design. Finally, it proposes the same model based on Symmetric multiprocessing (SMP) and forced interrupts.

## II. Multicore Processing and Parallel Computing Architectures

### A. *Multi-Core processors*

Most modern processors include huge number of transistors on one chip. The architectures of general purpose multicore processors allow multiple related tasks for execution, this would be conducted in different cores such as IBM Cell processor, Intel and AMD multicore processors. Usually, these cores are heterogeneous in time requirement because of advanced scheduling algorithms that intend to exploit these architectures effectively.

On the other hand, these architectures support shared access of global caches or memory, this support faces some limitations in accessing the same block by other cores which decreases their efficiency. Consequently, memory design has significant influence on high clock rates, and indexing references is important to attain high processing performance. Optimizing the instructions and developing the data queues may increase the performance. However, these solutions have obvious limitations in heavy processing queues like graphics manipulation [4].

For instance, SIMD (Single Instruction Multiple Data) technique was one of the earliest programming methods to stress parallelism in microarchitecture design. More instructions were added by Intel in 2004. The introduction of 90 nm process-based Pentium games processor was followed

by (Streaming SIMD Extensions) SSE3 and SSE4. This was to improve thread synchronization and math capabilities.

Nevertheless, computing field is extensively progressive, so conventional processors such as multicore CPUs (Central Processing Units) are replaced by power-aware multi-core CPUs coupled with GPUs (Graphics Processing units). The idea behind this new trend is to take the advantage of chip die area. This integration can increase the efficiency of SIMD, and it may provide superior environment that supports stream processing and vectorization. This approach uses large memory units and large register sets that are distributed among different levels of system hierarchy [5].

### B. Parallel Programming Architectures

HMPP (Hybrid Multicore Parallel Programming Environment) based on GPUs (Graphics Processing Units) can provide tremendous computing power. With current NVIDIA and AMD hardware group of graphic products, a peak performance can reach hundreds of gigaflops. GPUs designed originally for graphic cards, and it have emerged as the most powerful chip in high-performance workstations. Unlike CPUs with multicore architecture that uses two or four cores on chip, GPUs consist of manycore architecture that can run thousands of threads by hundreds of cores in parallel.

CUDA and OpenCL is a coevolved hardware/software architectures that enable HPC (High-Performance Computing). Developers were encouraged to utilize GPUs' tremendous power in computation and memory bandwidth, this by familiar programming environment -C programming language-. Apparently, the advantages of GPUs over CPUs are undoubtedly interesting. However, an application must retain certain characteristics in order to insure performance benefits: First, well-designed parallelism for massive amount of data. Second, Intensive Kernels that represent very large fraction of execution time should be computed (Amdahl's law) [6]. Third, an application that requires arithmetic intensity and density. These types of applications usually favor the use of the multi-computing units. Forth, Local memory access that is simple and regular, and it should avoid pointer tracking code. Last but not least, local memory accesses that can exploit the pipeline structure of GPU boards [7].

GPU programing mainly uses data-parallel paradigm, which frequently called stream computing. Stream computing relies on a map operation that consists of using the same computation on all elements of a stream such as arrays. Basically, a stream is one or multidimensional array with homogeneous points. Usually, parallel paradigm is based on operations such as Mapping were the map applies kernel function to all elements of stream, yet Kernel function can access elements from many input streams. Also it uses Reduction were an array of elements processes single value.

Typically, there are two types of memory-access operations that can be executed. First, gather operation which assumes kernel is able to read any element of a stream. It is usually of the form of ( x = s1[s2[i]] ) . Second, Scatter operation that assumes kernel is able to write any element of a stream. The form of memory operation is similar to ( s1[s2[i]] = ).

Previous hardware architecture of GPUs was not able to efficiently implement these operations. Luckily modern architectures have overcame this constraint, however, it should note that if ( s2[i] ) is not permutation of ( s1 ), then the result of this operation is non-deterministic. For example, if ( s2[i] = s2[j] = x ), we should consider that if ( i != j ) then ( s1[x] ) will be assigned more than once in undefined order.

### C. Systolic Arrays

Another form of parallelism is systolic arrays, in which the data flows between processors in synchronization. Systolic arrays are specialized form of parallelism where different data may flow in different direction (down or right). Processors compute and store data independently. H. T. Kung and Charles Leiserson were the first to introduce systolic arrays in 1978 showing multiple processors in arrays connected by short buses [8]. Typically, systolic arrays use joint form of parallel computing and pipelined flow of data.

Systolic arrays miss two key features that we aim to achieve in the context of processing big data clusterization. First, Systolic arrays rely on parallel programing, and we target an architecture that would utilize sequential programs by forced interrupts. Second, each processor stores data independently of each other, which adds unfavorable dependency in case we process variable-lengths of data. Finally, the multidirectional nature of systolic arrays adds global synchronization limits due to signal delays. Running time that allows parallel overhead on several processors may exceed the ideal program running time.

### D. Graphic Processing Units (GPUs)

Recently, the use of modern GPU (Graphic Processing Unit) increased considerably. The GPUs have been evolving since the very first computer system placing number of key challenges that are facing programmers to fulfill future application demands and support various platforms.

These challenges such as effective use of GPU's architecture and performance increase have led to outstanding results, yet this computational advancement stimulated software engineers to formulate innovative programming techniques in order to utilize GPUs' capability.

The increasing interest of parallel software requirements heightened the need for deep analyzing and scientific comparison of software methods whether in programming or architecture. The development of GPUs' technology has led to the hope that GPUs will contribute in many applied sciences and will open a new window for continuous growth. Recently, GPUs applications are being adopted by sciences which need processing massive data sets such as physical simulations, image processing, computer vision, data mining and text processing as well as smart phones and portable devices. Consequently, multicore processors and parallel programing has become favorable topic among HPC (High-performance computing) teams [9]. Many researches have been extensively studying general purpose GPUs coupled with multicore processors; they are looking for paralleling the tasks and keeping the sequential execution to its minimum. However, the optimal use of parallelism depends on GPUs architecture.

## III. Problem Statement and Contribution

### A. The Limitations of GPUs

In the context of big data streams, GPUs may not provide enough main memory for large chunks of data. GPUs mitigate this problematic situation by accessing multiple GPU boards to single node. PCIe (Peripheral Component Interconnect Express) can interface with many GPU boards providing an aggregated storage. For instance, eight GPU boards that contains 6 MB of internal memory can allow up-to 48 MB of shared memory. However, this solution does not work efficiently on algorithms that requires random access to large data due to local physics and multi-access constrains. Moving the data among GPUs by high-latency PCIe bus can create huge computational intensity. The moment when algorithms get larger than internal memory of GPU, the performance of PCIe's net system decreases dramatically. In fact, transferring anything over PCIe can lower the speed twentyfold compared to onboard main memory [10].

### B. Parallel Programming Constrains

Real-time processing has led to widespread use of multicore architectures. Experts toke the advantage of multiprocessors by embracing parallel programming in order to exploit parallel cores. This approach provided promising opportunities in HPC (High Performance Computing). However, it created big challenge for software industry. Most existing programming languages are designed to perform sequential execution. Parallel Programming added extra skill-set requirements whereas programmers already deal with many hardware and software complications. As a result, multicore processing unit such as GPU (Graphical Processing Units) has evolved to accommodate these challenges providing well-distributed and properly managed parallel cores, yet software engineers have to place additional effort and time in developing GPU applications that adhere to the promising parallel-core architecture [11].

### C. Contribution

Clearly, this paradigm of parallel computing relies on parallel programming in order to utilize parallel hardware arrangement. It has memory limitation where variable lengths of Big Data streams may require processing data on-the-fly. Nevertheless, the hardware design may change this fact, and it can overcome these constrains. This paper presents a novel multicore pipelined architecture by forced interrupts. It is parallel model of computing that executes Big Data clusters on-the-fly, while it still can utilize sequential programs. Our promising architecture designed to handle variable lengths of data, and it can achieve very low-latency in time. This discourse proposes reconfigurable FPGA hardware architecture, where pipeline length can comply with Big Data clustering work-load. It provides scalable framework that can add more processing units, and it can change HW configuration according to our processing needs. The proposed hardware organization is especially aimed at Big Data clustering. It consists of three main components. First, pipelined multiprocessing elements that operates on multicore tasks in parallel. Second, sophisticated Main Memory management by crossbar switching. Third, forced-interrupt that allows conventional software programs to utilize our proposed platform, and it may eliminate the overhead cost of parallel programming effort.

## IV. Discussion

### A. Multicore Pipelined Architecture for Executing Big Data Streams

The conventional realization of parallel programming introduced three main approaches that have been used in multicore processing. First, Task Parallel that partitions input software into functions and tasks. This approach schedules each function or task onto multicore processor. Second, Data Parallel that partitions input data, then it schedules data segments onto multicore processor system. Third, Pipeline Parallel that uses elements of sequential processing capability. It decomposes a program into states then run each state simultaneously.

While the first approach requires sophisticated software development, the second method is useful for data-independent applications. These applications may require complicated modules for data-scheduling. Pipeline processing is common solution for high-intensive volumes of data; however, it is limited by the maximum length of processing stage and/or the ability for task-decomposition [2].

Multiprocessor Pipelined Architecture proposes hardware design that utilizes the benefits of these three methods in order to process Big Data streams. It uses parallel computation coupled with data-parallel method and pipeline-parallel approach. The main goal of this architecture is to progressively process incoming data flows. We determine the length of the pipeline by the size of data chunk that we receive [11]. Multiprocessor Pipelined Architecture divides the program into equal processing times by forced interrupts as it is described in [Ber90][Ber94][Ber00]. This technology has US PATENT No. 6145071. Given specific time of processing for each processor, this pattern ensures the receiving of incoming data without interruption, while it still using the conventional implementation of sequential software [12][13].

### B. The Multiprocessor Pipelined Architecture by Forced Interrupt

Figure 1 shows multiprocessor pipeline architecture that uses forced interrupts in order to automatically slice each program into fixed durations. Each processing cycle starts with (L) loading, (P) processing and (U) unloading. Since each cycle has fixed duration, this design performs efficiently on large volume of data with relatively small algorithms. The initial design is limited by algorithm size that requires additional round of processing.

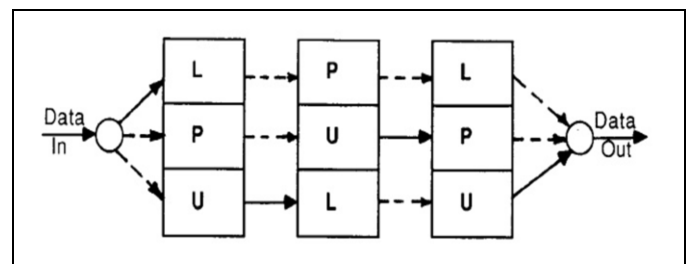


Fig. 1. General diagram of data flow in the pipeline [12]

Figure 2 zooms in the internal structure of each stage. Every microprocessor interacts with three memory blocks M1-M3. MPU sends addresses and control signals, while Sa1-Sa3 direct the data flow from previous stage or to the next cycle [12].

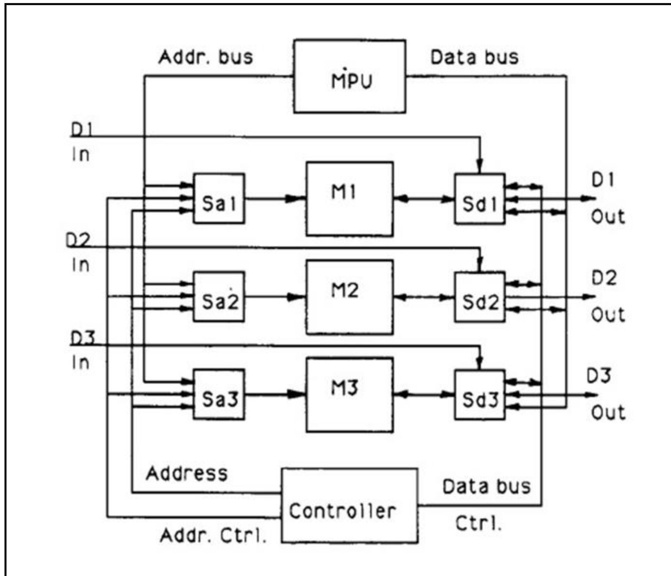


Fig. 2. Internal structure of one stage [12]

The challenges of algorithm size as well as the overhead of memory data transformation are well addressed in recent Multicore Pipeline Organization. The multicore system uses

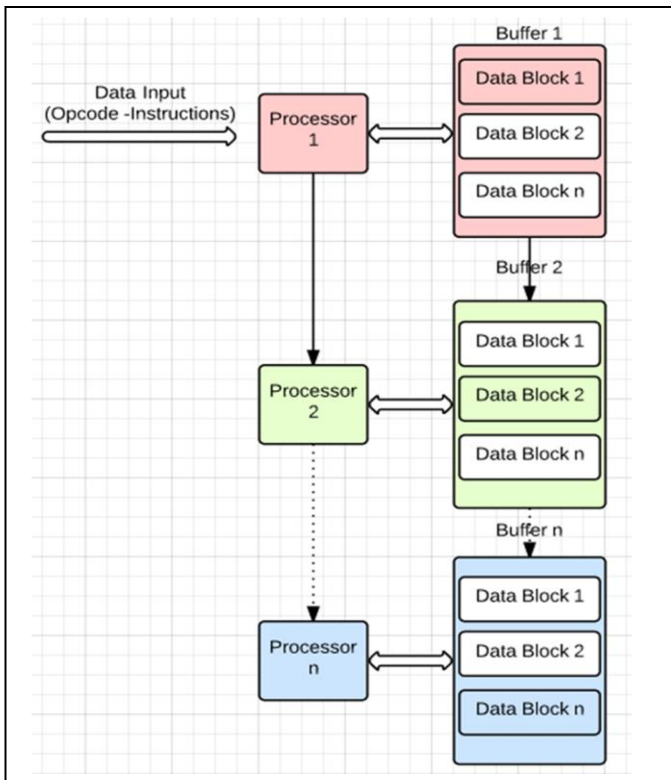


Fig. 3. Ideal situation where the blocks are in equal size

program slicing and forced interrupts. Theoretically, it receives data, then it generates blocks of different sizes based on slice function. Figure 3 shows the ideal situation where the blocks are in equal size, and each processor would execute one block respectively. In this case, each processor executes the block with the same color timing/slicing is not a big issue. In contrast, figure 4 illustrates variable lengths processing. This situation presents data blocks that are not equal in size or processing time. Hence, we need special handling by forced interrupts and program slicing. Each processor may process certain amount of data, then it can be stopped.

The number of processors required for one block execution may vary according to data length and processing time. The multiprocessor pipeline allows an arbitrary algorithm to be performed on-the-fly on a data chunk, given a sufficient number of processors. The major condition for continual mode of stream operations – equal durations of time intervals for computations at each section of the pipeline – is realized by forced interrupts at each processing stage. Time of processing is of no significance to this design due to forced interrupt. Figure 4 shows how different processors can share processing different blocks by forced interrupts.

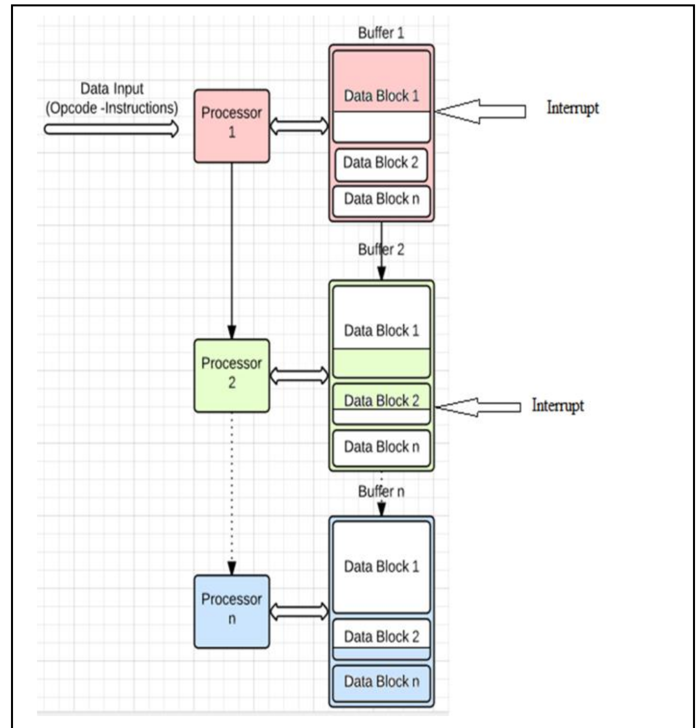


Fig. 4. Data blocks are not equal in size or processing time

In the context of processing Big Data clusters, memory management plays important role, and novel technology of memory and cache management is proposed. This multicore pipelined architecture provides very simple, yet, effective solution by switching the state of processors among data blocks, and it eliminates the overhead of internal-data transformation. As a result, the performance of this pipeline architecture increases significantly.

### *C. Multicore Memory and Cache Architecture Based on Crossbar Switching*

This technology does not relocate memory data down the pipeline; instead, it uses crossbar switch in order to assign memory data blocks to the corresponding processor. Memory data may include program status information that allows the next processor to resume the work starting from last state. This approach has been applied on multi-memory/multi-cache design. Figure 5 illustrates the use of Crossbar Switch internally.

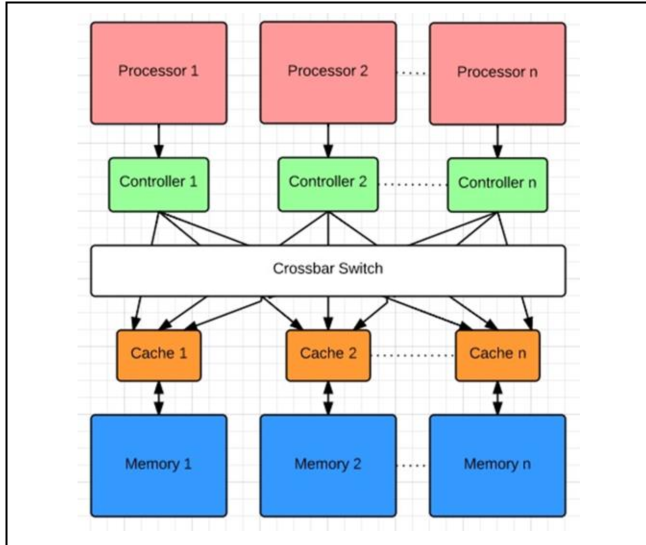


Fig. 5. The use of crossbar switch and shared cache management internally

This design does not assign corresponding memory for each processor; it reduces the cost of data relocation. Cache and memory can be grouped into one set, and Crossbar Switch would assign each processor to one group. The number of groups should be equal to processing elements. This

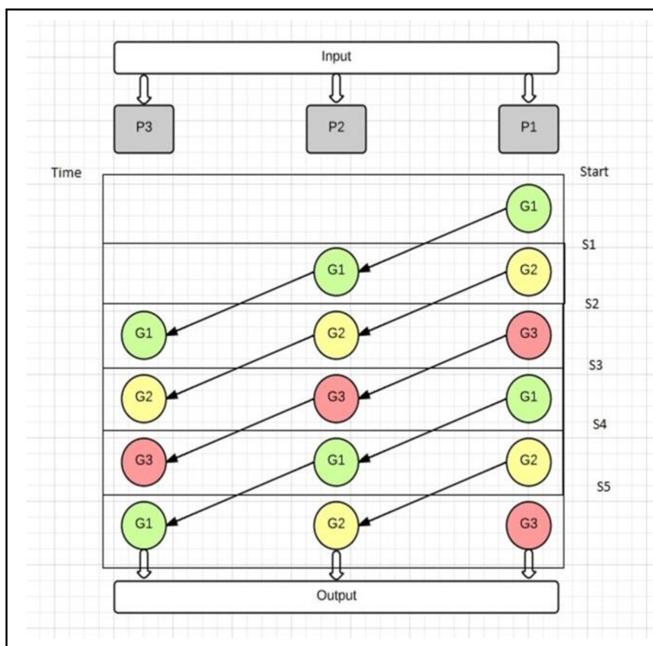


Fig. 6. Processing steps of the pipeline

P: Processor
G: Memory Cache Group
S: Switch

organization eliminates the cost overhead behind memory relocation as well as additional round setup as it described by forced interrupts technique. This organization replaces loading/unloading operations by switching mechanism. Theoretically, we can add as much processing elements as we need to reasonably process any given input of Big Data clusters.

Figure 6 shows the processing steps of Multicore Memory and Cache Architecture Based on Crossbar Switching. Once the data loaded in G1, the corresponding core P1 starts processing G1 data for fixed time, then crossbar switch assign the remaining data to P2. At this stage, P1 starts processing the new incoming assigned by crossbar switch [11].

### *D. New Multicore Architecture Based on Symmetric multiprocessing (SMP)*

This technology follows Symmetric multiprocessing (SMP) architecture whereas many processors can share single main memory. This approach solves the issue of algorithm size, one main memory can receive input data, and then it can assign each memory chunk to processor that would process the instruction set in fixed time. After-which it shifts to the next block of data respectively; that allow the next processor proceed operations where the previous processor stopped.

Computer cluster systems have proved the efficiency of this technique on intensive amounts of data. However, this organization works at processing level. In stream processing, data storage can add high-cost operation within processing path, hence, the system must minimize unnecessary storage operations to archive low latency. Figure 7 shows an architecture that decreases time-intensive operations by processing messages on-the-fly.

In real time situations, we try to avoid dependencies, the program must process messages in given time by timeouts during-which this architecture can proceed with partial data.

The system promotes multi-threading by allowing data partition among processing blocks without having the developer writes low-level code. That would prevent blocking external data thereby minimizing latency.

The objective of this system is to be able to efficiently process external data that can arrive in either variable-lengths, high-volumes or both. In order to achieve better performance, the system should optimize execution path, and it must minimize boundary-crossing overhead. The desired length of the pipeline must be tested with performance in mind to insure sufficient processing path while decreasing the additional cost of multi-processing passage.

Figure 7 illustrates the desired communication between each processor and main memory. This architecture provides more flexibility to add more processing unites into the pipeline sharing the same data source.

Raman and Clarkson [14] carried out interesting project that proof the efficiency of this specific type of architectures.

Their project recognizes parallelism with non-identical processing unites. These unites can work simultaneously with one shared memory.

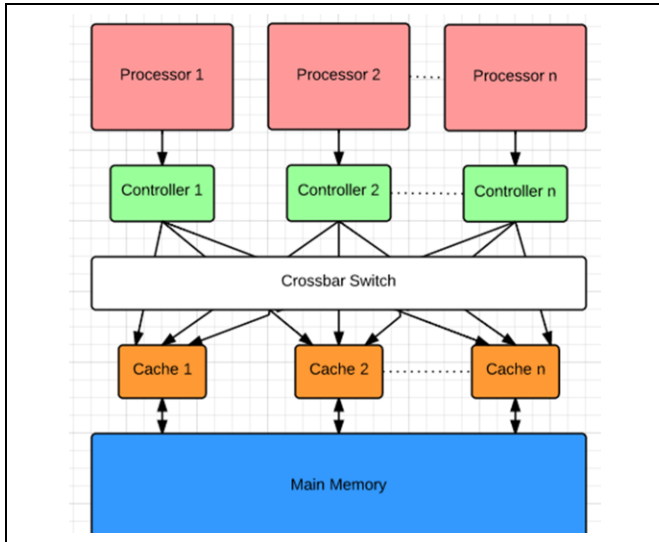


Fig. 7. The desired communication between each processor and main memory

Conventionally, the design of multiprocessor pipeline moves data chunks when processor loading-state changes. In contrast, this new architecture offers two advantages over conventional models. First, the data blocks does not relocate when switch operation occurs. Second, it allows other processors to load data as long it is in ideal state in order to utilize the pipeline.

### *E. Time Table and Analysis*

Table 1 demonstrates one-cycle processing throughout pipeline of three core processors. It explains how variable-lengths of data blocks can be executed in different processors using switching operations and forced interrupts. Switching Operations (S) and forced interrupts happen at the same time. Table 1 illustrates each data block (B) by one color throughout the pipeline. This design assure availability of processor 1 by a given input and hardware specification, and it hinders any complications behind scheduling and load balancing.

TABLE I. ONE-CYCLE PROCESSING THROUGHOUT PIPELINE OF THREE CORE PROCESSORS. L: LOADING. P: PROCESSING. B: DATA BLOCK. S: SWITCHING

| Time | P1 | P2 | P3 |
|---|---|---|---|
| 2 | L-B1 | | |
| 3 | P-B1 | | |
| 4 | P-B1 | | |
| 5 | S-B1 | | |
| 6 | L-B2 | P-B1 | |
| 7 | L-B2 | P-B1 | |
| 8 | P-B2 | P-B1 | |
| 9 | P-B2 | P-B1 | |
| 10 | S-B2 | S-B1 | |
| 11 | L-B3 | P-B2 | P-B1 |
| 12 | L-B3 | P-B2 | P-B1 |
| 13 | P-B3 | P-B2 | P-B1 |
| 14 | P-B3 | P-B2 | P-B1 |
| 15 | S-B3 | S-B2 | S-B1 |

Based on multi-cycle processing, we calculate the time required to perform system operation (T). We assume that a pipeline of (n) cores would execute an operation in (C) cycles. We discretize system operations in one cycle by (D).

A typical processor may execute in time of

$$Tc = (C) \times (D) \tag{1}$$

Given a number of cores (n) to execute input-data on average, the number of cycles performed on each given core presented as

$$X = C \div n \tag{2}$$

Therefore, the time given for typical processor can be also described as

$$Tc = (x) \times (n) \times (D) \tag{3}$$

The length of the required pipeline (L) can be found by

$$L = x + (n - 1) \tag{4}$$

factored by number of cycles that are required for equal duration process (m).

$$Lp = x + (n - 1) \times (m) \tag{5}$$

Let us assume that the internal operations of the pipeline take O = Lp/m that is equal to

$$x + (n - 1) \times (m) \div m \tag{6}$$

$$O = |x \div m| + (n - 1) \tag{7}$$

In multicore system, switching operations can also increase system latency, and each hardware may have different switching capability. We present this overhead for each switch operation by (S). The total switch overhead expressed by

$$Ts = (O - 1) \times (S) \tag{8}$$

Hence, the total number of cores required to process a given data block including switching overhead expressed by

$$L = Lp + Ts \quad (9)$$

and the time required to perform all operations described as

$$T = (L) \times (Ts) \quad (10)$$

This discretization allows us to estimate performance speedup of multicore parallel pipelined architecture compared to typical single core system where Improvement = (Tc)/T.

$$Improvement = (X \times n) \div ([X + (n-1) \times m] + [(|x/m| + n - 2)] \times S) \quad (11)$$

### F. Reconfigurable FPGA Design

#### 1) Overview

The design consists of three processing units all processors share the same AXI stream bus, each processor samples the incoming data when it receives an interrupt signal from the control unit, the output of these processors is send to a multiplexer which selects which output stream AXI bus to be used, the select for the multiplexer is also received from the control unit.

A MicroBlaze processor was chosen to collect the data from the processors; it receives no control signal from the control unit and it treats the data coming from the AXI stream bus as a data from single source.

The control unit is responsible for controlling the multiplexers at the processors input and output also it send interrupt signals for the processors to start sampling data.

#### 2) Processors

Each processor contains the following ports:

*a) AXI stream input*

*b) AXI stream output.*

*c) Interrupt input.*

*d) Busy output.*

*e) Offload input.*

The design consists of a finite state machine with 2 states and a 256 32-bit Ram. It stays on the first state waiting for the interrupt from the control unit, upon receiving an interrupt, the finite state machine goes to second state where it samples and stores the incoming data in the Ram. After storing 256 word, it goes to final state it enables the processor AXI stream output and reads from the Ram until it reaches final address then it goes back to first state.

#### 3) AXIS DEMUX

The demultiplexer routes the system input stream AXI bus to any of the three processors according to the select signal it receives from the control unit. It has four main ports:

*a) One AXIS stream input.*

*b) Three AXIS stream outputs.*

This module can be removed from the design of a single AXIS data source. It is used to feed the three processors (AXIS counter for example).

#### 4) AXIS DEMUX

The AXIS multiplexer is responsible for selecting which processor output to be fed into the MicroBlaze, it receives its select input from the control unit, all parameters are adjustable However, the control sends the select signal for clock cycles which is the time required by the processor to offload the data in its Ram.

#### 5) Control unit

The control unit is responsible for synchronizing the complete design, and it shares the same data valid signal with the processors from system AXIS input bus in order to track the number of data words in the design. It consists of a finite state machine with 3 states. In the first state, it gives control signals to enable the AXIS Input and output of processor 1, and it counts the incoming data words until it reaches 256 (all parameters are configurable). After that, it go to the second state where it selects the AXI input and output of the second processor and after 256 data word it go to final state where it selects the third processor and then it return back to first state to repeat the complete process.

*a) Interrupt:*

The control unit can send the interrupt signal for the processor according to processing requirements, in the initial design the control unit sends the interrupt signal when the data reaches 256 word. This can be adjusted to interrupt according to any data word length or to data rate if a timer is used.

### G. Design performance

This is a system designed using AXIS stream bus. Therefore all data transfer are constrained by the AXIS protocol timing performance as a result a word can be send each clock cycle in case of holding the valid signal high and sending the data each clock cycle.

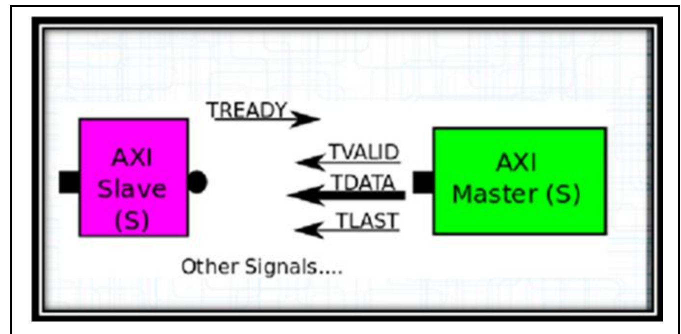


Fig. 8. AXIS protocol stream bus

In this design we assumed that TVALID is set high and a new data word is available each clock therefore the design processes a data word each clock cycle.

The complete design is synthesized at 100 Mhz which is the available clock source in the FPGA Zedboard [15]. For future work, we recommend to upgrade the project to include AXI timer and embedded design to access the timer; this would

enable reading the number of clock cycles for any future processing operations which my effect the processing time.

## V. Conclusion and Future Work

Parallel programming on multiprocessing systems is challenging software domain. We proposed program slicing by novel method of dynamic resource management that allows organizing on-the-fly processing of arbitrary complexity without parallel programming. The new architecture is very suitable for handling Big Data systems. The experimental results and performance comparison with existing multicore architectures demonstrate the effectiveness, flexibility, and diversity of the new architecture, in particular, for large data parallel processing.

The considered pipelining processing is of especial significance for applications of the presented technique of Golay Code clustering [16] as it involves very diverse and rather sophisticated computations for realization of multiple data cross-sections with sophisticated “Meta Knowledge” templates. Performance analysis introduces promising opportunity in real-time processing from pre-processing steps of clustering algorithms until final visualization.

## References


[1] Brown, John Seely, and Paul Duguid. The social life of information. Harvard Business Press, 2002.

[2] Liao, Duoduo, and Simon Y. Berkovich. "A new multi-core pipelined architecture for executing sequential programs for parallel geospatial computing." In Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application, p. 23. ACM, 2010.

[3] Dewdney, A.K. The (New) Turing Omnibus. Henry Holt and Company. New York. 1993.

[4] Raman, S. & Clarkson, T. (1990) Parallel image processing system – a modular architecture using dedicated image processing modules and a graphics processor. IEEE, Conference on Computer and Communication Systems, September 1990, Hong Kong, pp. 319–323.

[5] Che. S, Boyer. M and others. (2009), Rodinia: A Benchmark Suite for Heterogeneous Computing, Department of Computer Science, University of Virginia.

[6] Amdahl, Gene M. "Validity of the single processor approach to achieving large scale computing capabilities." Proceedings of the April 18-20, 1967, spring joint computer conference. ACM, 1967.

[7] Bodin, F., & Bihan, S. (2009). Heterogeneous multicore parallel programming for graphics processing units. Scientific Programming, 17(4), 325-336. doi:10.3233/SPR-2009-0292 From : http://ehis.ebscohost.com.proxygw.wrlc.org/eds/pdfviewer/pdfviewer?sid=0dc925aa-da7b-4e0e-852d-2df3c6def810%40sessionmgr112&vid=3&hid=2

[8] Quinton, Patrice, and Yves Robert. Systolic algorithms & architectures. Prentice Hall, 1991.

[9] Brown, J.D.; , "High Performance Processor Development for Consumer Electronics Game Processor Perspective," VLSI Circuits, 2007 IEEE Symposium on , vol., no., pp.112-115, 14-16 June 2007 doi: 10.1109/VLSIC.2007.4342680 URL: http://ieeexplore.ieee.org.proxygw.wrlc.org/stamp/stamp.jsp?tp=&arnumber=4342680&isnumber=4342661

[10] Novakovic, Nebojsa A. "CPUs Will Fight Back as GPU Computing Hits the Limits." - The Inquirer. The Inquirer, 1 Aug. 2012. Web. 02 Dec. 2014. <http://www.theinquirer.net/inquirer/feature/2195344/cpus-will-fight-back-as-gpu-computing-hits-the-limits/page/2>.

[11] Liao, Duoduo. Real-time solid voxelization using multi-core pipelining. Diss. The George Washington University, 2009.

[12] [Ber00] S. Berkovich, Z. Kitov, A. Meltzer: On-the-fly processing of continuous data streams with a pipeline of microprocessors. In Proceedings of the International conference on Databases, Parallel Architectures, and Their Applications (PARBASE-90), IEEE Computer Society, Maiami Beach, Florida, March 1990, pp. 85-97.

[13] [Ber00] S. Berkovich, E. Berkovich, and M. Loew, 2000. "Multi-Layer Multi-Processor Information Conveyor with Periodic Transferring of Processor's States for On-The-Fly Transformation of Continuous Information Flows and Operating Method Therefor", US PATENT No. 6145071, owned by George Washington University. Date issued - November 7, 2000.

[14] Raman, S. & Clarkson, T. (1990) Parallel image processing system – a modular architecture using dedicated image processing modules and a graphics processor. IEEE, Conference on Computer and Communication Systems, September 1990, Hong Kong, pp. 319–323.

[15] Zynq, Xilinx. "7000,“." Zynq-7000 all programmable soc overview, advance product specification-ds190(v1. 2) available on: http://www. xilinx. com/support/documentation-/data sheets/-ds190-Zynq-7000-Overview. pdf,” August (2012).

[16] F. Alsaby and S. Berkovich. Realization of clustering with Golay code transformations. Global Science and Technology Forum, J. on Computing (JoC) Vol 4 No 1, 2014.

[17] Alhudhaif, Adi, Tong Yan, and Simon Berkovich. "On the organization of cluster voting with massive distributed streams." Computing for Geospatial Research and Application (COM. Geo), 2014 Fifth International Conference on. IEEE, 2014.

[18] Spafford, Kyle L., Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures." In Proceedings of the 9th conference on Computing Frontiers, pp. 103-112. ACM, 2012.

[19] Vivado, H. L. S. "Vivado high level synthesis." (2012).