International Journal of Research in Engineering, Science and Management (IJRESM)
www.ijresm.com Volume-1, Issue-2, February 2018

Virtual Pipelining- A Reliable Processor Architecture

Rakesh[1], M.L. Sudheer[2]

[1]Student, Dept. of Electronics and Communications, University Vishveswaraya College of Engg., Bangalore, India
[2]Professor, Dept. of Electronics and Communications, University Vishveswaraya College of Engg., Bangalore, India

***Abstract*— The design of an architecture for an integrated digital processor requires attention to details of processing algorithms. Even a simple electronic gadget consists of a giga bytes of software which requires services both in terms of hardware and software. Many architecture designs are proposed to solve current requirements in processor industry especially in control unit of processors. Current architectures focus on centralized control unit which directly reduces reliability of that unit. Today's VLSI industry had grown to such a level where billion of transistors are placed on a 1cm2 silicon wafer. This causes increase in current density as well as large power dissipation, which greatly affects the reliability of these units. At the same time, reliability of future processors is threatened by the growing fragility of individual components. Large scale studies of have already shown that existing processors are susceptible to error rates that are orders of magnitude higher than previously assumed. Current architectures focus on delivering high performance at low cost; lifetime device reliability is a secondary concern. Hence there is a need for an architecture which enhances the reliability as well as performance.**

Index Terms— BSU, CMP, crossbar, ISA, microarchitecture, multicore, pipelining, single-point-fault, TMR, viper

I. INTRODUCTION

Traditional solutions to enhance performance are by direct use of multiple cores, which is less power efficient and also not considered the reliability concerns.

As the new trend where the real world is expecting performance as well as reliability, the in-order core solution is not fulfilled the requirements since it is more susceptible to more hardware faults. To cope with these hardware faults and to increase reliability, bullet-proof and stage-line architecture are proposed. In these architecture, the dedicated direct link between the modules which come in pipeline flow. These modules are bound by a crossbar-a reliable communication network, which establishes a path for both control and data. Because of this flexible link between modules of pipeline, effective utilization of hardware modules is achieved. Again the reliability is enhanced at a single core level but still communication between multiple core remains hardwired. Again the crossbar between modules also hardwired and has centralized control unit contributing to degrade in reliability.

In Fig. 1, the reliability and performance of such solutions are compared for throughput of a chip comprised of about 2 billion transistors as a function of the number of hardware failures in the device. A chip of this size could fit 128 standard in-order cores, 42 in-order cores in a TMR configuration, 27 bulletproof pipelines or 30 Stage Net pipelines (the latter two having a fault-free throughput equivalent to about four in-order cores). It demonstrates that the maximum performance of the unprotected design decreases steeply as the number of faults increases, while the performance of TMR is extremely poor throughout. The two hardened micro architectures can better cope with hardware failures, but as they rely on centralized logic, they still suffer significant performance degradations when subjected to a large number of faults.

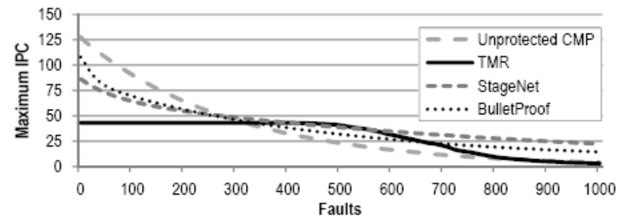

Fig. 1. Statistics showing reliability and performance as a function of faults.

At this end, we introduce Viper, a new architecture that decouples the functionality of a pipeline and its control logic. By removing the dependencies between all parts of a core, it becomes possible to build a highly redundant, error-resilient design that contains no single point of failure. Specifically, viper provides following contributions:

1) Viper provides a novel decoupled architecture that can reconfigure itself around hardware errors.

2) It proposes a new execution paradigm where instructions are split into bundles, each with a list of underlying tasks it needs to complete. The decoupled hardware components then complete these tasks.

3) Viper has a fully distributed control logic design, which allows performance to degrade gracefully without any single point of failure in the system.

Viper outperforms other reliable designs and surpasses the performance of a CMP built from in-order cores after only 160 faults in a two billion transistor chip

II. VIPER HARDWARE ORGANIZATION

Viper is based on a distributed execution engine that is dynamically configured to route instructions towards functioning hardware components. This allows Viper to degrade performance gracefully when subjected to hardware errors.

Viper is a service-oriented micro architecture, where instructions are presented as customers that use hardware components to complete an ordered sequence of services. For instance, a sequence of such services for a simple add instruction - add %al, [%ebx] - could be: fetch/decode instruction, retrieve value from registers, load memory value, add two operands, and write the result back to a register and, "compute the address of the next instruction. From Viper's perspective, an ISA consists of the set of services required by its instructions. Instead of pushing instructions through paths defined at design time, as classic architectures do, Viper relies on a flexible fabric composed of hardware clusters. These

clusters are loosely coupled via a reliable communication network to form a dynamic execution engine.

Viper can be partitioned into two parts:

1. A sea of redundant hardware clusters: hardware functional units connected through a reliable communication medium. Each cluster can perform some of the services required to execute instructions in the ISA.

2. Bundle Scheduling Units: memory elements that contain the state of in-flight instruction bundles and store the data necessary to schedule and organize the hardware clusters that form a virtual pipeline. A live BSU entry does not contain instructions or operands, but only the information required to control the bundle's execution.

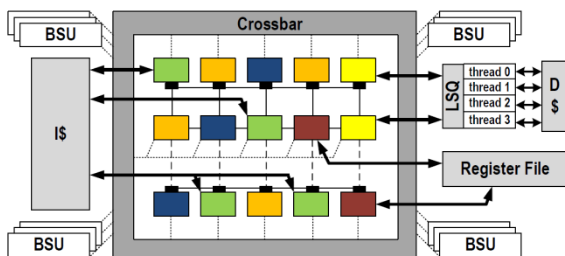

Fig. 2. Viper components and its organization. Every module is interlinked through crossbar. Some of the clusters, such as the ones capable of fetching instructions, have special connections to external hardware elements

The Fig. 2 presents a simple Viper design organized in a mesh, where each colored service is replicated in multiple identical clusters. BSUs are connected to the sea of clusters through a crossbar, which allows each BSU to interact with all clusters in the execution engine. Clusters that need access to external modules are connected to them through dedicated links. For instance, clusters capable of fetching instructions are directly connected to the instruction cache, and the register file and load/store queues are placed near clusters that need fast access to these units. Finally, clusters that support the write memory operations service are connected to the load/store queue to allow stored values to be written to memory once the related bundles are committed.

Note that, depending on the layout of the hardware, such special links might not have uniform communication latency. The easiest way to prepare your document is to use this document as a template and simply type your text into it.

III. Viper Execution

For understanding how viper executes an instruction in its regular operation we use the design provides only six services: fetch, decode, rename, execute, commit, and write-back and memory operations in our succeeding example. Viper's operation are be grouped as follows:

1) Bundle Creation
2) Virtual Pipeline Generation
   2.1) Service Proposal
   2.2) Service Assignment
   2.3) Configuring the Sea of Clusters
3) Operand Tags Generation
4) Bundle Termination
   4.1) Memory Operations
   4.2) Managing Bundle Sequence

1) Bundle Creation

In this example, we assume that an instruction bundle (with BID 5) has already successfully determined the starting address of the next bundle. Therefore, program execution proceeds to the next basic block, which starts at address 0x4013d2. Since a not-taken conditional branch concludes bundle 5, the NPC (Next Program Counter) field of its BSU stores the (correctly) predicted location, as shown in Figure 3.b. Because the PC of the next bundle is available, but no BSU has been assigned to it (the field Next BSU is empty), bundle 5s BSU assigns an available BSU entry to the following bundle, as shown in Fig. 3(c). When a bundle is first assigned to a BSU entry, the only two pieces of information available are: 1) the BSU entry number of the previous bundle and 2) the PC of the first instruction of the bundle. The former is needed because live BSU entries form a chain of in-flight bundles. This allows the system to track correct control flow and to commit bundles in order. The latter information is needed by the fetch component, as we discuss shortly. Since a new set of clusters is needed to form the virtual pipeline for the new bundle, the newly assigned BSU marks all required services as unassigned.

2) Virtual Pipeline Generation

Once the bundles are formed, each bundle is associated to a BSU and a new BSU entry assigned to control the execution of a new bundle is in charge of constructing a virtual pipeline capable of providing at least all services required by its instructions. Virtual pipeline generation consists of selecting which hardware clusters will collaborate in executing a bundle. Since using a centralized unit to perform this procedure would constitute a single point of failure in the system, Viper adopts a distributed mechanism to generate virtual pipelines. This negotiation mechanism is based on service proposals: clusters independently volunteer to execute services for a bundle in a live BSU. Similar to traditional processors, a path is needed for flow of both data and control during execution of the bundle. Hence we need to form tis path which is referred as Virtual pipeline in viper.

2.1) service proposal

In this stage, the new BSU which is just created is going to flood the requirement of resources on the cross bar network.
The resources which are available for providing service will reply back to the BSU. Depending on the routing and placement of clusters from BSU, the proposal message may suffer from different latency in each path and hence reply also arrive at different time instant. For instance, in Fig. 3(d) we show two clusters, F0 and W2, proposing their services to the bundle with BID 6. This may occur because clusters initiate the proposal negotiation independently, and therefore a BSU might receive multiple service proposals at the same time.

As soon as the cluster replies to BSU saying I'm ready to provide service, it changes its local status from idle to pending and waits for an award message from the BSU. A service proposal is not binding until a BSU notifies the proposing

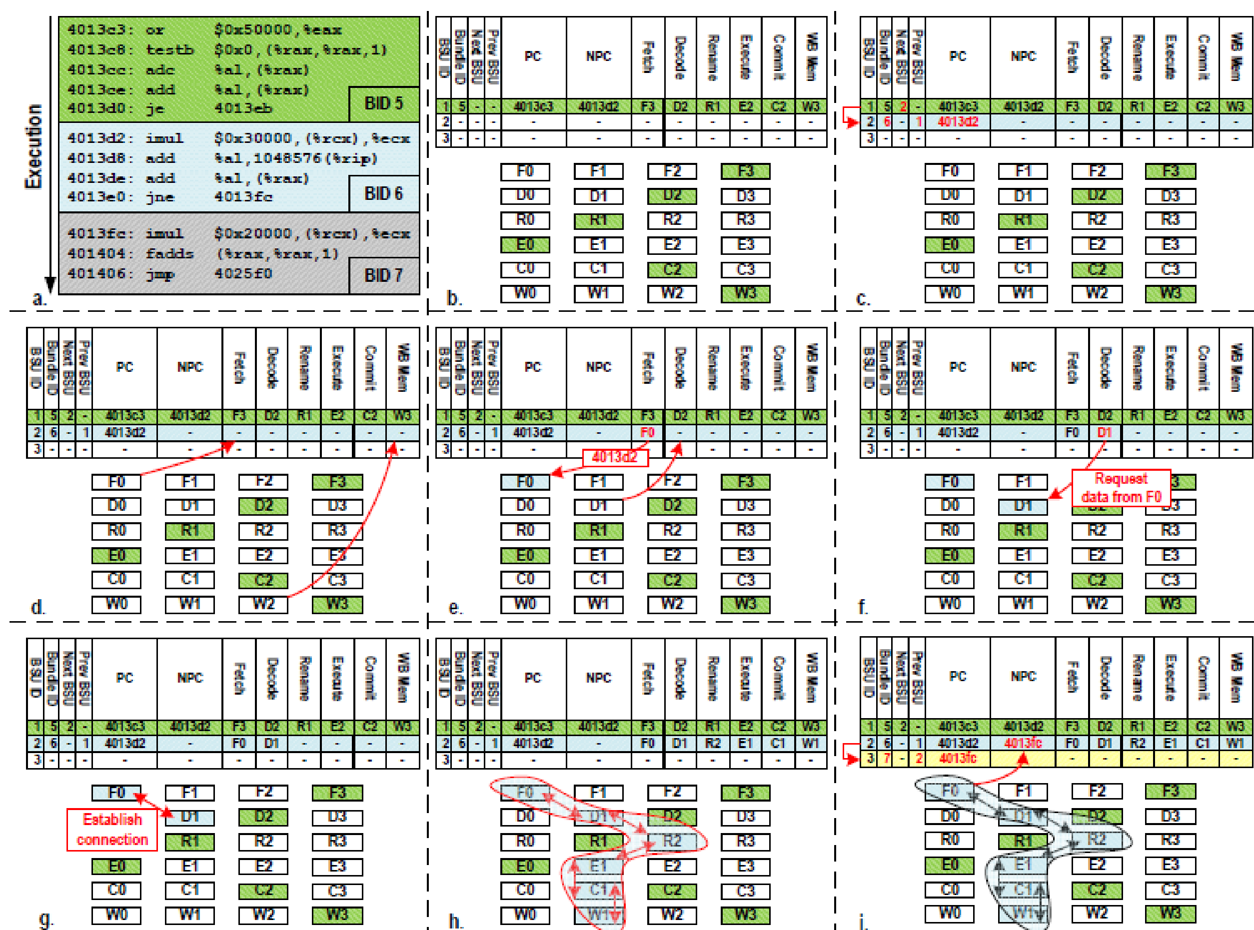

Fig. 3 (a) Virtual pipeline creation process for the second bundle in frame b) The BSU for bundle 5 creates the next bundle when the address of the following basic block becomes available in next PC field. c) The new bundle is created in an available BSU. d) Functioning and available hardware clusters in the system propose their services to the new bundle. e) Cluster F0 is selected to become part of the new virtual pipeline. f) A subsequent proposal from D1 is accepted. Clusters are also notified by the BSU about the other clusters composing the virtual pipeline. g) The clusters are configured to establish communication paths. h) After the configuration, a virtual pipeline is formed. i) Finally, as F0 detects the last instruction in the bundle, it updates the BSUs NPC field, which allows the next bundle to begin

party; if no service award is received within a timeout period, the cluster considers its proposal rejected, and the service negotiation sequence is re-initiated.

2.2) Service assignment

Here once BSU receives the reply from clusters it has to provide the details of the bundle to the clusters whose proposal was accepted. Explaining with example, when BSU 2 chooses F0 to be included in its virtual pipeline, it records that this cluster will accomplish the fetch service for its bundle. Besides the notification that a proposal has been accepted, confirmation messages carry information needed by the clusters to perform their service. Such information consists of either data fields directly stored in the BSU or routing information needed to retrieve data from other clusters. The former situation is shown in Fig. 3(e) as the BSU sends a notification to F0 that its proposal was accepted, it also forwards to it the first memory address of the bundle with BID 6.

2.3) configure the sea of cluster

Once BSU keeps on filling the resource entries, it should establish connection between the hardware clusters that going to take part in that virtual cluster. In Figure 3.f we show the BSU awarding it’s decode service to D1. This cluster is told which cluster will fetch the instruction bundle, in this case F0. D1 then establishes a connection with F0 through the reliable network, as shown in Fig. 3(g).

All services are similarly assigned in an ordered fashion and, as the BSU service list is filled, the sea of cluster is configured to generate a complete virtual pipeline through the network, as shown in Fig. 3(h). Viper can concurrently configure several independent active virtual pipelines, since the BSUs and execution clusters operate autonomously. Multiple virtual pipelines can work on a single program (as shown in the example), or can simultaneously execute multiple threads.

3) Operand Tags Generation

To resolve the data dependency between bundles, and enhance efficient parallel processing viper provides a tag generation mechanism. Each live BSU entry stores three tag versions for all the architectural registers in the ISA: input, generated and output. Compared to classical renaming schemes based on mapping architectural to physical registers, the input and output tags can be seen as two snapshots of a classic

rename table: the first before and the second after the execution of the entire bundle.

4) Bundle Termination

Once the execution of a bundle is completed, BSU going to release all resources that are currently in its virtual pipeline. The term completion of execution of a bundle" is nothing but 1) all clusters assigned to its virtual pipeline finish servicing its instructions; 2) all preceding bundles belonging to the same thread have already terminated.

4.1) Memory Operations

Once required conditions are met for bundle termination, the next phase is to detach the load store queue of that BSU entry and making it free so that other tasks can make use of that memory location. Since multiple bundles from the same program can execute in parallel, the load and store queues might receive misordered memory requests. This could cause a problem, as the forwarding logic in the load and store buffer might mistakenly: 1) forward to load instructions values produced by later stores or 2) receive a sequence of stores that does not reflect the program order. As the memory queue cannot dynamically address these issues, they are resolved by clearing all entries in the thread's load and store queue and canceling the execution of the conflicting bundles. In order to ensure forward progress, the oldest canceled bundle replays its execution starting from its original PC but is forced to include only one instruction.

4.2) Managing Bundle Sequence

Each live BSU maintains starting addresses for both its bundle and the one immediately following. This latter value is provided by the clusters performing the fetch service, as they can recognize the end of a bundle at control flow instruction such as jump. Such clusters communicate the starting address of the next bundle back to their BSU, as shown in Fig. 3(g): even before bundle 6 terminates, F0 can predict the starting address of the following basic block - 0x4013fc in our example - updating the NPC field of the BSU with this address. With this, the BSU can generate a new bundle (in our example with BID 7), and continue program execution.

IV. Handling Exceptional Events

Viper provide a special care to handle the exceptional events such as memory misprediction, trap flag generation and runtime failures. Most processors require several cycles to resolve the target of instructions that modify control flow. This delay might cause the system to start processing instructions from an incorrect execution path: these instructions need to be flushed as soon as a control flow misprediction is detected. Viper does same thing by releasing the allocated resources and deleting that BSU entry only after it gets confirmation from all clusters that are released. Similarly Interrupts, exceptions, traps, and page faults must be handled with particular attention. Without modifying the bundle termination procedure, these events can cause the system to deadlock. For instance, an instruction triggering a page fault might prevent its entire bundle from terminating. To overcome this issue, a bundle affected by one or more of these special events is canceled and split in multiple bundles, each including a single instruction from the original basic block. The bundle containing the faulty instruction will then steer program execution to the correct software handler. Other cases where bundles must contain only a single instruction are system calls and uncacheable.

Runtime failures: to handle these faults, viper monitors a memory accesses. As Viper's goal is to maximize processor availability in the face of hardware faults, we assume that other mechanisms will detect faulty hardware components. In our failure model, we assume that a hardware component detected as faulty can be disabled. Compared to previous solutions, our design provides an additional advantage to online testing, as it does not require interrupting program execution. A cluster detected as faulty for a particular service is disabled for that service, and it will not propose to complete that service for any BSU.

V. A Solution Reduce Full Flush

As stated earlier Bundles are the larger collection of instructions to which a BSU entry is created. In viper architecture, these are treated as customers who require services to complete the execution of the instructions present in that bundle. Every BSU entry contains a PC (Program counter-Holds the starting address of that bundle) and NPC (Next program counter-Holds the starting address of the next bundle) field which is communicated to all h/w clusters which are taking part in the execution of that bundle i.e. the cluster present in virtual pipeline of that BSU entry. Here an intelligence is required to predict or estimate the value of PC and NPC fields.

The prediction of the starting address of bundle is directly depends on the ISA of the processor being built. If we consider a simple GPP, there are specific instructions that alter the sequential flow of program execution such as conditional and unconditional branch instructions, software interrupts, special Hardware service calls (DOS) etc. Unconditional jump instructions are straightforward and hence it is easier to predict the next bundle address. But in case of conditional jump there is always a 50% probability of branch misprediction. This is because the target address for branching is known only after the specified condition is checked.

If we follow the simple bundle creation on the basis of branching instruction, the viper is going to flush all its current entries, and reinitiates bundle creation from the branched address. Again the assumed way to create bundles is sequential, once misprediction happens viper flushes complete succeeding BSU neglecting whatever the state of those BSU entry.

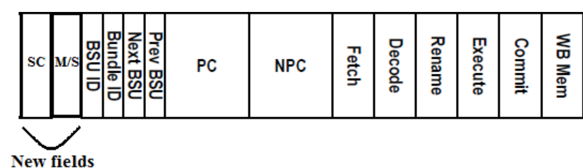

Fig. 4. Adding fields to BSU entry

Here we present a simple solution by the concept of master slave concept. In our 1st solution we are required a fetch and

decode unit dedicated for this purpose. This is because the address of the next instruction can be known only after decoding of the current instruction.
To implement this concept we are adding two new fields in a BSU entry. We are going to name them as M/S and Slave count.

M/S = 0-for Master and contains BSU id of master for slave BSU entry.
SC =Assume 2 bit. Hence at most of 4 slaves.

With the addition of these new fields, we are going to alter the NPC field by extending it have two addresses, with this we are ready to explain our solution.

The following pseudo-code gives the flow of our algorithm for bundle creation.

```
Fetch_instructions_from_I$( );
Decode( );
If (Decoded_inst== branching_instru)
        If(Branching==unconditional Branch)
        Provide_NPC_value( );
        Else
        Provide_possible_NPC_value(); /*here NPC holds 2 addresses for both true and false case*/
        If(only_default_master_is_alive)/*default master=0 the boot BSU*/
    Create_master_bsu(true); /*here after completion of master bsu, we continue to create slave bsu' s until there is need for another master bsu requirement and SC is reached its maximum*/
                Create_master_bsu(false);
        Else wait_for_flush(make_other_master_to_slave );
Loop_back( );
```

Here we created two bundles for both true and false branch. Once branching is estimated the BSU of branched entry details are broadcasted and hence the other master and its slave BSU entries are flushed hence leaving only one master active. Again the remaining master becomes the slave of default BSU and makes its slave to same.

With this we can avoid complete flushing and reduce the latency in execution.

To improve the above concept we can make use of internal cluster to perform the same task. Since virtual pipeline creation is sequential, once fetch and decode unit is assigned to a BSU, they have to wait until all other clusters are ready for execution of that bundle. During that time, the dedicated unit is made to communicate through the crossbar to these units and hence assign the task of branch prediction. By proving special handshaking and dedicated path, this bundling process can accelerated to reduce memory-miss latency in processors.

VI. An Application

LSI Axxia-Communication Processors family is intended for networking and datacenters (cloud). These processor family utilizes Virtual Pipeline® technology for High performance fast data path processing (IPv6, IPSec, etc.) and offload for up to 20 Gbps performance.

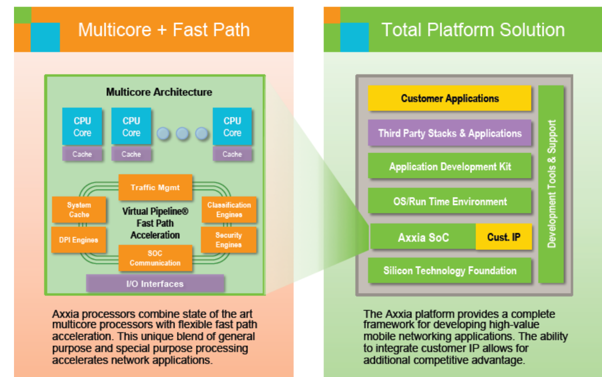

Fig. 5. Axxia platform for networking

In Fig.5, it is shown that where the concept of virtual pipelining is utilized. The technology is implemented for effective binding of the service engines which are assigned once services are requested. Here the main purpose is to accelerate the data path by providing only requested services.

VII. Conclusion

In today's world everyone looking for an reliable, high performance modules to enhance the product lifetime. Since viper best suits these situation its application is numerous. Since it provides a flexible reconfigurable and dynamic resource allocation, this can be used to eliminate redundant hardware in current designs and hence it helps to save space as well as power. Since it is fully distributed control logic it eliminates the single-point-of-failure and adds reliability. A program can successfully finish as long as its required services can be executed by a dynamic collection of the available components. By construction itself, viper enhances fault immunity while maintaining high performance.

Future work will include development of fast and reliable protocol for crossbar network to reduce IPC delay. Viper's performance could be improved by developing more efficient and faster techniques for building virtual pipelines and handling exceptions. Finally, by application of Viper's flexible execution engine to both gpp and dsp one can get more features from the same cost.

References

[1] Andrea Pellegrini ,Joseph L. Greathouse, Valeria Bertacco, Viper: Virtual Pipelines for Enhanced Reliability, *ISCA*, Portland, OR, 2012, pp-344-355.
[2] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway (2003).The AMD Opteron Processor for Multiprocessor Servers. *IEEETrans Micro*, *23(2), pp.*66-76.
[3] Brad Hallisey (Dec. 2012), Building a Virtual World: The Pipeline and Process, *IEEE Trans. Computer*, vol. 45(12), pp. 90-92.
[4] Wei Wang, Axxia® Communication Processor-Networking overview, LSI corp, Shanghai, China, Oct-2012.