INTELLIGENT CO-OPERATIVE PROCESSOR-IN-MEMORY ARCHITECTURES

Reza Sotudeh*, Zaki Ahmad, Faycal Bensaali
School of Electronic, Communication and Electrical Engineering
University of Hertfordshire

ABSTRACT

Advances in VLSI technology are enabling the processor-memory integration to bridge the processor-memory performance gap. It is also a key driver in the innovation of a new concept called Processor-In-Memory (PIM). The work described in this paper capitalises on the extensive work carried out on PIMs in general and develops a road map for an intelligent revision of a PIM architecture referred to as Co-operative Intelligent Memory (CIM). The journey made to reach the goal of achieving a CIM is taken via the route of developing a Cooperative Pseudo Intelligent Memory (CPIM), as proof of concept and mid point in the ratification of the intelligence needed for a full CIM implementation. Both architectures use a hierarchical two level CPU structure referred to as major and minor CPUs. By partitioning computation through dividing workload between major and minor CPUs in an intelligent manner and without any pre-processor compilation or kernel task scheduling, the PIM system can be made more efficient and co-operative for class of tasks, which are heavily reliant on memory-to-memory iterative processes. The proposed architectures exploit the key feature in the iterative process by using vectors that characterize the iteration. The process of identifying intelligently these vectors is described in this paper. In addition, the performance of the proposed architectures has been evaluated

Keywords

Processor-In-Memory, Co-operative Pseudo Intelligent Memory, Co-operative Intelligent Memory, Major CPU, Minor CPU.

1. INTRODUCTION

Current high performance computer systems use interface to the main memory through a hierarchy of caches and interconnect systems. This approach invests many resources to bridge the performance gap between CPU and main memory. The processor-memory performance gap has been and continues to be the primary obstacle to improving computer system performance [1, 2]. This gap was also the key motivation behind the concept called processor-in-memory (PIM) or intelligent memory. This concept capitalizes on merging the processing unit with its memory unit on the same chip [3].

This approach has led to much innovative architectures, which include Intelligent RAM [3], Computational RAM [4], Raw [5], Smart Memories [6], all of which strive to remove expedite processor-memory performance [7, 8].

All the above are based on concepts that treats the processor and memory unit as a complete architecture, as a main processing unit in the system. In contrast, architectures such as Active Pages [9], FlexRAM [10], and DIVA [11], are designed to be used as a co-processor in memory that executes code when signaled by the host (main) processor. Using an explicit job partitioning technique for the co-processor and the main processor, the memory-intensive or data-intensive functions are assigned to the co-processor and computationally intensive functions to the main processor. These architectures can be classified based on the role of the PIM chips: main processor or co-processor.

Data intensive applications require demandingly high number of memory accesses, which have operational characteristics that include a significant amount of memory-to-memory type of instructions. This is in contrast to the usual statistically distributed register-to-register, memory-to-register and memory-to-memory instructions that are found in most programs.

It is important to be able to expedite process with data-intensive computation loops, inherent in many applications, especially image processing [12, 13]. These applications usually input and output significant amounts of data which are processed with relatively simple operations. The algorithms deployed in these applications involve data intensive, iterative and most often, highly parallel tasks.

The work described in this thesis capitalises on the extensive work carried out on PIMs in general and develops a road map for an intelligent revision of a PIM architecture referred to as Co-operative Intelligent Memory (CIM). The journey made to reach the goal of achieving a CIM is taken via the route of developing a Cooperative Pseudo Intelligent Memory (CPIM), as proof of concept and mid point in the ratification of the intelligence needed for a full CIM implementation. Both architectures use a hierarchical two level CPU structure referred to as CPU_major and CPU_minor. The CPU_major has a conventional architecture while CPU_minor is a task specific processor dealing with highly iterative memory-to-memory processing. CPIM uses a pre-compilation task optimization process to determine the division of work between CPU_major and CPU_minor. However, by partitioning computation through dividing workload between major and minor CPUs in an intelligent manner and without any pre-processor compilation or kernel task scheduling, the PIM systems can be made more efficient and co-operative for class of tasks, which are heavily reliant on memory-to-memory iterative processes. These tasks include those of image processing algorithms deployed in real time image visualization applications. The proposed architectures exploit the key feature in the iterative process by using three vectors, Vector Starting Address (VSA), Vector Job Size (VJS) and Vector Job Nature (VJN) that characterize the iteration. These vectors denote the static program execution profile (a small window mapped on the overall dynamic program profile) on a range of memory locations where the corresponding data are stored. This relates to a single multi-iteration loop in a locality chart that cache memory exploits.

*Corresponding author: r.sotudeh@herts.ac.uk

Hence, these vectors form the basis for an architecture that complements the main CPU's activities and co-operates in expediting the overall task. An additional vector is introduced (Vector Instruction Block (VIB)) to facilitate the migration from CPIM to CIM and enable intelligent acquisition of run time parameters. The process of dividing the task intelligently between major and minor CPUs and the identification of the described vectors is investigated in this thesis. The work presented is backed by theoretical analysis and performance measures against conventional taxonomies, hierarchical memory, and the aspects of the proposed architecture are analyzed in hardware through use of FPGA as proof of fundamental concepts.

The structure of this paper can be split into four parts. In the first and second parts the proposed CPIM and CIM with their descriptions and performance analysis are presented respectively. A comparison matrix is given in the third part. Part four concludes the paper.

2. CO-OPRETAIVE PSEUDO INTELLIGENT MEMORY (CPIM)

A co-operative processing policy is adopted in this section. The core of this policy is the exploitation of the heterogeneity of the system by partitioning an application into two parts: one that benefits mostly for the high capability of the CPU_major and other that benefits from the processors in the PIM chips that provide high bandwidth and low latency access to memory. Task partitioning in CPIM is based on scanning the assembler output file by a program called "*Task Optimizer*". It extracts the vectors that characterize the iteration. These vectors are portraying the number of iterations in the loop, starting address of the operand block and the job nature of loop. Loops are then replaced with the corresponding vectors and as a result, a new assembler file emerges. Once the re-assimilated code is linked and executed by the CPU_major, it continues on its non-iterative job. When the by-passed part in the re-assimilated code is encountered, the vector components will be loaded into the CPIM registers. Once all the registers have been initialized, CPIM controller, an additional hardware unit, manages the transfer of related data from main to corresponding CPIM memory which initializes the task represented by the by-pass. Thereafter, CPIM will take care of the respective iterative loop by continuous reference to its own registers.Any reference to intelligence in the context of our proposed architecture is limited to the definition that an Intelligent System (IS) is a system which learns how to act towards a certain situation in order to reach its objectives by using experiences and knowledge gained previously.

From the above statement, we can conclude that an IS has two fundamental characteristics learning and serving. Typically, an IS achieves its objective through knowledge and experience which is something that has happened to the IS during some moment of its existence. It includes the situation that occurred, the action done, and the results, acquired through a learning process.

Vector loading into the CPIM registers demonstrates a learning stage, where the CPIM trained for a particular situation. Due to re-initialization of CPIM registers during the course of executing the same program, serving stage, with same data set shows that the system is unable to use experience and knowledge gained previously. However, in the presence of new data set, when a taught situation is detected (iterative loop), it partially (re-initialization of CPIM registers) behaves like an intelligent system, with a new set of results. This new set of results or output partially exhibits the use of experience and knowledge gained previously for a particular situation.

2.1 Task partitioning

The strategy for the distribution of workload is based on parsing the assembler output file for program flow-control instructions. The "*Task optimizer*" scans the assembler output file and figures out the iterative part of the job, extracting the vectors that characterize the iteration. The process of vector identification is shown in Figure 1 with each *job* comprising of few or many instructions that can be algorithmically described working on data entities. The vector components are then extracted from the *Intensity* and *address range axes*. Figure1 depicts a typical static program execution profile on a range of memory locations where the corresponding data are stored. This relates to multi-iteration loops that cache memory exploit. This program behavior supports the need of tasks-partitioning between iterative and non-iterative jobs. Hence alleviating the major CPU of mundane repetitive tasks. This can be done by extracting the vectors, starting address, job size and Job nature that describes the iteration.

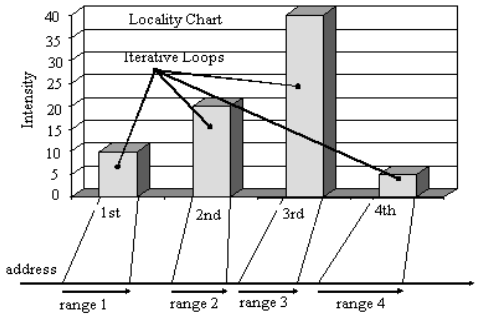

Figure 1. Locality chart

Figure 2 shows the sequence of extracting vector components and identification of task by parsing the assembler output file and locating flow control instructions defined in a database. Analysis of the flow control instructions that portray loops of iteration will result in identification of *Intensity* and *address range* vector components (stage 1). Further analysis of the iteration will yield the algorithm that is used to define the *job* (stage 2).

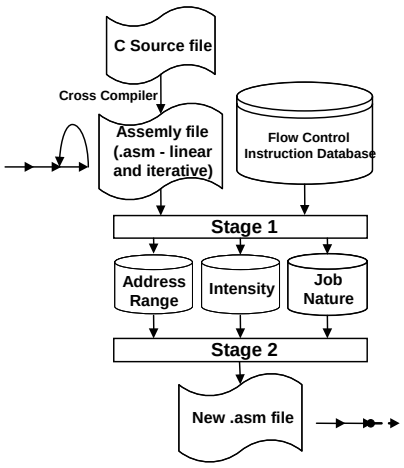

Figure 2. Code optimization process

The following actions are carried out by the task optimizer as shown in Figure 3:

- Extracting information that includes address range for the operands, the loop Intensity and finally the nature of job

including its granularity. This allows the formation of a bypass.

- Re-assimilating the .asm file to generate a new .asm file which includes the vector component information by replacing or bypassing its corresponding iterative loop.

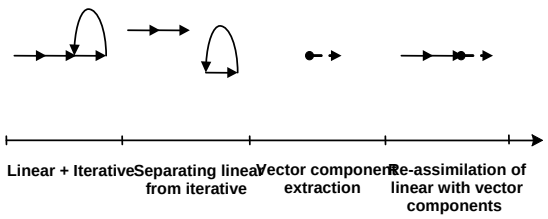

Figure 3. Re-assimilation process

2.2 Proposed CPIM Architecture

The simplified model of the proposed CPIM is shown in Figure 4.

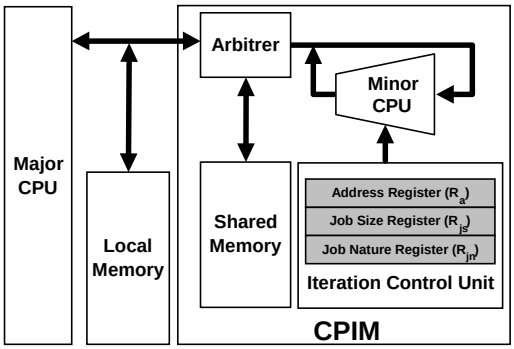

Figure 4. Proposed CPIM architecture

The CPU_major has a conventional architecture and poses no real design constraint on the CPIM architecture. It is backed up by a deep cache hierarchy and suffers high latency to access memory. The augmented system, which includes CPIM, introduces a new block of memory, which is shared through arbitration between the CPU_major and CPU_minor, an iteration control unit and CPU_minor. The arbitration circuitry optimizes for individual CPU accesses by offering cycle stealing to CPU_major and burst transfer to CPU_minor. CPU_minor is a task specific processor that consists of a small computational unit performing iterative processing. The CPU_major provides high Instruction Level Parallelism (ILP), and the processors in the CPIM chips provide high bandwidth, low latency access to the memory.

Our proposed architecture has the following characteristics:

- The memory capacity is large enough to hold large data frames synonymous with high-resolution image frames.
- The overhead associated with the time used to fetch and execute the instruction in a specific program loop is eliminated.
- No need for special instructions as required in the case of coprocessor.
- CPU_major can continue with other operations while the CPIM is completing its allocated task.

The CPIM's basic building blocks are described below.

2.2.1 Arbiter

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. In a shared memory multiprocessor system, more than one processor may request access to the memory simultaneously or at close intervals through the system bus. An arbitration mechanism is used to select those requests which can be honored, rejecting others requests. Rejected requests are generally re-submitted on subsequent processor cycles. The number of re-submission before a request is finally accepted is an important consideration and is dependent upon the arbitration protocol. A high level of system performance is achieved by choice of an efficient protocol. Different protocols give different system performance depending on the system demand, processor-memory interconnection network, and number of processors in the system, accepted traffic intensity. At this stage of research, a simple communication protocol is considered, once a CPIM vector registers or its memory (Shared memory) fills with fresh entries (active mode), the corresponding CPU_minor has the priority to communicate with its own local memory, otherwise is free for the others (sleep mode). In active mode, all components of the CPIM are active. In sleep mode, only memory part is active and external devices can access the memory for read-write operation.

2.2.2 Shared memory

A SRAM type memory, holding data related to the iterative job, having enough capacity to hold large frames synonymous with high-resolution image frames. In the context of the proposed architecture, shared memory holds true to the code optimization/task partitioning phase only and thereafter it becomes exclusive to minor CPUs.

2.2.3 Iteration Control Unit (ICU)

The ICU provides an instruction set for CPU_minor. It consists of three registers, namely address register (Ra), job size register (Rjs), and job nature register (Rjn). The initialization of the vectors needs the following aspects:

- A m-bit register (Ra) is required to hold the start address of the operand block. Once initialized, a counter will then increment the pointer, pointing to the next operand required by the task. This increment step size could reflect data granularity.
- A n-bit register (Rjs) is initialized with the total number of operands needed by the job (the number of iterations involved in the by-passed iterative loop), which is the number of data grains in the memory block on which the job is carried out.
- A k-bit register (Rjn) is initialized, representing the nature of job. Assumed 4-bits for the job definition (op-codes). The remaining bits could be used for byte, word and long word setup. In addition bits could be used for advanced operations. For example, bits can be used as status flags indicating CPIM module is busy with the task.

2.2.4 Minor CPU (CPU_minor)

It is a task specific processor that communicates with the shared memory. It consists of a dedicated computational unit, performing simple and iterative processing.

2.3 Execution

Once the re-assimilated code is linked and executed by the CPU_major, the by-pass is enforced as a series of memory store instructions, initializing CPIM registers as a single pass and hence reducing the time that would normally have taken to execute the replaced iteration. Hence execution of linear code without any diversion that flow control instructions pose will improve the CPU_major performance.

The CPU_minor commences the designated task once all 3 registers have been initialized and can interrupt (or otherwise indicate) the CPU_major upon completion of its task.

Assuming that the CPU_major has equally sized instruction grains (instruction length and execution cycle) yielding Δt instruction execution cycle time. Figure 5 shows the program execution profile with and without bypass where N is the number of iterations in the bypassed iterative part, and $\Delta t'$ is instruction execution time in CPU_minor.

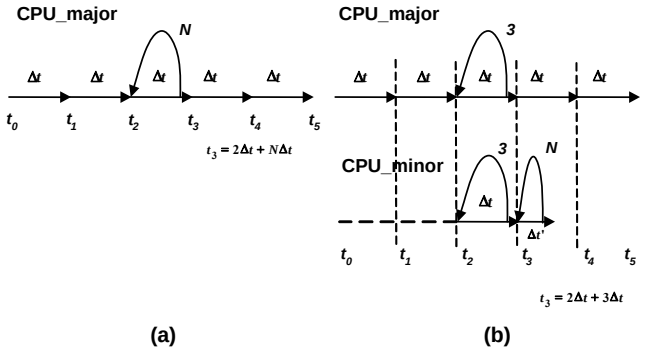

Figure 5. Program execution profile: (a) without bypass (b) with bypass

The rendezvous time at t_3 for CPU_major +CPU_minor scenario is therefore far earlier than CPU_major on its own. The difference is t_3' and given as:

$$t_3' = (2\Delta t + N\Delta t) - (2\Delta t + 3\Delta t)$$

$$t_3' = (N - 3)\Delta t$$

and as N → ∞ then $t_3' \rightarrow N\Delta t$. This reduces the arrival time at t_4 for CPU_major significantly and hence contributing to the overall speedup.

2.3.1 Memory access bandwidth

The CPU_minor offers speed enhancement over CPU_major by highly optimized job processing algorithm as well as its ability to be clocked at a higher speed. Additionally as the job performed by the CPU_minor is repetitive, therefore it has equal quanta or computational grain that lends itself to synchronous transactions in contrast to the CPU_major's asynchronous memory access needs. The latter then requires an arbitration mechanism, switching from cycle-stealing to burst-transfer (synchronous to $\Delta t'$). This accommodates seamless transition from asynchronous to synchronous memory access at time t_3 in Figure 6.

2.4 Performance Analysis

With all possible kinds of parallelism, a framework is needed to describe particular instances of parallel architectures. The Flynn's stream approach was found to be suitable for describing the performance of the CPIM at this evolutionary stage [14].

CPIM based system requires a setup time for code optimization and CPIM registers initialization. The setup time includes:

1. Additional time $T_{S/W}$: required for code optimization and CPIM registers initialization.
2. Data transfer time T_{DT} : transfer time of data related to the iterative loop from main to shared memory.

The additional time for the code optimization is required only once during the re-assimilation process (see Figure 3). The CPIM registers initialization time becomes negligible as the number of iterations increases. The data transfer time between main and shared memory makes processing time longer during first execution cycle only, which is a single learning phase to acquire the knowledge about the Static-Locale-Dynamic-Content type of data structure. Therefore, the impact of setup time has no considerable effect on the performance of CPIM based system in the serving stage during the course of executing the same program.

2.4.1 CPIM vs. SISD

The following notations are used in the performance analysis:

N = number of iterations in the loop.

T_S = Total time to finish a task on SISD machine.

f_S = Machine cycle frequency for SISD machine.

f_{CPIM} = Machine cycle frequency for CPIM.

M_{cycle} = Machine cycle.

$T_{S/W}$ = Time required for vectors extraction and the initialization of cpu_minor registers.

T_{DT} = Data transfer time.

T_{CPIM} = Total time to finish a task on CPIM.

S = Speedup of CPIM over an equivalent function SISD machine.

The following calculations are based on the assumption that one M_{cycle} is equal to one clock cycle.

In an SISD machine, a processor fetches instructions and data from a memory, operates on the data, and writes the results back into memory.

The number of machine cycles involved in SISD to complete one instruction cycle are as follows:

- 1 M_{cycle} for the instruction fetch;
- 1 M_{cycle} for the instruction decode;
- 2 M_{cycle} for the operand fetch (operand 1, operand 2) ; and
- 1 M_{cycle} for the instruction executes and writes back into memory.

Given the time period: $\tau_S = \frac{1}{f_S}$,

Then:

$$T_s = 5M_{cycle}N\tau_S \quad (1)$$

The number of machine cycles involved in the CPIM to complete one cycle are as follows:

- $2\ M_{cycle}$ for the operand fetch (operand 1, operand 2) ; and
- $1\ M_{cycle}$ for the instruction execute and write back into memory.

Given the time period: $\tau_{CPIM} = \frac{1}{f_{CPIM}}$,

Then:

$$T_{CPIM} = (T_{S/W} + T_{DT}) + 3M_{cycle}N\tau_{CPIM} \quad (2)$$

Therefore, the speedup of a CPIM over a SISD machine can be calculated as follows:

$$S = \frac{T_s}{T_{CPIM}} \quad (3)$$

Substituting (1) and (2) in (3),

$$S = \frac{5M_{cycle}N\tau_S}{(T_{S/W} + T_{DT}) + 3M_{cycle}N\tau_{CPIM}} \quad (4)$$

If $N \to \infty$, $T_{S/W}$ considered to be negligible. Then $(T_{S/W} + T_{DT}) + 3M_{cycle}N\tau_{CPIM}$ Approaches to $T_{DT} + 3M_{cycle}N\tau_{CPIM}$.

Therefore,

$$S = \frac{5M_{cycle}N\tau_S}{T_{DT} + 3M_{cycle}N\tau_{CPIM}} \quad (5)$$

Assuming that both systems are clocked at the same rate then $\tau_S = \tau_{CPIM}$.

Thus,

$$S = \frac{5N}{T_{DT} + 3N} \quad (6)$$

The data transfer time T_{DT} has no considerable effect on CPIM performance because it is only involved during the first execution cycle.

Therefore, Equation 6 can be written as,

$$S = \frac{5N}{3N} = 1.66 \quad (7)$$

Equation 7 indicates that the CPIM architecture can provide better result over a conventional SISD machine for highly iterative memory-to-memory tasks. However, the dramatic increase in the system performance on known program execution profile will be observed in each serving stage during the course of executing the same program.

2.4.2 CPIM vs. SIMD

SIMD machines typically are used to process array. There are multiple Processing Elements (PEs) supervised by the same control unit but operate on different data sets from distinct or multiple data stream. Multiple CPIMs, when work in a group and has the same tasks to do, function like an array processor with behavioral difference. This is due to the sequential activation of different PEs (CPIMs) in the learning phase of the proposed architecture instead of parallel activation of processing elements in SIMD machine. Multiple CPIM modules initialize with the same instruction, which is the vector job nature, and operate on the related data sets on distinct or multiple data stream.

The speedup of SIMD machine over a functionally equivalent SISD machine is given as [15]:

$$S = t_n \sum_{i=1}^{n} N_i \Big/ t_a \sum_{i=1}^{n} \frac{N_i}{m} \quad (8)$$

Where,

m = Number of PEs in an array processor.

t_a = Time required by a PE to compute the execution of a broadcast instruction from the control unit.

N_i = Length of vector operand (number of operand) in the i^{th} instruction.

S = Speedup of an array processor with 'm' PEs.

t_n = Time required by a SISD (assumed to be independent of instruction types).

The following additional notations are used in the performance analysis:

N = Number of iterations in the loop.

f_{array} = Clocking frequency of the PEs.

T_{CPIM} = Time required by a CPIM to compute the execution.

$Total_{CPIM}$ = Total time required to finish the task in multi-CPIM based system.

T_{Setup} = Setup time (initialization of individual CPIM modules involved in the specific task.

T = Total time required to finish the job in SIMD system.

S_1 = Speedup ratio between CPIM and SIMD based processing.

Assuming there are 'm' CPIMs corresponding to the 'm' PEs in SIMD machine.

Therefore,

$$T_{CPIM} = (T_{S/W} + T_{Setup}) + 3M_{cycle}N_i\tau_{CPIM} \quad (9)$$

Thus,

$$Total_{CPIM} = (T_{S/W} + T_{Setup}) + N_i[\frac{3M_{cycle}\tau_{CPIM}}{m}] \quad (10)$$

Given the time period: $\tau_{array} = \frac{1}{f_{array}}$

Then,

$$t_a = 3M_{cycle}\tau_{array} \quad (11)$$

Equation 11 shows that two machine cycles are required to fetch two operands and one machine cycle for instruction execute and write back.

$$T = 3M_{cycle}\tau_{array}[\frac{N_i}{m}] \quad (12)$$

The speedup ratio between the two systems is:

$$S_1 = T / Total_{CPIM} \quad (13)$$

Substituting (10) and (12) in (13),

$$S_1 = \frac{3M_{cycle}\tau_{array}[\frac{N_i}{m}]}{(T_{S/W} + T_{Setup}) + 3M_{cycle}N_i\tau_{CPIM}} \quad (14)$$

Assuming the activation time 'Δt' between the two CPIM modules is the same. Then, $T_{Setup} = N\Delta t$.

If $N \to \infty$, $T_{S/W}$ considered to be negligible. Therefore $(T_{S/W} + T_{Setup}) + N_i[\frac{3M_{cycle}\tau_{CPIM}}{m}]$ approaches to $(N\Delta t + N_i[\frac{3M_{cycle}\tau_{CPIM}}{m}])$.

Thus, speedup becomes

$$S_1 = \frac{3M_{cycle}\tau_{array}[\frac{N_i}{m}]}{N\Delta t + N_i[\frac{3M_{cycle}\tau_{CPIM}}{m}]} \quad (15)$$

Equation 15, speedup ratio $S_1 < 1$, shows the processing time of CPIM based system during its learning phase extended due to sequential activation of logically related CPIM modules. However, after learning phase on current execution profile the setup time $N\Delta t$ becomes negligible.
Then,

$$S_1 = \frac{3M_{cycle}\tau_{array}[\frac{N_i}{m}]}{N_i[\frac{3M_{cycle}\tau_{CPIM}}{m}]} = 1 \quad (16)$$

Therefore, the system enjoys the benefit of CPIM on known execution profile in serving stages with the computational result obtained in the learning phase described.

2.4.3 CPIM vs. MIMD

An intrinsic MIMD computer implies interaction among the '*n*' processors because all memory streams are derived from the same data space shared by all processors. If the 'n' data streams were derived from disjointed subspaces of the shared memories, then we have the multiple SISD operation, which is nothing but a set of 'n' independent SISD computer [15].
CPIM inherently acts as a SISD machine. However, when it works in a group (multiple SISD) and each module has the same or different job to execute, it behaves like a MIMD machine.
In MIMD, several processors are fetching their own instructions and operating on the data those instructions specify. In CPIM, once the re-assimilated code is linked and executed by the CPU_major, all the functionally active CPIM registers will be initialized sequentially.

Let assume that:
$T_s =$ Total time required to execute different tasks on '*m*' SISD computer. Or

$T_s =$ Total time taken by the MIMD system.

$T_{CPIM} =$ Total time taken by the CPIM based system.

Then,

$$T_s = m(5M_{cycle}N\tau_s) \quad (17)$$

$$T_{CPIM} = (T_{S/W} + T_{DT}) + m(3M_{cycle}N\tau_{CPIM}) \quad (18)$$

The speedup ratio between them,

$$S = \frac{m(5M_{cycle}N\tau_s)}{(T_{S/W} + T_{DT}) + m(3M_{cycle}N\tau_{CPIM})} \quad (19)$$

If $N \to \infty$, $T_{S/W}$ considered to be negligible.

Then,

$$S = \frac{m(5M_{cycle}N\tau_s)}{T_{DT} + m(3M_{cycle}N\tau_{CPIM})} \quad (20)$$

Assumed both systems are clocked at the same rate then: $\tau_s = \tau_{CPIM}$.

Therefore, speedup becomes

$$S = 5N / T_{DT} + 3N \quad (21)$$

Since data transfer time T_{DT} stretches processing time during the first execution cycle only. Therefore, Equation 21 can be written as:

$$S = \frac{5N}{3N} = 1.66 \quad (22)$$

Equation 22 indicates that the CPIM architecture exhibits better performance over its equivalent MIMD counter part. The computational results obtained during the first execution cycle, learning phase, increase the system performance drastically.

2.5 Implementation Scenarios

Two simple test benches have been implemented using the model in Figure 6 CPU model:

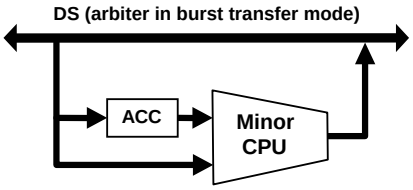

Figure 6. CPU model

The speedup is then measured against a SISD without significant performance acceleration methods (by modern standards) to ensure a speedup assessment is obtained against base-line architecture.

2.5.1 Scenario 1

Cumulative successive addition (Non-destructive): an array of 'y' numbers is added and the result is stored in the defined memory location. 'x' represents the result location in the shared memory.

RTL:

$$ACC \leftarrow M[0] + M[1]$$

$$ACC \leftarrow ACC + M[2]$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots..$$

$$ACC \leftarrow ACC + M[y-2]$$

$$M[x] \leftarrow ACC + M[y-1]$$

Figure 7 shows the cumulative successive addition pipeline. Both edges of the clock are used in each cycle. In the figure, the processor machine cycles are defined as follows:

- OF1= Operand 1 Fetch;
- OF2=Operand 2 Fetch;
- IE=Instruction Execution; and
- WBA=Write Back Accumulator.

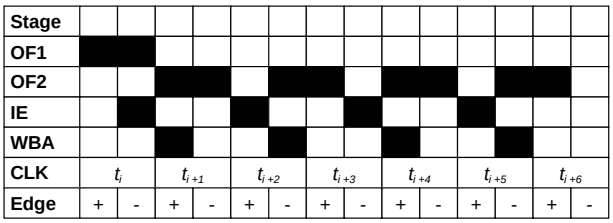

Figure 7. Cumulative successive addition pipeline

2.5.2 Scenario 2

Non-cumulative successive addition (Non-destructive): Data in consecutive memory locations are added and the result is stored in a defined memory location starting at $M[x]$. The last location for storing data is $M[x+(y-1)]$ covering a range of addresses signified by *jobsize*, where y=*jobsize*.

RTL: $M[x] \leftarrow M[0] + M[1]$

$$M[x+1] \leftarrow M[2] + M[3]$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$M[x+(y-1)] \leftarrow M[y-2] + M[y-1]$$

Figure 8 shows the non-cumulative successive addition pipeline. Both edges of the clock are used in each cycle. In the figure, the processor machine cycles are defined as follows:

- OF1= Operand 1 Fetch;
- OF2=Operand 2 Fetch;
- IE=Instruction Execution;
- WBA=Write Back Accumulator; and
- WBM=Write Back Memory.

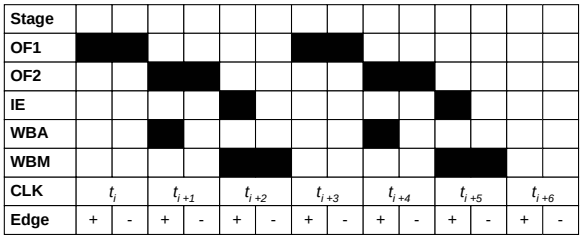

Figure 8. Non-cumulative successive addition pipeline

For the proof of concept, the architecture in Figure 4 has been used to implement the above two scenarios. The only difference between the mappings of the described scenarios is the functionality of the task specific processor (CPU_minor). The CPIM in Figure 4 has been implemented on a SPARTAN II, XC2S300E-6PQ208C FPGA using the Nexar 2004 EDS environment. The performance curves of the two scenarios under the proposed architecture, illustrated in Figure 9, show 18 fold increases in speed for iterative task compared to a non-pipelined SISD machine.

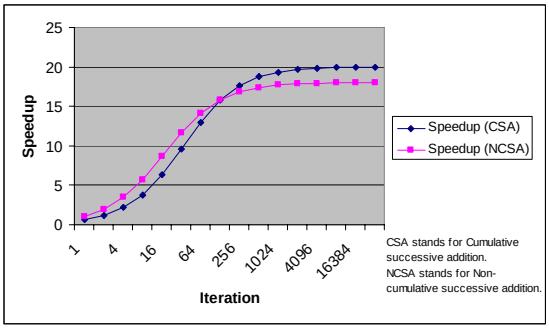

Figure 9. Performance curve

3. CO-OPRETAIVE INTELLIGENT MEMORY (CIM)

A *Learn* and *Serve* policy is introduced in this paper. During learning stage, CPU_major works on both iterative and non-iterative parts of the task to gather intelligence on current program execution profile. During serving stage, CIM serves the system through the knowledge gained during the learning stage.

3.1 Learning Stage

The strategy for the distribution of workload or task partitioning is conducted by the hardware based on the knowledge that the system gained from the learning stage rather than pre-processed by the software. An additional hardware unit responsible for the detection of iterative loops called *Observer* will monitor the activities operating on the address and data buses. When a task shows iterative behavior, after a qualifying threshold, which is the optimum number of iterations, the *observer* records the vectors that characterize the iteration. The following vectors, VSA, VJN and VJS can be used in CIM to improve the performance of conventional architectures for highly iterative memory-to-memory tasks. They allow specific logic block, which is the CPIM (the main building block of CIM), to work in parallel with the CPU_major then releases the burden of simple iterative tasks from CPU_major. Migration from CPIM to CIM needs an additional

vector, VIB. It corresponds to the start and end address of the by-passed loop. To enhance the capability of the CPIM, another additional vector, Vector Distribution Block (VDB) is also required. It corresponds to the start and end address of the memory locations, where the computed values are stored with additional bits for step-size.

A key feature in an iterative loop is that it mostly exhibits equal steps, often with the same offset as a step size. When this offset changes, then the *observer* recognizes that the loop is completed or terminated. The *observer* records the vectors in specific registers allocated for this specific loop. Once learning stage is completed, *Information Transfer Control* (ITC), a sub part of the *observer* initializes the corresponding CPIM block, and removes the detected loop from the instruction memory by storing the NOP type instruction code into the related instruction memory block. All this instruction does is increment the Program Counter (PC) so that the next instruction is ready to execute.

Figure 10 shows the sequence of extracting vector components and identification of task. Analysis of the read activity on the data will result in identification of the starting address of the operand block, total number of operand with step-size vector components (stage1). Further analysis of read-write activity will yield the algorithm that is used to define the intensity (number of iterations in the loop) and nature of job (stage2). It must be pointed out that granularity of algorithm detected is limited to simple tasks in this initial stage of the proposed architecture, but the system can be scaled up to cope with coarser granularity.

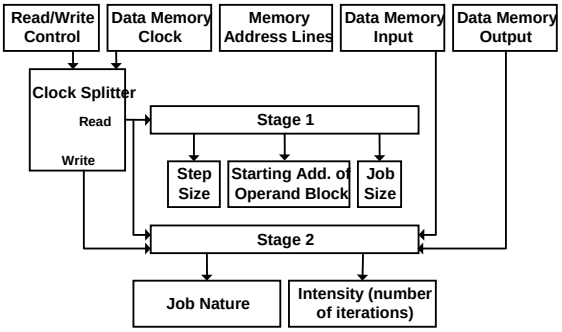

Figure 10. Activity on data memory

Figure 11 shows the identification of starting and ending instruction addresses of the corresponding iterative loop. An iterative part of the program mostly starts from CMP (compare source to destination) type instruction and ends on BRA (branch always) type instruction. Whenever the *observer* detects these instructions, records their locations (physical address) into two separate registers. Once the *observer* recognizes after a qualifying threshold the loop is terminated. It records the currently available CMP and BRA addresses, which are the starting and ending instruction addresses of the corresponding iterative loop.

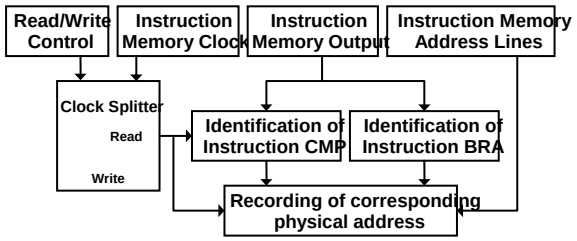

Figure 11. Activity on instruction memory

3.2 Serving Stage

Whenever the CPU_major encounters a process that it has already been executed in the course of executing the same program and it approaches the part in the program at which the first iterative task was located, it encounters NOP codes for the execution instead of the iterative loop described. Thus, the proposed CIM architecture does not need a set of instructions during its serving stage and complements the CPU_major activities by having acquired the knowledge about the current program execution profile during the learning stage. This intelligent characteristic makes proposed architecture more efficient and co-operative in expediting the overall task.

3.3 Extraction and Initialization of Vectors

Figure 12 illustrates the extraction of VSA and VJS.

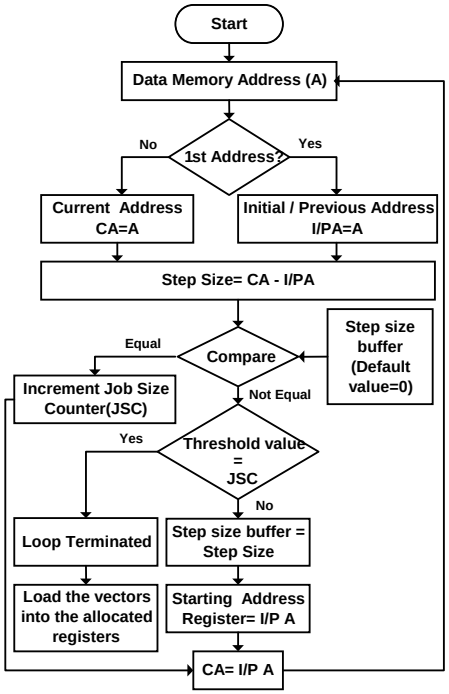

Figure 12. Extraction of VSA and VJS

The CPU_major generates the address of a data word to be read. This address is subtracted from the previous address to get the step size. In case of iteration, there should be a fixed addressing increment. A job size counter, counting the number of operand advances one-step on successive operand address. Once the step size changes after a defined threshold, which is the minimum limit for bypassing the iterative loop from the main stream, start address, address from where the iteration commences, and total number of operands are stored in the allocated registers. However, if step size changes before the minimum limit which tells that the activity is not suitable to bypass then the *observer* initializes the counter and loads the current address. The content of the counter and the initial address acts as a VJS and VSA respectively.The extraction of VJN and VDB is shown in Figure 13. During the read activity of the learning cycle, the two consecutive operands are inputted into different "*functional units*" and the results of these *functional units* are compared with the actual result materializes during write activity period of the learning cycle. The outputs of "comparators" appear at the input of the encoder that generates a specific code related to the action. This code acts as a

Vector Job Nature (VJN). The data memory addresses, during write period of the learning cycle can be used as a Vector Destination Block (VDB).

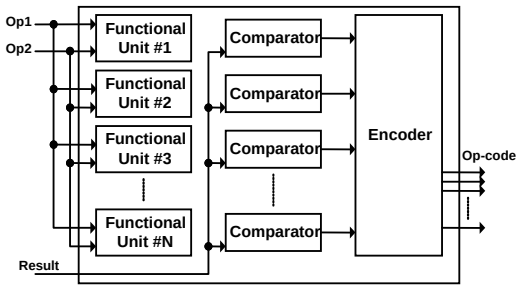

Figure 13. Extraction of VJN and VDB

Figure 14 illustrates the extraction of VIB. An iterative part of the program mostly starts from CMP (compare source to destination) type instruction and ends on BRA (branch always) type instruction.

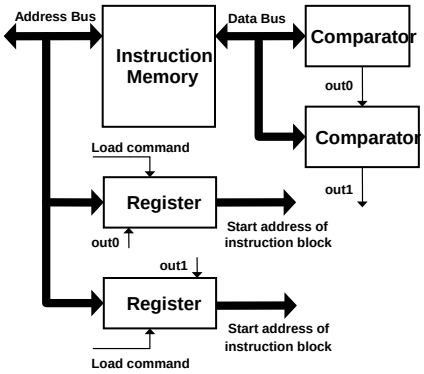

Figure 14. Extraction of VIB

Whenever the *observer* detects these instructions, records their locations (physical address) into two separate registers. Once the *observer* recognizes after a qualifying threshold the loop is terminated. It records the currently available CMP and BRA addresses, which are the starting and ending instruction addresses of the corresponding iterative loop. These addresses can be used as a VIB.

The initialization of vectors needs the following aspects:

- A register R_a is initialized with the start address (VSA) of the operand block.
- A register R_{js} is initialized with the total number of operands (VJS) needed by the job, which is the number of data grains in the memory block on which the job is carried out.
- A register R_{jn} is initialized with an m-bit value. n bits are used to represent the nature of job (VJN), where 2^n is the number of functional units used (see Figure 6). The remaining bits of the register could be used for byte, word and long word setup. In addition bits can be used as status flags indicating the CPIM is busy with its task.
- Two registers, R_{sai} and R_{eai}, are required to hold the start and end addresses of the instruction block (VIB).
- Two registers, R_{sdi} and R_{edi}, are required to hold the start and end addresses of the destination block (VDB) with step-size.

3.4 Information Transfer

Once all vectors are extracted and memory block is free meaning no read/write activities, the ITC transfers all the recorded information with the related set of data involved in the iteration into the CPIM placed in the memory system. The observer during transfer of information carries out the following actions (see Figure 15).

- Senses data memory is free which means no read/write operation or a flag that indicates the completion of learning phase.
- Interrupts the CPU_major for the control of the buses and wait for the acknowledgement.
- Issues the load commands for the specific logic block (CPIM) registers to copy the extracting vector components.
- Re-issues the address range of iteration and generate read-write cycles for reading the data from data memory (DM) and writing into the shared memory (SM) of the CPIM.
- Re-issues the address range of instruction block of the loop and generates write cycles that replaced the original code with NOP instruction code.
- Generates Data Transfer Complete (DTC) signal that indicates the ITC no longer requires the control of buses.

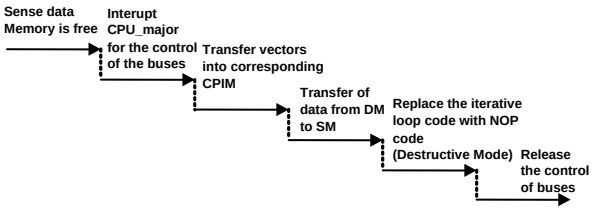

Figure 15. Information transfer stages

3.5 Intelligent in the Context of the Proposed CIM Architecture

Any reference to intelligence in the context of our proposed architecture is limited to the definition; an Intelligent System (IS) is a system, which learns how to act towards a certain situation in order to reach its objectives by using experiences and knowledge gained previously.

From the above statement, we can conclude that an IS has two fundamental characteristics learning and serving. Typically, an IS achieves its objective through knowledge and experience acquired through a learning process, which is something that has happened to the IS during some moment of its existence. It includes the situation that occurred, the action done, and the results.

The distinction of the CIM from the existing PIM systems is its run-time learning capability to gather knowledge on current program execution profile. During the first execution cycle, the *observer* collects information about the desired loop. Eventually information related to the vectors. The information collected cover the address range that includes: starting address, ending address, addresses where the computed results to be stored and

addresses of instruction block that corresponds to the situation with the vectors job size, job nature and the number of iteration in the loop. Once the task is completed the vectors component loaded into CPIM registers and the related data transferred from main to corresponding CPIM memory. The vector instruction block is used to apply bypass. The bypass effectively removes the set of instruction related to the iterative loop. The bypass provides a significant difference in problem solving approach among conventional architectures and the proposed intelligent architecture. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions to solve the problem. On the other hand, the proposed CIM architecture learns by observation/experience. It does not need a set of instructions during serving stage to perform a specific task.

Vector loading into the CIM registers demonstrates a learning stage, the CPIM trained for a particular situation. During the course of executing the same program, serving stage, due to use of bypass with same data set or in the presence of new data set system demonstrates the ability to use experience and knowledge gained previously. Thus, the proposed system obeys the basic rules developed for an IS.

3.6 Co-operative in the Context of the Proposed Architecture

Any reference to co-operative in the context of our proposed architecture is limited to the definition; a co-operative system is the one which performs its functions through co-operation from all the components of that particular system. In the context of the computer a time sharing system is a co-operative system in which CPU performs a specific task through the co-operation of memory, bus and the peripherals.

In co-operative systems the task is split hierarchically into independent subtasks and co-ordination is performed when assembling partial results from these subtasks. Co-ordination is a synchronous activity which is the result of a continued attempt to construct and maintain a shared conception of a problem.

From the above statement, we can conclude that a co-operative processing is computing which requires two or more distinct processors to complete a single task. Co-operative processing is related to distributed processing where two or more distinct processors are requested to complete a single task.

3.6.1 Co-operative in the Context of Processing

In the proposed architecture, we have a heterogeneous mixture of processors: CPU_major and memory processor which is the CPU_minor. The main processor is a conventional processor. It is backed up by a deep cache hierarchy and suffers a high latency to access memory. The memory processor is tailored to the specific needs of the inner most loop of the program being executed makes it more powerful from the organizational point of view for specific needs as compare to CPU_major.

We think the best way to exploit the term co-operative in the processing scenario is to exploit the heterogeneity of the system by partitioning an application into two parts: one that benefits mostly from the high capability of the main processor, and other that benefits mostly from the low latency high bandwidth access to the memory (memory-intensive). We call this approach co-operative processing. In previous work on these systems [3, 4, 5, 6, 9, 10, 11, 16], the programmer is expected to identify and isolate the code sections to run on the memory processors. In addition, previous works have largely focused on executing sections of code on only a set of identical memory processors. Such an approach is often not much different from running code on a conventional parallel processor.

The proposed architecture uses automatic task partitioning between iterative and non-iterative tasks to intelligent memory systems to exploit the heterogeneity of the processors in the system. This task partitioning forms the basis for an architecture that complements the main CPU's activities and co-operates in expediting the overall task, while encouraging the overlap execution of CPU_major and CPU_minors.

3.6.2 Co-operative in the context of memory

The possibility to integrate memory and logic on the same chip (intelligent memory architecture) has a large impact on system integration and performance, memory sizes, on-chip memory interfaces and memory structures. Most important is that the fabrication of the chip can be optimized for the most suitable process and designer can adjusted the bandwidth and memory size to its application.

The possible cases in which memory is wasted, when the memory system is composed of commodity devices are,

1) The granularity of the memory devices forces more memory.
2) The memory bandwidth forces parallel access to memory devices.

Memory bandwidth of a commodity device BW_{device} can be calculated as,

$$BW_{device} = IO_{width} \times f_{IO} \quad (23)$$

Where IO_{width} is the width of the memory device and f_{IO} is the data *IO* frequency. However, *IO* frequency affected by the two factors, one is page miss penalty and other is load capacitance. Page miss penalty is directly related to the application and load capacitance is related to the length of the off-chip buses. A reported difference of a factor of 10-50 exists between on and off-chip load drives [17]. Merging logic and related data on the same chip, reduces the need of the off–chip buses that reduces load capacitance of the system (shorter wire lengths that connect the logic and related data) effectively increases data *IO* frequency f_{IO}. Another factor, which influences the memory bandwidth, is the width of the memory device IO_{width}. In commodity devices, the IO width is limited to certain number (16-24) of pins due to packaging. Since the memory interface is on-chip, the total pin count of the chip is not limited, likely reduction in the pin count, and pad-limited designs may be transformed into non-pad limited ones. Obviously intelligent memory architecture can offer a finer granularity in memory sizes and required bandwidth than commodity device. Low power is another important issue, which can be positively influenced by intelligent memory. Power can be mainly optimized by minimizing *IO* power or deactivating idle memory banks. In addition, inductivity caused by the package and the tracks is also eliminated, thus system noise immunity is enhanced.

3.7 Proposed CIM Architecture

The basic CIM is shown in Figure 16. The architecture differs from CPIM in terms of approach; instead of von Neumann where instruction and data are stored in a single memory, it requires a Harvard approach towards memory where separate memories are used for instruction and data storage. This approach may simplify read/write mechanism, particularly as programs are normally read during execution, while data might be read or altered. Also establish a path for the extraction of vector components by monitoring the activities operating on the address and data buses.

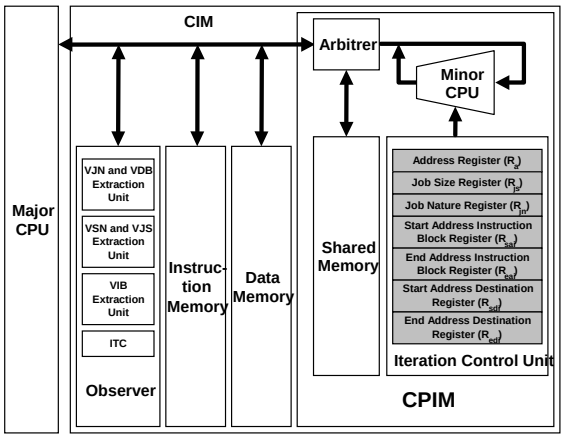

Figure 16. CIM architecture

Our proposed architecture has the following characteristics:

- The shared memory capacity is large enough to hold large data frames synonymous with high resolution image frames.
- The overhead associated with the time it takes to fetch and execute the instruction in a specific program loop is eliminated.
- No need for special instructions as required in the case of coprocessor.
- CPU_major (Major CPU) can continue with other operations while the CPIM is completing its allocated task.

The major characteristics that make CIM distinctive from the existing PIM systems, is its run-time learning capability to get intelligence for the current program execution profile.

The CIM's additional building blocks are described below.

3.7.1 Observer

The detection of iterative loops is conducted by the *observer* having additional knowledge of the location of the CPIM with reference to their operational capability. The observer performs the following jobs:

- Extraction of vectors that characterize the iteration using the three different extraction units.
- Transfer of vector components with the related set of data into the CPIM. This is done using the ITC.
- Removal of selected/corresponding iterative loop from the main stream.

3.7.2 Iteration Control Unit (ICU)

The ICU provides an instruction set for CPU_minor. It consists of five different registers registers, namely address register (R_a), job size register (R_{js}), job nature register (R_{jn}), start address instruction block register (R_{sai}), end address instruction block register (R_{eai}), start address destination block register (R_{sdi}) and end address destination block register (R_{edi}).

Figure 17 and 18 demonstrate the CIM system activity during learning and serving stage respectively.

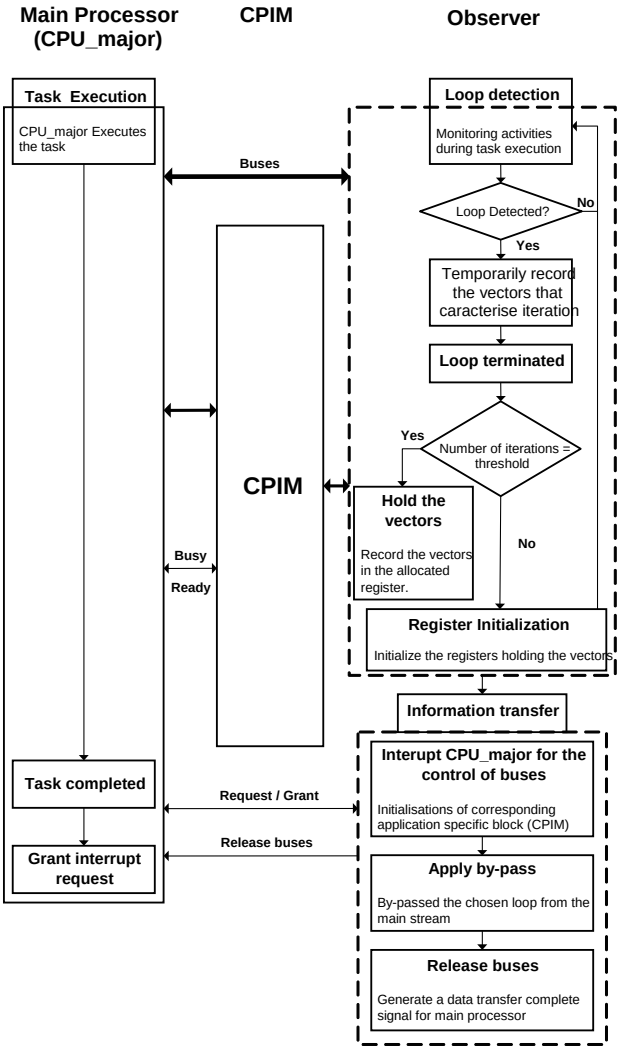

Figure 17. Inter-block communication in learning stage

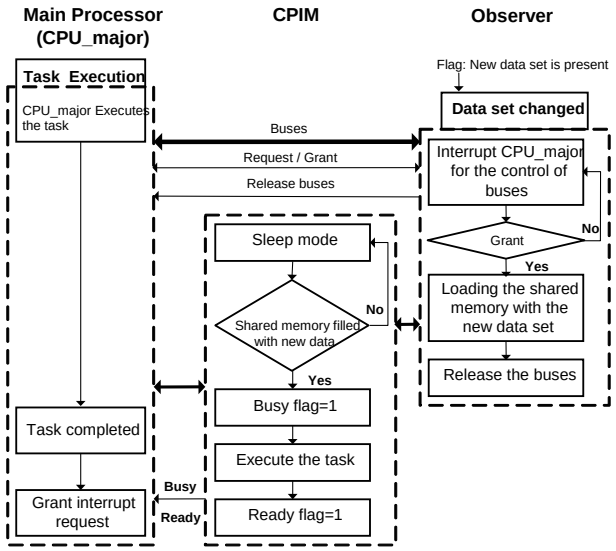

Figure 18. Inter-block communication in serving stage

3.8 CIM Computation Time

Assuming that the program execution profile exhibits equally sized grains of computation, then the CPU_major will be presented with equal sized instruction block to execute. The instruction execution cycle time for each job is assumed to be the same and denoted by Δt . Figure 19 illustrates the learning stage. In the presented scenario, Job4 is the only iterative loop with N iterations and no data dependency is considered.

In conventional architecture, CPU_major ends its task, comprising of job1 to Job6, at t_6. Hence

$$t_6 = 5\Delta t + N\Delta t \tag{24}$$

In the proposed architecture, the completion time of the said task extended to $X\Delta t$ which is the information transfer time during its learning stage. Therefore,

$$t_6 = 5\Delta t + N\Delta t + X\Delta t \tag{25}$$

The impact of $X\Delta t$ on system performance is reflected during a single learning stage. Rest of the time, the application specific block takes care of the by-passed iterative loop by continuous reference to its own registers and memory. Then,

$$t_6 = 5\Delta t + N\Delta t \tag{26}$$

During serving stage, Figure 20 shows the same task is repeated and finished at t_5.

Hence

$$t_5 = 5\Delta t \tag{27}$$

The completion time for the task at t_5 is therefore far shorter than during the learning stage. The time difference $N\Delta t$ shows that the proposed architecture reduces the completion time for the task significantly and hence contributing to the overall speedup.

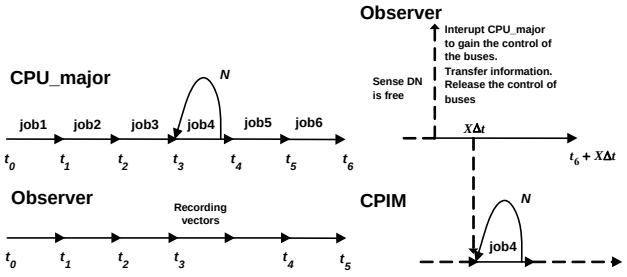

Figure 19. Learning stage: Identification of loop and data transfer

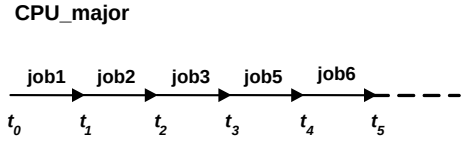

Figure 20. Serving stage

4. COMPARISON MATRIX

The global comparison matrix shown in Table 1 provides a way to demonstrate the facts that the proposed architecture is different compared to the previously proposed/existing architectures, where PIM chips act as a co-processor in memory, which executes codes when signaled by the host or CPU_major.

Table 1: Comparison matrix (● Validity-○ Non-validity)

FEATURES/ISSUES	CO-PROCESSOR	CPIM	CIM
Has its own specialized instruction set	●	○	○
Tailored to the specific needs	●	●	●
Compatibility	●	○	○
Software overhead for the distribution of workload	○	●	○
Hardware overhead for the distribution of workload	○	○	●
Communication policy between main CPU and corresponding machine			
I) Request and service	●	○	○
II) Learn and serve	○	●	●
Mode of data/information transfer			
I) Cycle stealing	●	○	○
II) Burst transfer	●	●	●
Implementation issue merged logic and memory on the same chip with special emphasis on cost, performance/speed and density	○	●	●

5. CONCLUSION

Memory systems are primary bottleneck in the performance of high speed computers. Advances in VLSI technology are enabling the processor-memory integration to bridge this gap, is also a key driver in the innovation of a new PIM concept.

In this paper, CPIM and CIM architectures have been proposed as a viable solution to address this problem. The CPIM is used as the basic building block of the CIM. A Learn and Serve policy forms the basis for the proposed intelligent CIM architecture. The major characteristics that make proposed CIM architecture distinctive from the existing PIM systems is its run-time learning capability to gather knowledge on current program execution profile. Real time task partitioning is conducted by the hardware on the basis of the knowledge that the system acquired from the learning cycle rather than pre-processed by the software.

The general method behind the implementation of intelligent memory architectures is to associate a large number of processing hardware components with the data storage hardware elements of the memory. The proposed CIM system achieves this for the memory intensive applications by integrating processing logic close to the related data into the CPIM module.

REFERENCES

[1] W. Stallings, "Computer Organization and Architecture", Prentice Hall, 6th Edition, 2003.

[2] J. Hennesy and D. Patterson, "Computer Architecture: A Quantitative Approach". Morgan Kaufmann Publishers Inc., CA, 2003.

[3] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas and K. Yelick. "A Case for Intelligent RAM: IRAM", IEEE Micro, Vol. 17, No. 2, pp. 34-44, April 1997.

[4] D. Elliot, M. Stumm, W. M. Snelgrove, C. Cojocaru and R. McKenzie, "Computational RAM: Implementing Processor-in-Memory", The IEEE Design and test of Computers, Vol. 16, No. 1, pp. 32-41, March 1999.

[5] E. Waingold et al., "Baring it All to Software: Raw Machines", IEEE Computer, Vol. 30, No. 9, pp. 86-93, September 1997.

[6] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally and M. Horowitz "Smart Memories: A Modular Configurable Architecture", The 27th International Symposium on Computer Architecture, pp. 161-171, British Columbia, Canada, June 2000.

[7] A. Saulsbury, F. Pong, A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration", Proceedings of the 23rd International Symposium on Computer Architecture, pp. 90-90, Philadelphia, PA, USA, May 1996.

[8] W. Wulf and S. Mckee, "Hitting the Memory Wall: Implication of the Obvious", ACM Computer Architecture News, Vol. 23, March 1995.

[9] M. Oskin, F. T. Chong and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory", The 25th IEEE International Symposium on Computer Architecture, pp. 192-203, July 1999.

[10] Y. Kang et al., "FlexRAM: Toward an Advanced Intelligent Memory System", The IEEE International Conference on Computer Design, pp.192-201, October 1999.

[11] J. Darper et al., "The Architecture of DIVA Processing-in-Memory Chip", The 16th ACM International Conference on Supercomputing, pp. 14-25, June 2005.

[12] Y. Kang, J. Torrellas and T. S. Huang, "An IRAM Architecture for Image Analysis and Pattern Recognition", The 14th International Conference on Pattern Recognition, Vol. 2, pp. 1561-1564, August 1998.

[13] J. Parker, "Algorithms for Image Processing and Computer Vision", John Wiley and Sons Inc., 1997.

[14] Z. Ahmad, "Cooperative Intelligent Memory", PhD thesis, University of Hertfordshire, April 2007.

[15] K. Hwang and A. Briggs, "Computer Architectures and Parallel Processing", McGraw-Hill Book Company, London, 1984.

[16] R. Manohar and M. Heinrich, "A Case for synchronous Active Memories", ISCA 2000 Solving the Memory wall Problem Workshop, pp. 1-10, June 2000.

[17] D. Keitel and N. Wehn, "Issues in Embedded DRAM Development and Applications", The 11th International Symposium on System Synthesis, pp. 23-28, December 1998.