Viper: Virtual Pipelines for Enhanced Reliability

Andrea Pellegrini Joseph L. Greathouse Valeria Bertacco
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI, USA
{apellegrini, jlgreath, valeria}@umich.edu

Abstract

The reliability of future processors is threatened by decreasing transistor robustness. Current architectures focus on delivering high performance at low cost; lifetime device reliability is a secondary concern. As the rate of permanent hardware faults increases, robustness will become a first class constraint for even low-cost systems. Current research into reliable architectures has focused on ad-hoc solutions to improve designs without altering their centralized control logic. Unfortunately, this centralized control presents a single point of failure, which limits long-term robustness.

To address this issue, we introduce Viper, an architecture built from a redundant collection of fine-grained hardware components. Instructions are perceived as customers that require a sequence of services in order to properly execute. The hardware components vie to perform what services they can, dynamically forming virtual pipelines that avoid defective hardware. This is done using distributed control logic, which avoids a single point of failure by construction.

Viper can tolerate a high number of permanent faults due to its inherent redundancy. As fault counts increase, its performance degrades more gracefully than traditional centralized-logic architectures. We estimate that fault rates higher than one permanent faults per 12 million transistors, on average, cause the throughput of a classic CMP design to fall below that of a Viper design of similar size.

1 Introduction

Through tremendous efforts, Moore's Law has continued to hold, allowing denser transistor integration in each successive silicon generation. Unfortunately, this leads to increased current and power densities, negatively affecting the reliability of already fragile nanoscale transistors [6].

The reliability of future processors is also 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 [22]. Furthermore, leading technology experts warn that device robustness may decline even further for technology nodes below $32nm$ [6, 34].

As permanent failures in transistors and wires become more likely, architects' design priorities must shift to consider robustness as a primary constraint. Traditional solutions for high-availability and mission-critical computers address reliability through dual- or triple-modular redundancy [2]. However, these solutions are far too costly to be adopted in mainstream commercial systems, which will require new low-cost architectures that can survive a large number of hardware malfunctions.

Modern single-chip devices often contain several processors, providing a straightforward method for increasing reliability. In such designs, faulty cores can be disabled without affecting the behavior of other portions of the machine. This solution requires limited engineering effort and does not significantly hinder either the performance or the power budget of fault-free systems. However, because a single fault can disable large portions of the design, this technique does not scale well to higher fault rates [24].

Recent research on reliable processors has focused on online tests [23], fault isolation [12], redundant functional units [31] and runtime checks [1, 20]. Though these solutions improve reliability, they still rely on centralized control logic: a single point of failure that can allow one fault to disable an entire core.

To compare the reliability and performance of such solutions, we statistically computed the maximum expected 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 [31] or 30 StageNet pipelines [12] (the latter two having a fault-free throughput equivalent to about four in-order cores). Our estimation, presented in Figure 1, 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 microarchitectures 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.

In classic architectures, hardware components are tightly interdependent for performance reasons. This challenges our ability to isolate faulty components, as an error in one part of the chip can effectively disable all other components that depend on it. We must therefore relax both these con-

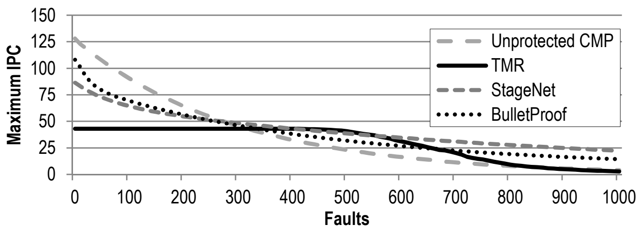


Figure 1: Comparison of maximum throughput achievable for a CMP of two billion transistors. The four analyzed configurations are: 128 in-order cores with no extra fault-tolerance hardware, 42 sets of TMR in-order cores, 30 StageNets [12] or 27 BulletProof pipelines [31].

straints if we wish to design processors that can tolerate more than a few hundred faults per billion transistors.

To 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, this work makes the following contributions:

- We present a novel **decoupled architecture** that can reconfigure itself around hardware errors.
- We propose 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.
- We demonstrate a **fully distributed control logic** design, which allows performance to degrade gracefully without any single point of failure in the system.

In a similar experiment to that reported in Figure 1, we found that 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.

2 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 microarchitecture, 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", "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*.

Each cluster can accomplish one or more services and, if faulty, can be disabled without affecting the rest of the system. Additionally, a cluster providing multiple services can be partially disabled and only used for instructions that need its functioning services. Such a design greatly simplifies fault isolation, as each cluster is fully independent. In Viper, a program is always able to successfully execute as long as the working hardware clusters can, in aggregate, perform all the services required by its instructions. Lifetime system reliability can be arbitrarily improved by connecting more clusters.

Once an instruction is decoded, it is possible to know which remaining services it needs, and the instruction can be directed towards clusters that can provide these services. The set of clusters that contribute to the completion of an instruction form a *virtual pipeline*. Because clusters can be distributed across the chip, it may take many more clock cycles to transfer instruction information through these virtual pipelines than through a traditional hardwired pipeline. To mitigate this potential performance loss, Viper operates on larger collections of instructions called *bundles*, which, like basic blocks, typically end in control flow instructions. Bundles can successfully execute as long as at least one cluster can complete all their required services.

Functional units within a cluster can service a bundle's instructions out of order, and thus the maximum throughput achievable by a single cluster matches that of an out-of-order processor with an execution instruction window equal to the maximum number of instructions in a bundle.

Moreover, Viper must be able to dynamically determine which hardware clusters will participate in any particular virtual pipeline. To avoid reliability-reducing centralized logic, Viper utilizes a collection of distributed, independent and robust structures called *Bundle Scheduling Units (BSUs)*. Each BSU has some amount of reliable memory (*e.g.*, ECC protected) that stores information such as the services that its bundle needs to complete, its virtual pipeline configuration, and the status of all operations performed on the bundle up to this point. This information is used to control the execution of a single bundle of instructions as it works its way to completion through its virtual pipeline.

Every BSU also has some amount of local logic that is used to determine which hardware clusters will be used in its bundle's virtual pipeline. Clusters independently signal their ability to complete particular services to each BSU. The BSU then chooses, without consulting any centralized logic, which clusters will form the virtual pipeline that will service its bundle. This process is detailed in Section 3.

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.

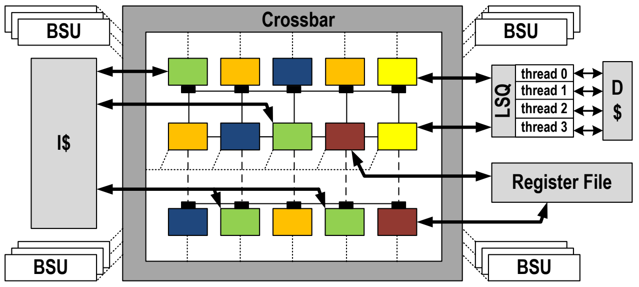


Figure 2: Organization of a Viper system with several redundant clusters that communicate through a mesh and that are connected to the BSUs through a crossbar. Some of the clusters, such as the ones capable of fetching instructions, have special connections to external hardware elements.

Figure 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.

Viper is reliable because the clusters are redundant, meaning that individual clusters can be disabled without jeopardizing the design's ability to execute instructions. As long as all clusters and memory structures are redundant, Viper does not present a single point of failure. In order to maintain availability, Viper utilizes reliable communication infrastructures (the mesh of clusters and the cluster-BSU crossbar) and robust memory elements (BSU, LSQ, RF, etc.). Several solutions have been proposed for generic reliable packet-switched interconnects [10, 38]. As this work focuses on the basics of the Viper architecture, we do not explore these mechanisms further.

We assume that all of Viper's memory structures are protected through ECC. Faults in the - rather small - control logic of these memory elements can be handled either through replication or by disabling faulty entries. Since memory elements in the RF and the LSQ are intrinsically redundant, disabling some of them only affects performance. Similarly, BSUs are also redundant and generic, so individual BSU entries can be deactivated without hindering the design's functionality.

In the following two sections we will illustrate how Viper executes a program using a running example. We first explain the steps necessary to execute a bundle of instructions and then detail how Viper handles special events such as branch mispredictions and exceptions.

3 Regular Execution in Viper

For the sake of simplicity, the Viper design used in our example provides only six services: "fetch", "decode", "rename", "execute", "commit", and "write-back and memory operations". *Even though the services used in this example might resemble stages in a classic pipeline, it is important to stress that our architecture does not impose any constraint on how to partition services. This partition is an arbitrary design choice and should be driven by considering: 1) functionalities exposed by the ISA; 2) tasks accomplished by the underlying hardware; 3) degree of reconfigurability needed by the system.* In Figure 3 we show the Viper design used in this example: it contains four redundant copies of the six different cluster types, each providing one of the services.

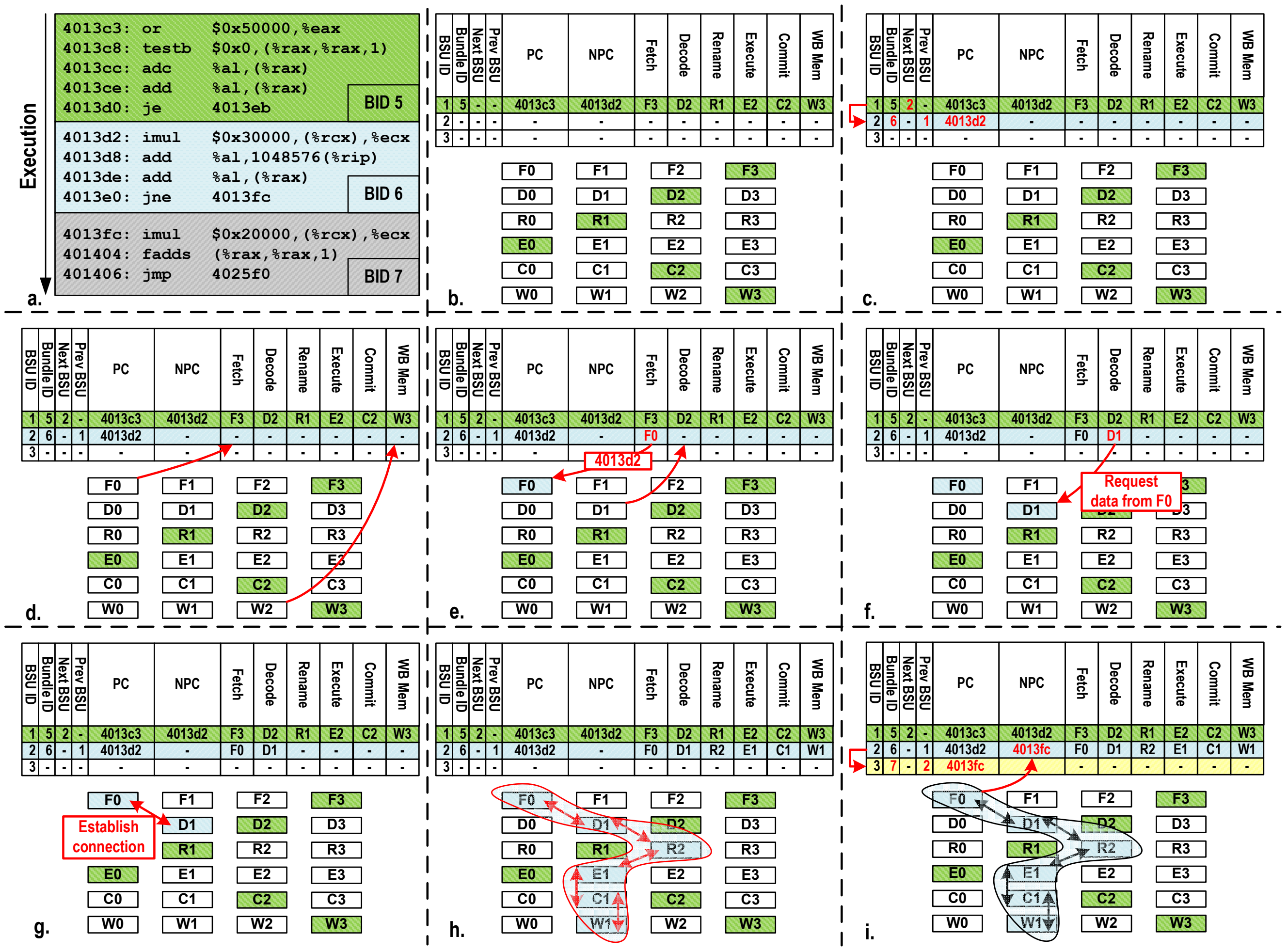
The program stream is dynamically partitioned into bundles of instructions, which typically have basic block granularity. In Viper, each in-flight instruction bundle is associated with a live BSU entry. Figure 3.a shows the example program's three basic blocks. As bundles are created in order, they are assigned a sequential Bundle ID (BID). Each BID is paired with the thread ID of its process to form a unique bundle identifier throughout the entire machine.

In this section we illustrate how Viper can maintain correct program flow for these three bundles (with BIDs 5, 6 and 7) and detail the generation of the virtual pipeline for the second instruction bundle - which starts and terminates with the instructions at addresses `0x4013d2` and `0x4013e0`, respectively. The color coding of the instruction bundles in Figure 3.a matches the hardware resources assigned to their execution and is maintained throughout all steps shown in Figure 3. New events and BSU updates are marked in red.

3.1 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 5's BSU assigns an available BSU entry to the following bundle, as shown in Figure 3.c. The mechanism for choosing the next BSU is described in Section 3.4.2.

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. Figure 3.c shows that BSU 2 is assigned to



```
4013c3: or     $0x50000,%eax
4013c8: testb  $0x0,(%rax,%rax,1)
4013cc: adc    %al,(%rax)
4013ce: add    %al,(%rax)
4013d0: je     4013eb
4013d2: imul   $0x30000,(%rcx),%ecx
4013d8: add    %al,1048576(%rip)
4013de: add    %al,(%rax)
4013e0: jne    4013fc
4013fc: imul   $0x20000,(%rcx),%ecx
401404: fadds  (%rax,%rax,1)
401406: jmp    4025f0
```

Figure 3: Virtual pipeline creation process for the second bundle in frame **a**. **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 BSU's NPC field, which allows the next bundle to begin.

keep track of a new bundle (with BID 6), and therefore the list of clusters assigned to its virtual pipeline is reset.

A similar process is also used to bootstrap Viper: when starting the system, a bundle with BID 0 is assigned to a BSU and its initial address is set to the reset address.

3.2 Virtual Pipeline Generation

A 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.

3.2.1 Service Proposal

Several distributed mechanisms can be used to allow service proposals to reach the BSUs - solutions based on exchange of credits, token broadcasts or service queues could all fit this purpose. For the sake of simplicity, and without losing generality, we adopt a technique based on service queues in this example. In such an implementation, a live BSU enrolls all needed services in queues accessible through a crossbar by both the hardware clusters and BSUs. BSUs requesting clusters are arranged in ascending order based on their BIDs, and service proposals from clusters are first forwarded to the oldest BID.

In our example, as the bundle with BID 6 has just been created, all six required services need to be assigned. Available clusters independently propose to service the BSUs that are enrolled in the service queue. Each cluster maintains a list of the virtual pipelines that have accepted its proposals, though a cluster can simultaneously be part of only a limited number of virtual pipelines.

In our example, we assume that a cluster cannot propose its service to multiple BSUs: clusters F3, D2, R1, E0, C2, and W3 are already assigned to the previous bundle and therefore refrain from proposing their services to BSU 2. Nevertheless, any other available cluster (shown with a white background) can propose its services to the service queues, which redirect such proposals to needy BSUs. For instance, in Figure 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.

After submitting a service proposal, a hardware cluster 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 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.

3.2.2 Service Assignment

BSUs notify clusters accepted into the new virtual pipeline with an award message. In order to correctly build a new virtual pipeline, BSUs award clusters in the exact sequence as their services will be performed on the bundle. For instance, proposals for the "decode" service will not be accepted until the "fetch" service has been assigned to a cluster. In our example, the BSU cannot accept W2's proposal (shown in Figure 3.d) and cluster W2 therefore automatically returns to the "idle" state.

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 Figure 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.

The other services are assigned to clusters in a similar fashion. Figure 3.e shows a service proposal sent by D1. Some of the services - such as "fetch" - are common to all bundles, while others can only be assigned once instructions in a bundle have been fetched and decoded. As the list of services is populated, functional hardware clusters are chosen in order to construct a complete virtual pipeline.

3.2.3 Configuring the Sea of Clusters

Clusters are dynamically selected to service bundles, and communication channels must be established between them to transfer information through the virtual pipeline. To perform this task, each cluster needs to know which clusters precede it in the virtual pipeline. In Figure 3.f we show the BSU awarding its "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 Figure 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 Figure 3.h. Viper can concurrently configure several independent active virtual pipelines, since the BSUs and execution clusters operate autonomousl. Multiple virtual pipelines can work on a single program (as shown in our example), or can simultaneously execute multiple threads.

As faults accumulate in a device, a bundle might require a set of services that none of the execution clusters can provide alone. For instance, bundles 6 and 7 in our example can only execute on clusters that can service both the `add` and `mul` instructions. However, Viper can overcome this problem as long as at least one cluster can execute each one of the needed services. This case is addressed by canceling the execution of the unserviceable bundle and splitting it into multiple bundles, each consisting of a single instruction. While this technique reduces performance, as virtual pipeline creation overhead is not amortized across multiple instructions, it maximizes system availability by minimizing the set of services necessary to complete each bundle.

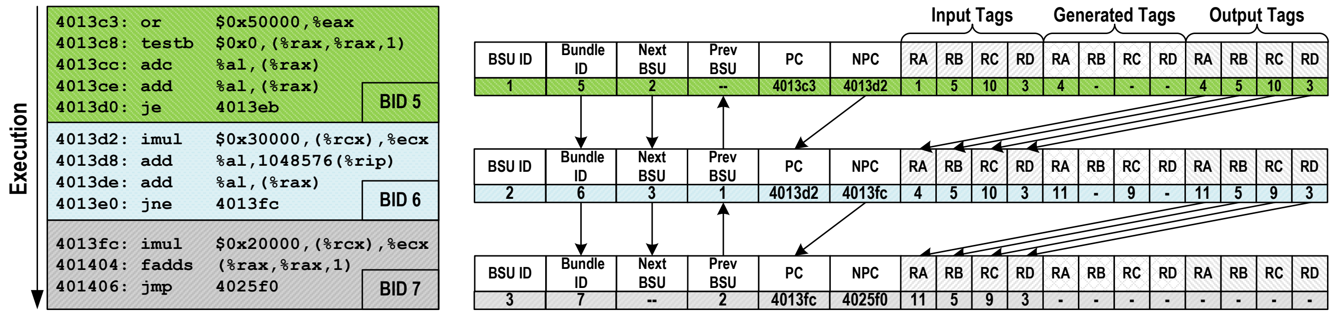
3.3 Operand Tags Generation

Viper does not enforce execution ordering on the different bundles, as long as 1) bundles belonging to the same thread commit sequentially and 2) cluster allocation avoids resource starvation. Thus, there is the opportunity for clusters to concurrently work on multiple bundles from the same program. For instance, in our example we show Viper concurrently executing BIDs 5 and 6.

Viper can improve performance by exploiting a program's ILP and capitalizing on the available hardware resources. However, this also creates inter-cluster data dependencies, as operands produced by clusters in one virtual pipeline might be needed by others. Viper utilizes operand tags to distribute values within the sea of clusters.

Adopting a centralized rename unit is not feasible, as this would create a single point of failure in the system. Thus, we developed a BSU-based mechanism for generating and distributing tags to values produced by bundles. Because each bundle consists of an ordered sequence of instructions, only values live at a bundle's exit point can be used by following instructions. Thus, only live registers will have an associated tag: if multiple instructions in one bundle write to the same architectural register, only the last value produced is associated with a tag.

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.



```
4013c3: or      $0x50000,%eax
4013c8: testb   $0x0,(%rax,%rax,1)
4013cc: adc     %al,(%rax)
4013ce: add     %al,(%rax)
4013d0: je      4013eb
4013d2: imul    $0x30000,(%rcx),%ecx
4013d8: add     %al,1048576(%rip)
4013de: add     %al,(%rax)
4013e0: jne     4013fc
4013fc: imul    $0x20000,(%rcx),%ecx
401404: fadds   (%rax,%rax,1)
401406: jmp     4025f0
```

						Input Tags				Generated Tags				Output Tags			
BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	RA	RB	RC	RD	RA	RB	RC	RD	RA	RB	RC	RD
1	5	2	--	4013c3	4013d2	1	5	10	3	4	-	-	-	4	5	10	3

BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	RA	RB	RC	RD	RA	RB	RC	RD	RA	RB	RC	RD
2	6	3	1	4013d2	4013fc	4	5	10	3	11	-	9	-	11	5	9	3

BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	RA	RB	RC	RD	RA	RB	RC	RD	RA	RB	RC	RD
3	7	--	2	4013fc	4025f0	11	5	9	3	-	-	-	-	-	-	-	-

Figure 4: Distributed rename table for an ISA containing four architectural registers - RA, RB, RC, RD. Tags assigned to the registers in previous bundles are used by the following to retrieve operands and solve data dependencies.

In Figure 4, we illustrate how the tag generation and distribution process works for the three bundles used in our example for an ISA containing four architectural registers: RA, RB, RC, and RD. The first tags, "input tags," are used to allow a bundle to retrieve its input operands: in our example in Figure 4, the cluster fetching register values for the instructions in the bundle with BID 5 will use tag "1" to retrieve the value of register RA from the physical register file, tag "5" to retrieve RB, and so forth.

The second set of tags, called "generated tags," is used only if instructions in a bundle write to architected registers. In our example, two instructions update RA: `adc %al,(%rax)` first and `add %al,(%rax)` later. Because both of these instructions update the same register, only the operand computed by the last operation - `add %al,(%rax)` - needs to generate a new tag, 4. Tag generation could either be accomplished by the BSU or it can be serviced by the clusters. Tag generation for two sequential bundles must be serialized to guarantee program semantics. Solutions to overcome operand aliasing are: 1) maintaining the set of free tags in a memory array protected by ECC; 2) piggybacking a list of available tags in the live BSUs; 3) generating a large number of tags with limited lifetime.

Finally, the "output tags" are associated with the operands that can be used by subsequent bundles. Output tags are produced by overwriting the input tags with any newly generated ones. Output tags of one bundle are provided as input tags of the next, as shown in Figure 4.

The set of input tags in a BSU is also annotated with the hardware cluster that produces the operand, thus avoiding the need to broadcast operand requests to the entire sea of clusters. A cluster working on a bundle will request its input operands from both the register file and from the clusters that contributed to previous bundles. Finally, operands can be requested in advance, as tags are available from the preceding bundle.

3.4 Bundle Termination

A bundle can terminate only if two conditions are met: 1) all clusters assigned to its virtual pipeline finish servicing its instructions; 2) all preceding bundles belonging to the same thread have already terminated. If both these conditions are met, a bundle's instructions are then checked for exceptions. If no exceptions are detected, the bundle is terminated atomically, its instructions update the architectural register file and its "store" operations are committed to memory. In the example in Figure 3.h we show two bundles in-flight (with BID 5 and 6). As instructions need to commit in program order, bundle 6 is not allowed to terminate before its predecessor, bundle 5. Bundle 5, on the other hand, is the oldest bundle in flight ("Prev BSU" is empty) and can terminate as soon as all the clusters in its virtual pipeline complete their services.

It is worth noting that Viper does not need a reorder buffer, as program order is enforced by terminating bundles in sequential order.

3.4.1 Memory Operations

In order to enforce ordered accesses to memory and detect address conflicts, our design includes one load and store queue for each simultaneously supported thread. Each entry in the load queue keeps track of the cluster that generated the memory requests, so as to deliver the data retrieved from memory to the correct destination. Each entry in the store buffer also maintains information about the bundle that originated the store instruction, as memory updates are committed or canceled at the bundle granularity. Before terminating, a bundle with pending store instructions signals to the store buffer associated with its thread that its memory operations can be committed. Such a signal will cause all store instructions in the bundle to update the memory state in program order.

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. This replay mechanism is also used to handle exceptional events, such as page faults, and is detailed in Section 4.

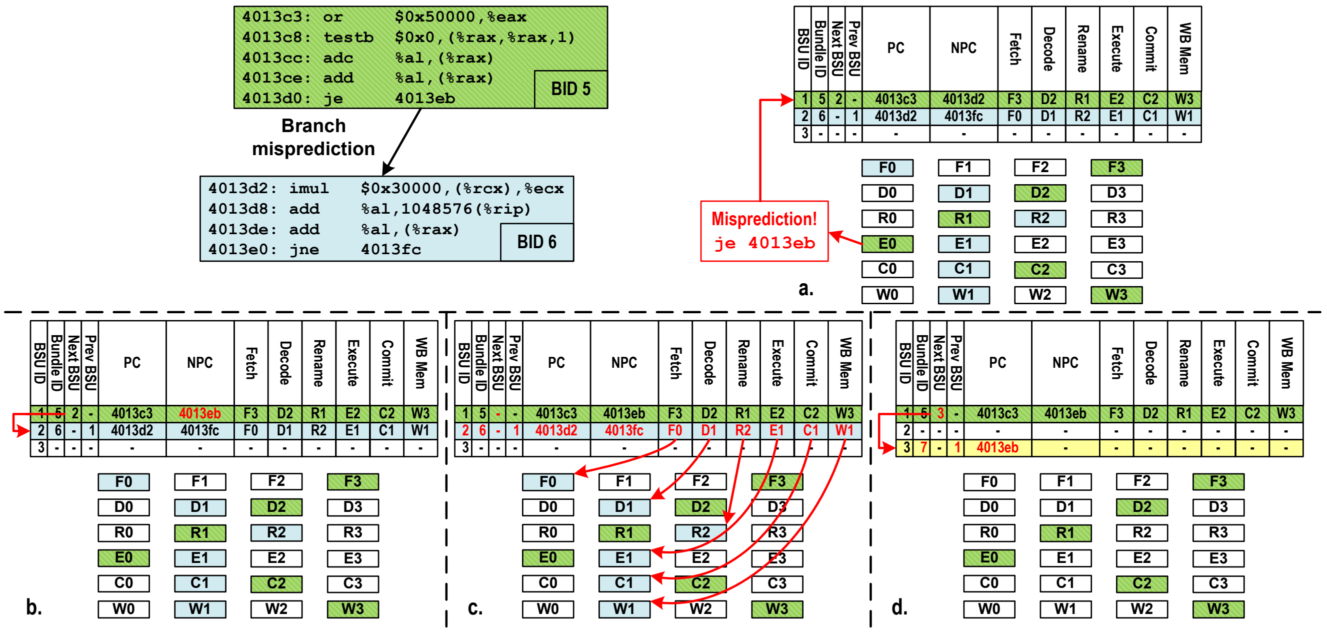


Figure 5: The cluster that detects a branch misprediction updates the program counter of its virtual pipeline (**a**), which consequently clears the state of the following bundles (**b**) and resets the virtual pipelines for the misspeculated bundle (**c**). Finally, program flow is steered towards the correct execution path (**d**).

3.4.2 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 Figure 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.

BSU assignment is performed with the same mechanism used for cluster service negotiation. An active BSU needing to initiate a new bundle requests a new service, "initiate a new bundle", to the service negotiation system. Idle BSUs will then propose to accomplish such a task, as previously detailed in Section 3.2.1.

4 Handling Exceptional Events

Due to the fact that hardware clusters are fully decoupled, our architecture cannot rely on classic techniques - such as broadcasts of clear signals - to flush stale instructions from the system and correct erroneous control flows. In this section, we detail how Viper can resolve such events through its BSUs.

4.1 Mispredicted Branches

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.

Similarly, Viper needs to cancel the execution of bundles generated by misspeculated program paths. We use the example shown in Figure 5 to illustrate how our architecture can tackle such events. We assume that bundle 6 is mistakenly predicted to follow bundle 5, and that all services required by both virtual pipelines are already assigned to clusters in the execution engine - Figure 5.a. Once a cluster in a virtual pipeline resolves a branch target, it reports the computed address to its BSU. This case is shown in Figure 5.a, where cluster E0 reports to bundle 5 that the correct initial address of the next basic block is `0x4013eb`. If this target address does not match the one stored in the NPC field of the BSU, a bundle misprediction is detected - Figure 5.b. All bundles generated from a mispredicted address - in our example bundle 6 starting from address `0x4013d2` - are canceled. Through the BSUs, the clusters composing virtual pipelines of canceled bundles are notified to stop their work and clear their state - Figure 5.c. Finally, the most recent non-speculative bundle recovers program execution, creating a new instruction bundle starting at the correct basic block address - Figure 5.d.

4.2 Exception and Trap Handling

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 memory accesses.

4.3 Runtime Failures

In this work we assume that multiple permanent faults could hit any hardware component among the clusters, the BSUs and the interconnect. 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 [17, 23, 31]. 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.

Our architecture can recover from an online fault through checkpoint techniques such as ReVive and SafetyNet [26, 33]. In Viper, program state is distributed, as program counters are stored in the BSUs, while architectural register values are placed next to the hardware clusters: these can be synchronized to a unique checkpoint state when bundles commit. In the case of runtime fault detection, all in-flight bundles are canceled and the faulty component is disabled. Both Viper's architectural state and memory system are restored to a previous safe checkpoint, and the checkpointed program counter is used to start a new bundle and restart program execution.

5 Experimental Setup

We simulated a Viper implementation that uses the x86-64 ISA to evaluate Viper's performance and reliability. We compared Viper against two similarly sized CMP designs comprised of either 2-wide in-order or out-of-order cores. We chose the former design as it constitutes the backbone of a number of current many-core systems, such as the Intel KNF and the Tilera TILE64 [3, 28]. We envision the latter as the natural successor to in-order CMP cores, when future technology nodes will allow the integration of a larger number of transistors, but performance returns from thread-level parallelism wanes [7, 29].

We first study the effects of bundle size on Viper's performance and follow this with an evaluation of Viper's area overhead. Next, we compare the performance of Viper against traditional CMPs comprised of in-order and out-of-order cores. These tests measure the single-threaded and multi-program performance of workloads from the MiBench and SPEC CPU2006 benchmark suites [13, 15] when executed on chips that contain no hardware faults. However, because future CMPs are expected to experience many transistor failures over their lifetime, we examine how these systems react to hardware failures.

5.1 Hardware Model

The Viper architecture we evaluated in this work offers only six services: "fetch", "decode", "tag generation", "execute", "commit" and "write to memory". Adding more services would increase the reliability of the design, but we left such analysis to future work. Our modeled Viper design includes five types of clusters. The first four services are each executed by four different kinds of clusters, each capable of performing a single service. The fifth type of cluster can accomplish both the "commit" and the "write to memory" services.

The sea of hardware clusters is organized in a mesh connected through 256-bit wide links. Routes in the interconnect can be warmed up before transmitting the data packets, so we modeled the cluster-to-cluster latency as one extra cycle of delay per hop, with data transmission between clusters fully pipelined [21]. Communication between the BSUs and the clusters requires very little bandwidth, since it is limited to a few control bits. For these connections, we used a crossbar with a latency of 4 cycles [39].

The Viper design we modeled adopts two optimizations to improve efficiency and utilization:

1. Early virtual pipelines generation: service proposal negotiation can be performed ahead of time; for instance, while clusters are working on previous bundles, thus reducing their idle time.
2. Non-blocking instruction migration: instructions can be transferred from a cluster to the next in the virtual pipeline as soon as they complete, instead of waiting for the whole bundle to be serviced.

Viper, the in-order and the OoO cores are modeled to fetch, decode, execute and commit up to two instructions per cycle. They are all clocked at 2 GHz. Each core contains 2 integer pipelines, 2 FP units, 1 load/store unit and 32KB of L1D and L1I. In order to fairly compare Viper's performance against classic processors, each of the "execute" clusters in our design has functional units identical to those in the in-order and OoO cores. Finally, in our performance evaluation, we compare Viper against a baseline OoO processor with a comparable instruction window - 64 ROB entries and 5 RS entries per functional unit. Both the OoO machine and Viper's "execute" clusters can issue up to 5 instructions per cycle to the functional units.

5.2 Simulation Infrastructure

We developed a microarchitectural model of our design in the gem5 simulator [5], relying on full timing simulations in system-call emulation mode. The Viper system modeled is based on the C++ implementation of the OoO core provided by the original gem5 distribution. Building on this model, we organized the system in fully decoupled clusters and augmented it with the required communication infrastructures (inter-cluster mesh and crossbar) and the BSUs. Timing models for all hardware components have been modified to better match the deeper pipelines of typical of modern CISC processors [14]. The number of cycles for each logical stage are listed in Table 1, and sum to a minimum of 12 cycles.

Cluster Names	**Number of clock cycles**	**Viper area (transistors)**	**In-order area (transistors)**	**OOO area (transistors)**
Fetch	3	4M	1.5M	4M
Decode	3	2.5M	2.5M	2.5M
Rename/Tag Generation	3	3M	0	3M
Execute	Variable (min 1)	6.5M	6M	6.5M
Commit and load/store logic	2	4.5M	4.2M	5M
Switches		0.65M	0	0
Communication buffers		1.2M	0	0
Total pipeline		22.35M	14.2M	21M
Number pipelines		4	6	4
BSU		0.5M	0	0
Crossbar		0.12M	0	0
Total area		90.02M	85.2M	84M

Table 1: Area estimations and delay of the pipelines stages for the in-order and OoO designs and for Viper's clusters

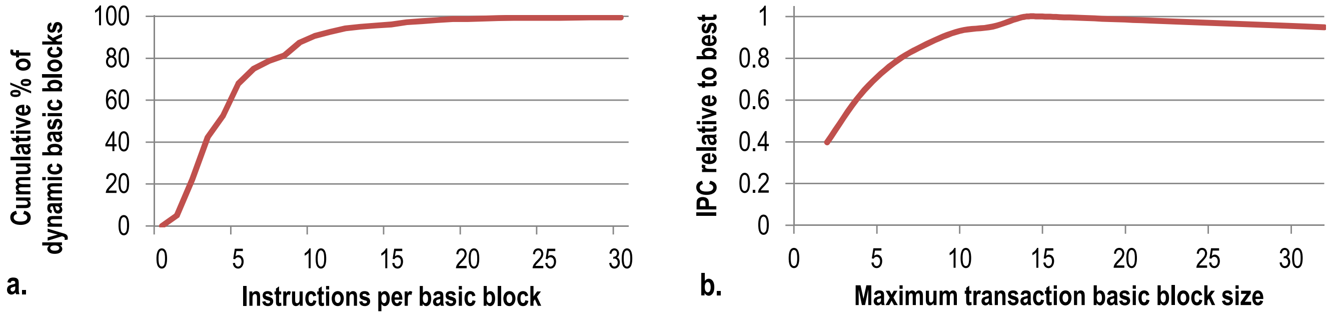


Figure 6: **a**) Cumulative distribution of the basic block size in our benchmarks. **b**) Sensitivity study on the maximum number of instructions allowed in Viper's bundles.

6 Experimental Results

6.1 Design Choices

We first analyzed the benchmarks' performance as a function of the number of available BSUs. Because the BSUs hold the dynamic state of bundles that are running in parallel, their number directly affects the maximum ILP achievable by the execution engine. We found that single-threaded performance reaches a plateau for a system composed of 4 BSUs for every full set of hardware clusters. The minimum number of operational BSUs needed by our proposed microarchitecture is 2. However, we estimated that a Viper system with only 2 BSUs operates 24.4% slower on average than one with 4 BSUs. Since the system analyzed in this section supports up to four concurrent threads, a model with 16 BSUs is used for all further experiments.

Another parameter to select is the maximum number of instructions allowed in a single bundle. On one hand, bundles with a large number of instructions have the potential to depend less on operands produced by clusters in other virtual pipelines. This can provide a significant advantage, as it reduces instruction reliance on long latency inter-cluster operand requests. On the other hand, partitioning program execution in smaller bundles allows more clusters to execute instructions concurrently. This leads to a performance tradeoff, which we analyzed through Pin [18] by gathering statistics on the distribution of basic block sizes in our benchmarks. Our finding are reported in Figure 6.a. More than 95% of the dynamic basic blocks in our applications are smaller than 16 instructions. We performed a sensitivity study on how this parameter affects Viper's performance, and report our results in Figure 6.b. A slight performance slowdown is shown for bundles larger than 16 instructions, mostly due to the higher probability of conflicts between instructions in the load/store queue and to the higher costs of recovering from canceled bundles.

6.2 Area

We measured the performance and reliability of small CMPs with comparable transistor counts, as reported in Table 1. All area estimations have been computed for a technology node of $65nm$. According to our estimations, the computational core of a CMP holds 90 million transistors, which can fit: 4 complete sets of Viper clusters, 6 in-order cores or 4 out-of-order processors. CACTI 5.3 was used to estimate the area of memory structures, such as the BSU, ROB and instruction buffers [37]. We estimated that Viper hardware organization would increase the total area of the CMP's computational core by 7.2% compared to the one composed of OoO processors.

Processor modifications - We compared the area of a single set of Viper hardware clusters against two 64-bit x86 processors: an in-order core (similar to Intel's Atom Silverthorne core [14]) that requires 14.2 million transistors in roughly $13mm^2$, and an out-of-order pipeline (scaled from AMD's Opteron Deerhound microarchitecture [16]) that requires 21 million in roughly $19mm^2$.

Every cluster must include input and output buffers for

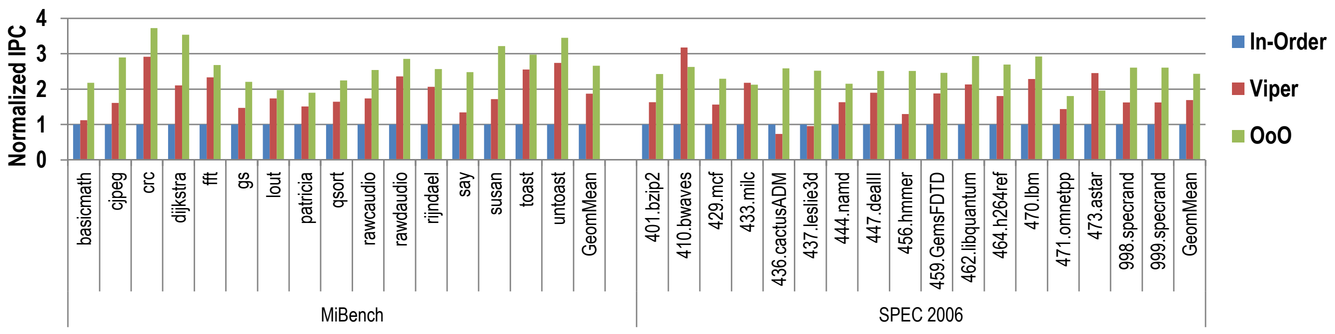


Figure 7: IPC achievable by fault-free configurations of the in-order core, OoO core and Viper. IPC of the in-order core is used as baseline.

moving bundle data during execution, which adds an area footprint of $0.31mm^2$. Additionally, as the data for inter-cluster communication is already buffered, it needs only simple MUX-based switches, which can be built using about 150,000 transistors. One of these switches is required for each hardware cluster. In our model each cluster is paired with a NoC router enhanced with the reliability features in [10] and its area is estimated accordingly (40% increase in size).

BSU - We modeled a BSU with 16 entries in these experiments. The storage required for each entry is reported in Table 2, and each entry is enhanced with 58 bits of ECC, for a total storage of 1,536 bytes. The area footprint for this structure has been estimated to be $0.52mm^2$.

Crossbar - Our design requires a crossbar connecting 20 clusters with 16 BSU entries. Detailed area estimation for a comparably sized crossbar (18x18) reports a footprint of $0.12mm^2$ [27].

6.3 Fault Free Performance

Due to our detailed simulations, we limited our performance and fault degradation experiments to CMPs comprising: 4 complete sets of Viper clusters, 6 in-order cores or 4 out-of-order processors. We report the IPC achievable by Viper when no faults are present in the design for single-threaded and multi-programmed workloads. Figure 7 reports performance figures for single-threaded benchmarks: Viper outperforms the in-order core in most workloads. The performance increase for the vast majority of workloads is above 50% and is higher for benchmarks such as *410.bwaves* and *470.lbm* that can expose high ILP and MLP. An interesting exception is *436.cactusADM*, which does not perform well in this model of Viper. This benchmark is composed of large basic blocks [11], and their parallel execution in Viper increases the probability of replays due to conflicts in the load/store queue.

A healthy Viper design loses an average of only 24% performance compared to the OoO core, as reported in Figure 7. This is primarily due to the overhead of generating virtual pipelines. Still, Viper's ability to execute on multiple clusters allows IPC improvements for *473.astar*, *410.bwaves* and *433.milc*. We believe that are many possible optimizations that could recover most of the other performance loss.

Figure 8 plots Viper's throughput on multi-programmed benchmarks. In this case we compare the aggregate IPC achievable by 4 sets of Viper clusters against 6 in-order cores and 4 OoO pipelines. In this evaluation we run the same copy of the program for the maximum number of times allowed by the each of the three designs. For these workloads Viper's performance disadvantage compared to OoO is more significant, as the higher hardware utilization does not allow programs to execute bundles out of order.

Content	Size [Bits]
Bundle ID	16
Basic block program count.	64
Next basic block program count.	64
Branch pred. data	4
Previous bundle	4
Next bundle	4
Virtual pipeline	7*6
Input Tags	16*24
Output Tags	16*24

Table 2: BSU storage: Since there are only 16 BSU in our implementation, only 4 bits are needed to index other BSUs. Six services are present in our design, each requiring 2 control bits (assigned, proposal pending) and 5 bits to index the assigned hardware cluster. Finally, 16-bit tags are maintained for the 24 registers of the x86 architecture (9 "general purpose", 6 "segment pointers", 8 "MMX" and 1 for execution flags).

Performance figures for Viper are - on average - lower than for regular OoO CMP designs. However, state-of-the-art reliable microarchitectures are only applicable to in-order cores: compared to such previous solutions, Viper provides a significant performance advantage [12, 31].

6.4 Faulty Behavior

To better compare the reliability and performance of Viper against previous works, we measured the expected throughput of four different designs for a chip of 2 billion transistors. In Figure 9 we compare Viper against the following designs: in-order CMP, Bulletproof, StageNet and Viper. As the graphs show, performance of the unprotected solution, though initially higher, quickly degrades as the number of faults increases. Performance degradation for both Bulletproof and StageNet is more graceful, but their reliance on centralized control logic affects performance as the number of hardware errors grows. On the other hand, thanks to its distributed control logic, Viper is capable of maintaining higher performance even on silicon substrates tainted by hundreds of permanent faults.

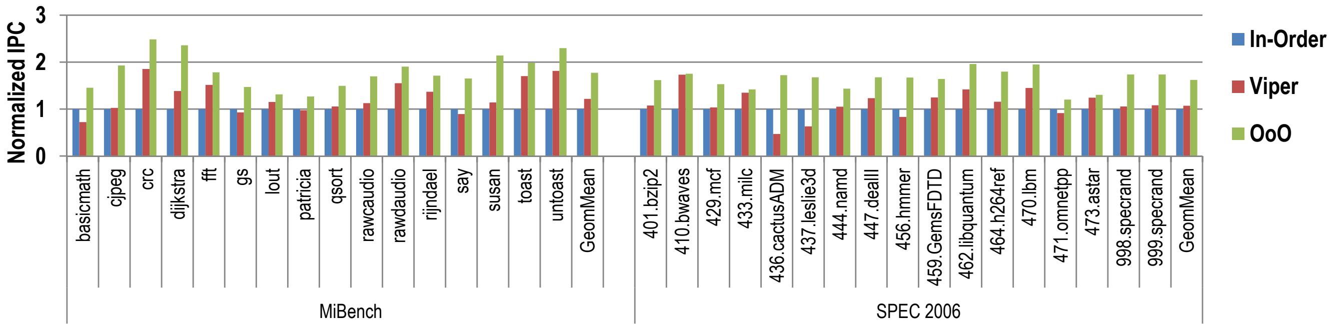


Figure 8: IPC achieved in our experiments by fault-free configurations of the three different CMP systems using multi-programmed benchmarks. These experiments are performed on similarly sized designs, containing respectively: 6 in-order cores, 4 OoO cores and 4 copies of all types of clusters in Viper. IPC of the in-order core is used as baseline.

7 Related Work

Mission critical and high-availability computers currently rely on coarse-grained redundancy to improve reliability. Systems such as the HP NonStop and the IBM zSeries depend on processors with many RAS (reliability, availability, and serviceability) features [19], and often run them in dual or triple modular redundant configurations [2, 4]. They trade high area, power, and performance overheads for their required reliability. Moreover, due to the coarse granularity of redundancy, such systems would still be unable to perform well when every processor is subjected to a high fault rate.

More recently, several research projects proposed low-cost reliable processor designs. BulletProof targets VLIW designs, but its scalability to dynamic out-of-order processors is not clear [31]. Other works on reliable CPUs only focus on components with natural redundancy, such as the reorder buffer, branch history table, caches and other arrays of regular structures [8, 30]. Architectural core salvaging maintains ISA compliance at the die level, but still presents single points of failure due to its reliance on classic centralized control logic [25].

StageNet is a more radical approach to processor reliability that proposes to use a reconfigurable fabric connecting multiple identical pipelines in a multi-core machine [12]. Broken pipeline stages are disabled, and the fabric is reconfigured offline to force programs to utilize the functioning hardware. Unfortunately, this solution does not scale to complex designs, such as OoO cores. Additionally, control logic constitutes a single point of failure, the number of threads supported by its fabric is limited by the number of instruction fetch stages available, and redundant component granularity cannot be varied beyond pipeline stages.

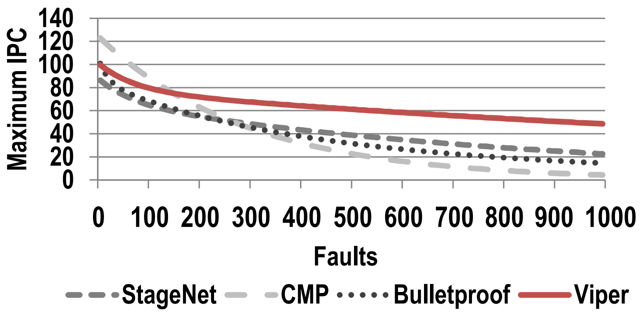


Figure 9: Comparison between performance of unprotected in-order cores (CMP), Bulletproof pipelines, StageNets and our solution, Viper.

Viper's execution model might seem similar to those adopted by data-flow machines [9, 32, 35, 36]. Programs running in any of these designs require binaries annotated with architectural-specific information and thus they cannot execute legacy binaries. Among these designs, Wavescalar [35], is the only one that could dynamically avoid assigning instructions to faulty processing elements. However, as instruction scheduling is managed by a centralized hash table, this design also has a single point of failure.

8 Conclusions and Future Directions

In this work, we presented Viper, a novel microarchitecture that uses a reconfigurable execution engine built from redundant components steered by fully distributed control logic. Instructions in this design are viewed as clients that require a number of services. These customers are served by the currently available hardware components, and a program can successfully finish as long as its required services can be executed by a dynamic collection of the available components. Viper avoids a single point of failure by construction, resulting in a complete distributed design while maintaining high performance. Viper can tolerate numerous hardware faults, and its performance degrades gracefully as fault count increases.

There are a number of future directions for this research. Viper's performance could be improved by developing more efficient and faster techniques for building virtual pipelines and handling exceptions. We would also like to perform a more thorough study of the reliability of our architecture and investigate cluster reconfiguration granularity and its tradeoff between performance and fault-tolerance. Finally, we would like to better explore the performance advantages available for workloads that can capitalize on Viper's flexible execution engine.

Acknowledgments

We wish to thank Prof. Todd Austin, Debapriya Chatterjee, Anthony Gutierrez and Biruk Mammo for their valuable feedback in the development of this work. In addition, we acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *MICRO*, 1999.

[2] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *DSN*, 2004.

[3] S. Bell et al. TILE64 - Processor: A 64-core SoC with Mesh Interconnect. In *ISSCC*, 2008.

[4] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *DSN*, 2005.

[5] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[6] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[7] S. Borkar. Thousand Core Chips: A Technology Perspective. In *DAC*, 2007.

[8] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *DSN*, 2004.

[9] D. Burger et al. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[10] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester. Vicis: A Reliable Network for Unreliable Silicon. In *DAC*, 2009.

[11] K. Ganesan, J. Jo, and L. K. John. Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads. In *ISPASS*, 2010.

[12] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The StageNet Fabric for Constructing Resilient Multicore Systems. In *MICRO*, 2008.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Int'l Workshop on Workload Characterization*, 2001.

[14] T. R. Halfhill. Intel's Tiny Atom. *Microprocessor Report*, Apr 2008.

[15] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[16] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, 2003.

[17] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *ASPLOS*, 2008.

[18] C.-K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[19] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, 2005.

[20] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *MICRO*, 2007.

[21] G. Michelogiannakis, D. Pnevmatikatos, and M. Katevenis. Approaching Ideal NoC Latency with Pre-Configured Routes. In *NOCS*, 2007.

[22] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *EuroSys*, 2011.

[23] A. Pellegrini and V. Bertacco. Application-Aware Diagnosis of Runtime Hardware Faults. In *ICCAD*, 2010.

[24] A. Pellegrini and V. Bertacco. Cardio: Adaptive CMPs for Reliability through Dynamic Introspective Operation. In *HLDVT*, 2011.

[25] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance. In *ISCA*, 2009.

[26] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Mem Multiprocessors. In *ISCA*, 2002.

[27] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. D. Micheli, and L. Benini. Bringing NoCs to 65 nm. *IEEE Micro*, 12(5):75–85, 2007.

[28] L. Seiler et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *ACM SIGGRAPH*, 2008.

[29] M. Shah et al. SPARC T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro*, 32(2):8–19, 2012.

[30] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy for Defect Tolerance. In *ICCD*, 2003.

[31] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *ASPLOS*, 2006.

[32] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA*, 1995.

[33] D. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *ISCA*, 2002.

[34] A. W. Strong, E. Y. Wu, R.-P. Vollertsen, J. Sune, G. L. Rosa, and T. D. Sullivan. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley-IEEE Press, 2006.

[35] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *MICRO*, 2003.

[36] M. B. Taylor et al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[37] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical report, Hewlett-Packard Laboratories, 2008.

[38] A. Vitkovskiy, V. Soteriou, and C. Nicopoulos. A Fine-Grained Link-Level Fault-Tolerant Mechanism for Networks-on-Chip. In *ICCD*, 2010.

[39] M. Woh, S. Satpathy, R. G. Dreslinski, D. Kershaw, D. Sylvester, D. Blaauw, and T. Mudge. Low Power Interconnects for SIMD Computers. In *DATE*, 2011.