Cobra: a Comprehensive Bundle-based Reliable Architecture

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

***Abstract*—The declining robustness of transistors and their ever-denser integration threatens the dependability of future microprocessors. Classic multicores offer a simple solution to overcome hardware defects: faulty processors can be disabled without affecting the rest of the system. However, this approach becomes quickly an impractical solution at high fault rates.**

Recently, distributed computer architectures have been proposed to mitigate the effects of faulty transistors by utilizing fine-grained hardware reconfiguration, managed by fully decoupled control logic. Unfortunately, such solutions trade flexibility for execution locality, and thus they do not scale to large systems. To overcome this issue we propose Cobra, a distributed, scalable, highly parallel reliable architecture. Cobra is a service-based architecture where groups of dynamic instructions flow independently through the system, making use of the available hardware resources. Cobra organizes the system's units dynamically using a novel resource assignment that preserves locality and limits communication overhead. Our experiments show that Cobra is extremely dependable, and outperforms classic multicores when subjected to 5 or more defects per 100 million transistors. We also show that Cobra operates 80% faster than previous state-of-the-art solutions on multi-programmed SPEC CPU2006 workloads and it improves cache hit rate by up to 62%. Our runtime fault detection techniques have a performance impact of only 3%.

I. Introduction

Current digital systems comprise billions of transistors in a single chip, and such dense integration is expected to grow even further at future fabrication technology nodes. Experts agree that shrinking device sizes will cause severe degradation in overall system reliability by increasing the susceptibility to both transient and permanent faults [5]. The consequences of this trend are twofold: 1) lower production yields due to higher rates of manufacturing defects and 2) higher incidence of runtime failures in deployed systems – for instance due to electromigration, gate oxide breakdown, negative-bias temperature instability (NBTI) and hot carrier injection [25]. Neglecting runtime hardware faults can lead to serious consequences, such as service disruption and output corruption [19].

Faulty cores in modern CMPs can be individually disabled without compromising the availability of other components [23]. Unfortunately, since a single permanent defect is sufficient to disable an entire core, this solution is only viable for systems subjected to a limited number of faults. As emerging technologies promise to deliver massive integration of highly unreliable nanodevices [26], a scenario where digital systems are affected by a large number of failures is likely. Therefore, future computers will require novel architectures that can perform even when faced with high fault rates. To this end, researchers recently proposed to break apart the hardware units of classic hard-wired pipelines, dissolving them into a sea of redundant hardware components.

These architectures are composed of smaller hardware units organized in a **distributed fashion** to avoid single points of failure – such as centralized control logic – by construction [11, 18]. Distributed architectures can achieve high reliability without sacrificing performance. Upon a fault detection, such designs can modify the hardware configuration by dropping the faulty hardware component and including a new one. This dynamic approach enables distributed-control architectures to achieve high-throughput, even on silicon substrates tainted by hundreds of permanent faults. Hence, these designs enable seamless **fault isolation and dynamic reconfiguration**.

Recent studies evaluated the performance achievable by distributed architectures, and found that these designs can tolerate orders of magnitude more hardware failures than previous solutions [11]. Unfortunately, in achieving good reliability, current distributed architectures compromise execution locality, and thus suffer major limitations, including: communication bottlenecks, dispersed resource allocation, memory contention, and inefficient testing. These shortcomings jeopardize the scalability, performance, and testability of such reliable design. In this work we present Cobra, a novel distributed architecture that overcomes the issues above, enhancing and extending prior distributed solutions. Cobra provides a comprehensive, scalable and reliable solution for highly parallel chips.

A. Contributions

Cobra is a new distributed-control architecture that promotes both scalability and reliability as top-priority design concerns, developing a novel system design and a new mechanism to dynamically assign hardware resources to running applications. Furthermore, Cobra supports a variety of fault detection techniques, ranging from redundant instruction execution to periodic online testing. Software requiring maximum reliability can either make use of fully redundant execution or of periodic hardware tests. Meanwhile, applications that strive to maintain fast fault detection latency at a lower performance price can opt for protecting only the most vulnerable portions of a program. Finally, software that does not require any correctness guarantee can disable all online reliability mechanisms for a performance benefit: in Cobra each application can employ any reliability feature independently from other workloads. We make the following contributions:

1. A scalable highly parallel design that relies on the service-oriented execution paradigm. This work overcomes the issues tainting previous fully distributed architectures, making this powerful execution paradigm viable in practice. Compared to previous state-of-the-art solutions, Cobra has better performance (80%), improves memory utilization (up to 62% and 3% higher cache hit rates for local instruction and data caches, respectively) and reduces the overhead of the reconfiguration mechanism by roughly 50%.

2. A full-chip dependable system that can sustain a large number of defects and gracefully degrades performance as faults accumulate in its hardware. We believe that this is the first work to analyze in detail the reliability characteristics of a fully distributed computer architecture. In our experiments we measure that, for systems affected by more than 1 permanent fault in 20 million transistors, Cobra always outperforms a

classic CMP of comparable size.

3. A system that allows designers to tradeoff performance for robustness, and users to adapt runtime protection to application demands. Cobra can adapt to different reliability needs, enabling designers to tune a system to a targeted fault rate and allowing applications to trade performance overhead (as low as 3%) for fault detection latency.

II. Distributed Execution Model

Classic pipelined processors partition an instruction's execution into multiple stages and assign specific hard-wired tasks to each. In such designs, hardware modules are tightly interconnected for performance reasons. In contrast, distributed-control architectures organize the hardware in a network of isolated, loosely coupled components. Each such component can accomplish one or more *services* for the instructions running on the system (examples of such services can be: "fetch", "decode", "execute addition", *etc.*).

To successfully execute an instruction, components are dynamically arranged in a *hardware configuration* capable of providing all the services required by the instruction. Hardware configurations can be established at compile time [15], when a workload is launched [11], or while it is executing by dedicated *scheduling units* [18]. Although this latter case requires hardware components and scheduling units to *negotiate service assignments* on the fly, it appears to be the most versatile, as it can dynamically mesh workloads' needs with the available resources. Several recent solutions share most or some of the traits defining distributed computer architectures. Among these designs, Wavescalar [27], for instance, is capable of dynamically allocating instructions to available processing elements; StageNet [11] and Viper [18] use a finer granularity and allow the dynamic allocation of all functionalities provided by an ISA. These three solutions differ greatly in the granularity of the allocated services, reconfigurability and program execution model. However, they all operate on clusters of instructions (instead of individual ones) to improve performance or to lower energy consumption. Expensive operations, such as scheduling and hardware reconfigurations, can then be shared among the entire cluster. These groups of instructions are called *waves* in Wavescalar, *macro operations* in StageNet and *bundles* in Viper. Here we adopt this latter term since, like Viper, Cobra partitions a dynamic stream of instructions into sequences terminating with a control operation (*e.g.*, jump).

In order to create a robust system, the resource allocation is not managed by a centralized unit, but it is instead achieved by allowing hardware units to independently advertise their availability to the instruction bundles awaiting execution. Service advertisement and negotiation can be handled through several distributed mechanisms, as long as resources are assigned sequentially to avoid deadlocks – examples of possible solutions are: exchange of credits, token broadcasts, and service queues.

Each instruction bundle running on this system is uniquely identified through a sequential ID, generated by increasing the ID of the bundle immediately preceding it. Each bundle in flight is associated to a scheduling unit, which is responsible for managing and tracking all hardware resources operating on its instructions. Scheduling units do not store operations or values, but only the information relative to bundle execution progress: allocated hardware units, pointers to operand locations, data to manage program flow, and instruction order. Scheduling units are connected in a linked list to maintain program order. Once the address of the next instruction bundle is computed, a new scheduling unit is allocated to coordinate

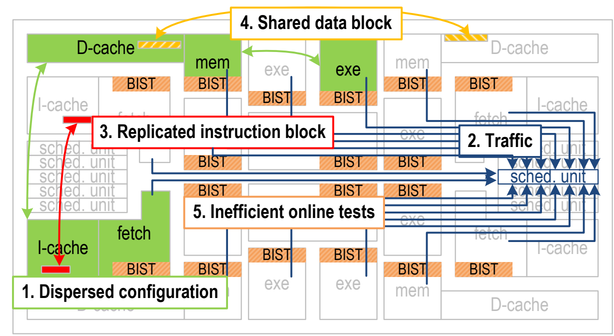

Fig. 1. Limitations of traditional distributed control architectures: 1) dispersed resource assignments lead to high communication latencies between units (in green) leading to *dispersed hardware configurations*; 2) *hardware configuration setups* entail a large communication overhead (in blue); 3) *scattered workload executions* penalize the performance of stateful resources, such as caches (red); 4) *random accesses to memories* affect memory operations' performance and correctness (in yellow); 5) *inefficient hardware tests* affect system's performance (in orange).

its execution. Even though bundles can execute out-of-order, bundles commit their results sequentially, following the order enforced by their IDs [18].

A. Limitations

In a distributed control architecture, instructions can be scheduled to utilize any available hardware unit that suits them. On one hand, this execution paradigm maximizes system availability and increases its resiliency to failures. On the other hand, distributed architectures present several drawbacks that hinder their scalability, performance, and testability. Figure 1 illustrates these issues, summarized below:

1. Physical dispersion of its hardware configurations. Hardware resource assignment based exclusively on instruction demands does not account for the physical distance between the selected components, possibly leading to longer execution runtimes, due to increased communication latencies.

2. Large communication overhead due to the frequent dynamic reconfiguration of the hardware resources assigned to execute an application.

3. Scattered execution of a workload. Since resources are allocated dynamically, consecutive code segments may execute on different sets of units, erasing the benefits of data and instruction locality.

4. Random accesses to memory structures that can affect both the correctness and the performance of load and store memory operations.

5. System's inefficiency in self-testing its components and checkpointing and restoring software state upon a failure. Indeed, processes executing in such architectures could potentially execute on any component, and thus a single hardware failure could taint all software applications running on the machine.

III. Localized Hardware Configurations

The first issue that Cobra strives to overcome is the physical dispersion of hardware configurations. Naïve resource assignment policies do not account for the time spent to transfer instructions and operands across the system. As the number of components in a system grows – and therefore the average distance among them increases – this issue is greatly exacerbated, undermining the scalability of a distributed architecture. Cobra overcomes this issue by leveraging a simple but cost-effective algorithm, which preserves locality without compromising reconfigurability. Note that a straightforward solution would

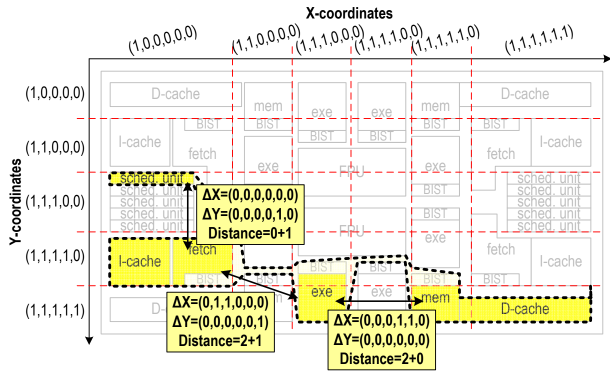

Fig. 2. To setup hardware configurations in Cobra, we use location coordinates. Each coordinate is always encoded as a sequence of bits, whose length corresponds to the maximum number of segments it span. Each segment is identified by the number of "1s" in the leading bits of its coordinate. The distance between two components in a hardware configuration (in yellow) is then determined by summing each direction's hamming distances.

consist of limiting the reach of a service broadcast to a small local region of the chip; however, this approach would limit Cobra's ability to reconfigure around faulty components.

A. *Creating a Localized Hardware Configuration*

When a scheduling unit assembles a hardware configuration, Cobra evaluates the physical location of the units that advertise their availability. In order to do so, components are logically organized in a mesh, so that the distance between any pair of hardware units can be easily computed as their Manhattan distance. Instead of greedily allocating resources as soon as they become available, Cobra's scheduling units store service providers' (hardware units') proposals for a certain number of cycles so as to choose the best among several alternative configurations. Each scheduling unit then generates a hardware configuration based on the services required, taking into account the distances between units in its candidate pool. The target is to minimize communication latency by reducing the overall distance among all the hardware elements forming a configuration. An optimal solution to this problem would require the prohibitive (in hardware) application of Dijkstra's algorithm. Instead, in Cobra we opted for a simpler approach (described below) that fits our purpose while imposing a very small overhead.

Once a new bundle is initiated, its associated scheduling unit accepts proposals from the available hardware units. In order to avoid starvation and deadlocks, the service assignment policy prioritizes the oldest bundle in flight, and then it assigns resources in the same order as they are needed by the instructions in a bundle. Differently from previous solutions, Cobra's scheduling units do not immediately include components in their configurations as their services become available. Instead, our algorithm starts from the location of the scheduling unit and adds each required service provider, one at a time, selecting the available unit that is closest to the previously allocated one. To accomplish this, a scheduling unit stores, for a preset number of cycles, every service proposal that 1) provides forward progress towards the generation of a complete hardware configuration and 2) is physically closer than other proposals received within the preset time window. This technique quickly converges to generate a configuration that minimizes communication latency. This mechanism only requires some additional storage for the location of the next best candidate and a cycle counter to track the search time-window.

Figure 2 provides an example of this process. The location of each hardware unit is represented by a pair $< X, Y >$ of coordinates. Components' physical locations are encoded so that distances can be calculated by computing the Hamming distance between coordinates in each dimension and summing them together. Specifically, we use as many bits as the size in the corresponding dimension (6 for the X coordinate, 5 for the Y, in Figure 2), and encode the position by setting a corresponding number of bits to 1.

IV. Hardware Configuration Lifespan

Dynamic on-demand hardware reconfiguration is the key feature that allows a service-oriented distributed architecture to seamlessly adapt the execution of a workload to the available resources. One method of negotiating services between scheduling units and providers is to allow hardware components to independently advertise their availability to the scheduling units. Components whose services are accepted are then notified back by the scheduling unit. This process dynamically allocates the hardware resources that fit a program's requirements without requiring centralized control structures. However, this mechanism leads to message proliferation, due to the large number of advertisements and notifications exchanged among the hardware components.

To understand the magnitude of this problem, consider for instance a distributed architecture composed of 5 services, 4 providers for each service and 2 active instruction bundles. The maximum number of messages exchanged between the hardware units and the scheduling units is: $5(services) \times 4(providers) \times 2(bundles) = 40$ service proposals and $5(services) \times 2(bundles) = 10$ acceptance notifications. Considering that more of 95% of the bundles in typical applications contain up to 16 instructions [18], the number of messages exchanged to execute a rather short portion of the program is significant. Such overhead grows linearly with the number of service providers and in-flight instruction bundles and it is further exacerbated for workloads requiring redundant execution for reliability purposes.

This overhead occurs because of the short life-span of a hardware configuration: one for each instruction bundle. In practice, however, it is infrequent for a system to reconfigure because of a newly discovered fault or because of changes in the hardware demands of a workload. For instance, it has been empirically shown that program execution often presents a phasic pattern, with sections of millions (or billions) of cycles where instructions rely only on a subset of the available hardware resources [17]. It thus makes sense to use the same set of resources over longer execution periods, amortizing hardware setup costs among many groups of instructions. The remainder of this section introduces the mechanisms that we developed in Cobra to extend the lifespan of a configuration.

A. *Hardware Configuration Transferring*

Cobra allows an instruction bundle to pass on its hardware configuration to the following bundle. Compared against a baseline distributed system that generates a hardware configuration for each new bundle, this enhancement does not entail any major design modification, as it only requires transferring the already established configuration to the scheduling unit assigned to manage the next bundle. This simple operation greatly reduces the cost of the service negotiation procedure, since – in the common case – it only consists of notifying the hardware components servicing the current bundle that they will also be employed by the following one. If the instructions

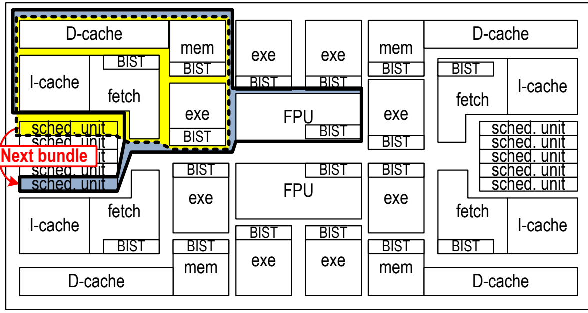

Fig. 3. When a scheduling unit and its associated hardware configuration (yellow area) complete the execution of a bundle, the hardware configuration is transferred to the scheduling unit servicing the following bundle (blue area), adding more units if needed (an FPU in the Figure).

in this latter bundle require additional services, the associated scheduling unit will initiate a service negotiation procedure to acquire providers only for the additionally required services. Note that this process serializes bundle execution; therefore Cobra trades off out-of-order performance to reduce hardware configuration overhead.

Figure 3 illustrates this approach with an example. The units in the yellow area belong to a hardware configuration that has completed the execution of a bundle. When the next bundle begins execution, its corresponding scheduling unit (marked in blue) receives the set of hardware units from the previous one. This setup information can be piggybacked on the notification that allocates a new bundle to a scheduling unit. Lastly, the bundle associated with the blue scheduling unit requires the services of a FPU, which is then added to the set of hardware units servicing the bundle.

B. Hardware Configuration Tearing Down

While our configuration transferring approach greatly reduces communication overhead, at times it is beneficial to tear down a configuration and create a new one from scratch. This is the case, for instance, when a program no longer needs a service, or when a fault manifests in an service provider from an active hardware configuration. In either circumstance, the corresponding scheduling unit simply prevents the transfer of its current hardware configuration to the following bundle.

To track the relevant events related to resource utilization, Cobra's scheduling units monitor resource utilization via a single-bit flag per service. Flags are propagated from a bundle to the next, updated every time a new instruction bundle is fetched, and reset periodically to keep them relevant.

Monitoring service utilization is also beneficial in managing scarce resources. Scheduling units that cannot obtain availability from one or more service providers, can request the scheduling units owning such resources not to forward their hardware configuration – thereby forcing a new service negotiation procedure. This solution guarantees forward progress and can rely on those same mechanisms used by hardware units to negotiate their services.

Finally, hardware configurations are torn down when at least one of their units is hit by a fault or must go offline for testing. At this point the scheduling unit that did not benefit from a transfer will proceed to set up a new hardware configuration – faulty units or units under test will simply not advertise their services, and thus they will not be included in any new configuration.

V. Temporary Data Persistency

Workloads running on a dynamically configured architecture may experience scattered execution due to the fact that consecutive bundles belonging to the same process can be executed by different hardware units. This severely affects the effectiveness of units that leverage temporal locality to enhance performance, such as caches and branch predictors.

In Cobra, this problem is solved as a by-product of our hardware configuration transferring approach. Indeed, when the same hardware configuration is used to service a large number of subsequent bundles, temporary data is naturally and effectively used in caching structures. Our experimental evaluation quantifies this benefit over a baseline distributed-control architecture.

VI. Boosting Memory Access Performance

Managing data memory operations is particularly challenging in a distributed architecture because program semantics expect memory operations to be issued and completed in order. This expectation may not be easy to meet when instructions from distinct bundles execute on different hardware configurations. Moreover, different bundles (possibly one logically following the other) might need to update a same memory location, leading to multiple "store"-service hardware units requesting exclusive access to the same cache line – a rather expensive procedure, as it requires all caches to invalidate their local copy.

A simple solution is to provide a single memory access point for the entire system. While this is effective in boosting the single cache hit rate, it comes with a high impact in memory access time and leads to poor system scalability. In contrast, our target is to keep a multitude of caches in the system, geographically distributed and relatively small, to boost performance. To this end, Cobra maps caches in the system to a set of threads: each thread in the set shares the cache assigned to it exclusively with the other threads in the same set. This is realized by mapping each Load-Store Queue (LSQ) to only one data cache. Since each hardware configuration (and thus, each thread) can only include one LSQ, this guarantees that memory ordering and data locality benefits can be attained within the set. At the same time, this solution provides system scalability, since overall, a system can still leverage multiple memory structures.

An available LSQ unit is assigned to the first bundle generated by a new thread – this is performed through the same negotiation mechanisms used to allocate any other service in the system. Systems that want to maximize throughput and availability can share a LSQ among multiple threads. The mapping between a thread and a LSQ unit is recorded in the relevant scheduling units, and it is propagated from one bundle to the next, releasing the LSQ only when a thread terminates or it is de-scheduled by the OS. Note that a bundle can be associated to a hardware configuration with multiple execution units; however, the configuration would still have only one LSQ unit. In addition, memory operations ordering is enforced by using the bundle sequence ID.

VII. Handling Runtime Failures

Cobra's objective is to maximize system availability in the face of hardware failures. In the remainder of this section we detail how distributed architectures can detect and manage runtime faults. Overcoming these events is a three stage process consisting of: 1) fault detection, 2) hardware reconfiguration, and 3) system state restoration.

First, a comprehensive reliable system must be able to dynamically detect errors and diagnose faulty components. Three techniques are available for this purpose: redundant execution, symptom-based detection, and online testing. Unfortunately, none of them has been tailored to distributed architectures, thus here we focus on enabling these fault detection mechanisms for this novel design paradigm, particularly Cobra.

Second, the distributed processor architecture developed in Cobra empowers a system to automatically reconfigure itself around hardware errors. Hardware units deemed faulty can be selectively turned off, thus preventing them from advertising their services to the rest of the system.

Third, independently from the fault detection mechanism deployed, upon fault detection all bundles in flight are flushed through a system-wide broadcast signal to all scheduling units. The hardware failure is then diagnosed, and the newly discovered faulty component disabled. Both architectural state and memory system is restored to a previous safe checkpoint through techniques such as ReVive or SafetyNet [21, 24], and each checkpointed program is restored and mapped to an available scheduling unit. Cobra stores the information necessary to manage program execution in the scheduling units, while the architectural register values are located in the reconfigurable fabric. These two pieces of information are synchronized to build a single checkpoint state when instruction bundles commit.

It is worth noting that handling faults in the scheduling units does not require advanced mechanisms. The vast majority of the area in these units consists of storage elements, which can be protected through ECC, while their relatively small logic can be made resilient through hardware duplication [18]. Finally, in this work we do not account for failures on caches and on-chip interconnect, as these subsystems can be effectively protected by other techniques [1, 9].

The remainder of this section considers three classes of fault detection solutions: full redundancy, selective redundancy and periodic online testing.

A. Full Redundancy

Full redundancy is the simplest and fastest mechanism to detect any type of hardware errors. Each instruction executes multiple times on different hardware components, and the results are then compared to detect or possibly correct eventual errors. Although highly effective, full redundancy is very performance and resource demanding, and only applications that value reliability as a key requirement adopt it. Corrupted results are recognized immediately, and a faulty hardware component is diagnosed by comparing the outcomes of multiple executions.

To deploy this fault detection mechanism in Cobra, we enhance the scheduling units to accommodate redundant hardware configurations, containing non-overlapping sets of service providers. In addition, they must include a voting unit to compare results produced by the redundant configurations. This check can be done at a fine granularity, comparing each result produced by every instruction, or at the bundle level, comparing only the results that are transferred to subsequent bundles. In addition, the scheduling unit must allow results to be committed to memory only after they have been successfully checked.

Bundles to be executed redundantly are tagged by a special flag, so that all units servicing such bundles are aware that the instructions within require special handling: they are not allowed to alter process state, memory, or program flow until after they have been deemed fault-free – in our system we maintain a single software process for all the redundant executions.

B. Selective Redundancy

Most programs do not require the degree of protection guaranteed by a fully redundant execution. In fact, it has been shown that simply monitoring software anomalies, such as kernel panics, fatal traps and illegal memory accesses, can reveal up to 95% of permanent faults and 60% of transient faults [8, 13, 28].

Unfortunately, even though this approach provides a significant fault coverage for most hardware failures, it is not very effective in detecting faults that silently corrupt an application's outputs. Cobra provides an ad-hoc solution for checking the portions of a workload that are particularly susceptible to silent data corruptions (SDC). It has been empirically shown that particular types of instructions – integer divisions and multiplications, floating point operations, and SIMD instructions – are particularly prone to SDCs [13, 19]. Since the scheduling units of a distributed system maintain detailed information about the services required by the executing bundles, they can dynamically flag the portions of a program that are vulnerable to SDCs. Therefore, we allow processes to request "selective redundancy": Cobra will activate redundant executions only for the bundles that include operations vulnerable to SDCs. This technique represents a good compromise for processes that need high fault coverage but that cannot afford the cost of a fully redundant execution, and it is also very effective in exposing permanent failures [16].

C. Periodic Online Testing

Periodic online testing is a low-cost solution to protect software from permanent hardware failures. This approach assumes that results generated by a processor cannot be trusted until the underlying hardware components have been tested. Only after all tests succeed, a process is allowed to commit its results. With this approach, execution time is partitioned into epochs, which are typically a few million of instructions long, and hardware tests are executed at the end of each epoch. Periodic hardware testing has been shown to be both very economical and effective [7, 17]. Handling faults in the test logic is a problem common to all solutions that adopt this fault detection technique, and in this work we assume that a faulty self-test unit causes its related hardware component to be marked as non-functional.

VIII. Experimental Setup

We modeled a Cobra system that implemented the x86-64 ISA and evaluated both its performance scalability and its ability to endure hardware failures. To this end, we compared Cobra against two similarly sized designs: a classic CMP comprising 2-wide out-of-order cores and an unoptimized Viper design [18]. We chose the former because it matches the characteristics of modern CMPs [4, 10], and the latter because it represents a state-of-the-art distributed architecture. In order to measure Cobra's scalability, we considered systems which can execute 1, 2, 4, 8 and 16 threads, and whose processor logic (caches excluded) occupies 20M, 40M, 80M, 160M and 320M transistors, respectively.

A. Hardware Model

All three architectures evaluated in this work are clocked at a frequency of 2.0GHz. Cores in the classic CMP configuration have an execution window of 32 instructions, can commit up to

Service	Hardware unit	Number of units	Test cycles	Transistors
Fetch bundle	Fetch	4	1.25M	4M
Generate next PC				
Decode bundle	Decode	4	1.25M	2.5M
Tag generation	Tag	4	1.25M	3M
Integer ALU	Execute	8	1.25M	1.6M
Load & Store		4		951K
Integer mult. & div.		8	327K	1.27M
FP ALU		8	230K	635K
FP mult. & div.		8	230K	635K
SIMD		4	1.57M	635K
Update register file	Commit	4	1.25M	1M
Commit stores				
Create new bundle	Scheduling Unit	32	–	15K

TABLE I
CHARACTERISTICS OF THE DISTRIBUTED ARCHITECTURES EVALUATED: COBRA AND VIPER [18]. BOTH ARCHITECTURES PROVIDE 13 SERVICES USING 6 DIFFERENT HARDWARE UNITS.

2 instructions per cycle, and contain: 2 integer pipelines, 2 FP units, 1 load/store unit. Each OoO core uses dedicated 32KB L1 data and instruction caches, while all service providers in Cobra and in the baseline distributed architecture can make use of similarly sized data and instruction caches – one for each thread executed on the machine. In order to ensure program correctness, both distributed architectures tie a process to only one data cache before starting execution, as discussed in Section VI.

To setup both the distributed architectures under analysis, Cobra and Viper, we partitioned the x86-64 ISA into 13 different services, which are provided by six different hardware units, as listed in Table I. In our fault model, a failure hitting a multi-service unit disables only one service in that unit. For instance, an "Execution" unit, which is hit by a fault in its "FPU ALU" service provider, will no longer be able to execute any bundle that requires that service, but can still provide its other services. Periodic self-tests are scheduled independently on each unit after a certain number of completed instructions, 20M in our case [17]. In order to trigger self-tests, we embed an instruction counter in each hardware unit: once the limit is reached, the hardware tests are serially performed on all the services still available. With reference to our example, the "Execution" unit would skip testing the hardware of its already faulty "FPU ALU" service. Table I reports several characteristics of the distributed architectures considered. Configurations supporting a different number of programs are scaled proportionally (the 16-threads configuration will have 16 fetch units). The last two columns of the table report the number of cycles required to test each service and an area estimate for the unit. Connectivity between scheduling units and service providers is established through a crossbar, which has a point-to-point latency of 4 cycles [29]. Both distributed architectures arrange the "Fetch", "Decode", "Tag", "Execute" and "Commit" units in a mesh, each unit connects to its neighbors with 256-bit wide links. The communication latency between two adjacent nodes is 1 cycle [18].

B. Software Benchmarks

We used the SPEC CPU2006 benchmark suite [12] to evaluate Cobra's performance. Due to the detail and complexity of our simulations, we could not run all benchmarks to completion. Instead, we evaluated their performance when they reached a steady execution state. The benchmarks in this suite were run with the "test" input set. We evaluated perfor-

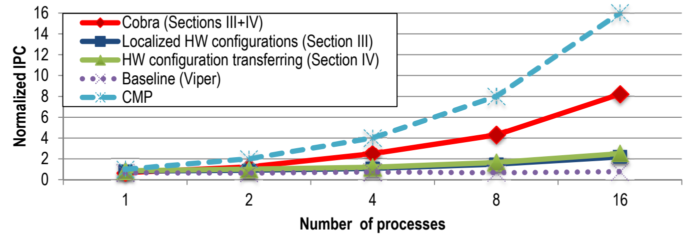

Fig. 4. Throughput vs. system's size. The plot compares Cobra (solid lines) against a CMP and a baseline distributed architecture.

mance for both single– and multi– programmed workloads, executing independent copies of the same benchmark. In order to measure Cobra's scalability we evaluated a system executing 1, 2, 4, 8 and 16 programs, and scaled the resources available accordingly. As for the other experiments, for reasons of space, we focused our attention on a medium-sized system executing four programs, and analyzed in more detail its performance and reliability. We decided to run multiple copies of the same benchmark at the same time – instead of a mix of benchmarks – because we wanted to stress the system by creating high contention on the hardware components exercised by their instructions. Finally, reliability estimations are reported as throughput on the multiprogrammed workloads running four copies of the same program.

C. Simulation Infrastructure

The microarchitectural simulation platform adopted for this work is based on the gem5 simulator [3], employing full timing simulations in system-call emulation mode. The model of the OoO core provided with gem5 was modified to match the deeper pipelines typical of modern high-performance processors. The minimal execution latency of an instruction in such design is 12 cycles. We developed a model for the distributed architectures under evaluation building on the template of the OoO core publicly available with the gem5 simulator.

Fault Model – Since gem5 does not natively include a fault injection model, we augmented the baseline simulator with: 1) a parameter representing transistor count for each hardware component; 2) a fault injector capable of randomly triggering a fault in one of the hardware components.

We focus on permanent failures, which were injected into the components listed in Table I with a uniform probability proportional to the number of transistors in each. In order to gain statistical confidence in our results, each fault injection experiment was repeated 20 times, each one with a different random selection of fault locations. Transistor counts are estimated from known transistor counts of modern commercial designs [18]. Faulty service providers are selected and disabled before starting the simulations. Note that we did not consider defects in memory arrays since those can be easily avoided through ECC and redundant entries. Finally, we did not inject faults in the intra-chip interconnect, as other techniques can protect the communication subsystem from hardware failures [9].

IX. EXPERIMENTAL RESULTS

We first analyzed Cobra's performance and scalability compared to the CMP and the unoptimized distributed system. Hence, we measured the impact of the solutions developed in Cobra on the throughput of systems of different sizes, running multiple instances of the SPEC2006 benchmarks. No faults were injected in these first simulations.

We then evaluated Cobra's robustness to an increasing

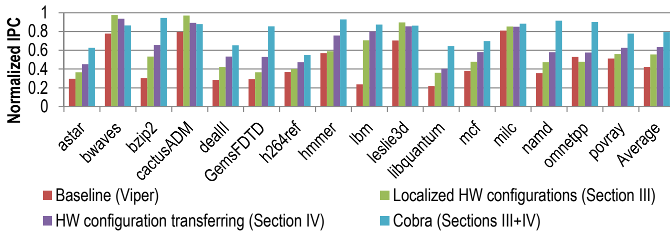

Fig. 5. Contribution of Cobra's performance features to overall IPC for a 4-threaded system. IPC is normalized to that of a similarly-sized CMP.

number of permanent failures and compared it against a classic CMP solution. To the best of our knowledge, we are the first to quantitatively assess the impact of hardware failures on a fully decentralized architecture – previous work only provided reliability projections for such designs [18]. Finally, we investigate the performance impact of deploying online fault detection mechanisms on Cobra.

A. Fault-free Throughput

In the first set of experiments, we compared the throughput of a CMP design and distributed architecture against Cobra, and evaluated the impact of each technique discussed in Sections III (localized hardware configurations) and IV (hardware configuration transferring). We did the comparison using 1, 2, 4, 8 and 16 concurrent processes to gauge the scalability of the systems analyzed. Our findings are reported in Figure 4, where we compare the throughput of Cobra (3 solid lines), against that of a similarly sized CMP design and an unoptimized baseline distributed solution (Viper). It can be noted that our two performance-boosting techniques, "localized HW configurations" and "HW configuration transferring" enable a distributed-control solution to approach the performance and the scalability of a classic CMP system.

Localized Hardware Configuration – To avoid starvation, scheduling units must assign hardware units to services in an ordered fashion, thus we proceed top-down through the services listed in Table I. The unoptimized Viper design in our experiments allocates each required service to the first hardware unit available in a greedy fashion. In contrast, Cobra uses our localized configuration approach, buffering service proposals for some cycles (in our experiments we used 10, that is, slightly more than twice the crossbar transmission latency). Figure 5 plots the contribution of this technique over a baseline distributed architecture (Viper) in a 4-threaded configuration for each SPEC2006 benchmarks, showing that it betters performance by 23% on average.

Hardware Configuration Transferring – We then evaluated the impact of allowing a bundle to directly transfer its hardware configuration to its successor. Hardware configurations are torn down every 20 million instructions (a reasonable length for a computational epoch, as shown in [7]). This technique brings an average performance improvement of 42% on a 4-threaded system, as indicated in Figure 5. Correspondingly, we observed a significant reduction in the number of messages through the crossbar – 61% and 52% for single process and multiprogrammed benchmarks, respectively.

Cobra – Our experiments report that these two techniques combined contribute an average performance boost of 79% over an unoptimized distributed design. When compared to a CMP system, the performance of Cobra falls short by only 21% – a small incidence when considering its reliability.

We also evaluated the ability of Cobra to boost memory access performance. Data caches are affected by only a minimal performance difference, since both Cobra and Viper map one cache per process to maintain correct execution. In contrast, instruction caches improve their hit rate by 16%, on average.

B. Reliability

The next set of results evaluates Cobra's reliability and measures the effects of permanent faults on its performance. For these experiments we only considered the multiprogrammed SPEC2006 benchmarks and injected faults at the beginning of each simulation. Since our reliability-boosting techniques are unaffected by the scale of the system, we only provide results for 4-threaded systems and workloads.

Performance in presence of faults – We measured the performance degradation of our system when subjected to permanent failures as reported in Figure 6. For this study we only considered faulty systems that can still execute all ISA instructions, and we averaged our results over all SPEC2006 benchmarks, running 20 distinct simulations per benchmark to gain statistical confidence.

It can be noted that Cobra's average throughput degrades gracefully with increasing faults, roughly halving at 15 faults. Figure 6 reports max and min relative IPC, in addition to the average overall all 20 runs for the 24 benchmarks. We observed that injecting a moderate number of faults (between 1 and 5) can occasionally lead to a performance boost (as it can be noted from the max line in Figure 6). This is due to the fact that our resource assignment algorithm is based on algorithms that searches for a local optimum. Thus, small variations in the available resources may lead to a better-performing solution.

Finally, Figure 7 compares Cobra's performance against that of a CMP design in presence of faults. Values are reported relatively to the performance of a single fault-free OoO core. The CMP design considered in these experiments is not affected by the interactions between the different cores; thus, in its fault-free state, its performance is 4 times that of the reference OoO core. The curves reporting Cobra's performance are obtained by disabling faulty components at the hardware unit or service granularity (see Table I for a list of units and services). Note that under fault-free conditions, Cobra is outperformed by the CMP by more than a factor of 2. This is due to Cobra's larger area footprint and its overhead to setup and manage dynamic hardware configurations. However, as faults increase, the margin of benefit is reduced quickly with Cobra providing better performance after only 4 faults. Since the systems we modeled roughly occupy 100 million transistor, this crossover point in our experiments occurs when a device is affected by 1 fault every 20M transistors.

C. Online Fault Detection

Finally, we evaluate the performance impact of deploying online fault detection mechanisms in Cobra. The full redundancy approach (dual-module redundancy) experiences the highest performance costs of 26% for single-process benchmarks and 61% for multiprogrammed ones. The reason for the smaller impact on single-process programs lies on the

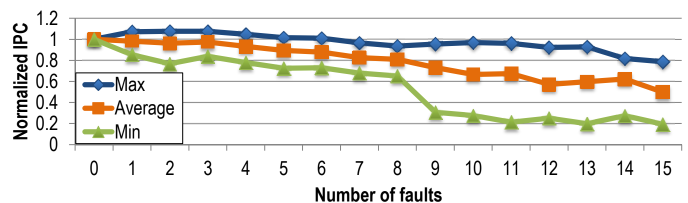

Fig. 6. Cobra's performance degradation in presence of failures, averaged over 24 SPEC2006 benchmarks. Results are normalized to a fault-free Cobra.

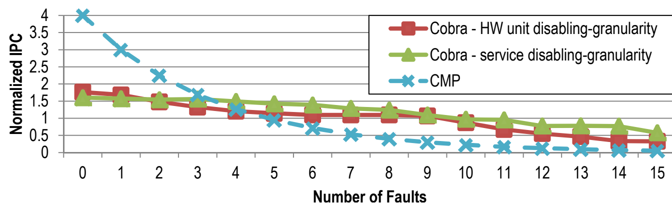

Fig. 7. Cobra's performance degradation in presence of failures vs. a CMP. Cobra outperforms the CMP beyond 4 faults.

multiplicity of hardware units available, which can be used to hide the additional computation required. The performance cost of selective redundancy is less steep: 19% and 26% for single- and multi- process benchmarks, respectively. For this solution we protected with dual redundancy only bundles including FPU and SIMD operations and mult/div instructions.

Finally, the performance impact of periodic testing is only 3%. We implemented this solution by performing a periodic self-test of all hardware units (not already known to be faulty) every 20M instructions – a typical interval for processors self-tests [7, 17]. The performance we measured is much lower than that of similar approaches in pipelined architectures (16% as reported in [17]). Such limited impact is due to the fact that in Cobra it is straightforward to take a unit off-line temporarily for testing purposes, without affecting the rest of the system.

X. Related Work

High availability commercial systems ensure high reliability by sporting modular redundant configurations and thus invest large portions of silicon area and large performance overhead to guarantee performance in the face of failures [2].

Several research focused on developing low-cost fault-tolerant techniques for classic pipelined processors. Proposed solutions can rely on online testing [17], runtime fault detection [14], defect isolation [11] or replication of hardware units [6]. Researchers have also considered options that salvage partially functional cores to improve total system availability [20].

Core cannibalization is a first example of distributed architecture [22]. This CMP architecture comprises several simple cores and different pipeline stages from each of them can be composed to build a functional processor. StageNet extends this concept to a next level: its reconfigurable fabric connects multiple hardware units, each of which can perform the tasks associated with the individual stages of a typical pipelined processor [11]. The hardware stages are partitioned into islands to allow the solution to scale, but this constraints the system's connectivity and reconfigurability. Viper is the first design to propose a service-oriented execution paradigm [18]. Since it is affected by a number of limitations typical of distributed-control architectures, its performance and scalability compares poorly against traditional CMPs. In contrast, Cobra's performance is very close to that of a CMP under no-fault conditions, and it quickly out-performs faulty CMPs.

XI. Conclusions

We presented Cobra, a new reliable and scalable distributed-control architecture. We designed a novel memory organization and developed a new algorithm to dynamically assign available hardware resources to instructions to be executed. As a result, Cobra provides high system scalability and boosts the performance of distributed-control architectures. In addition, Cobra enables the adoption of low-cost fault detection mechanisms on a distributed architecture. By analyzing Cobra's reliability, we found that it outperforms a traditional CMP design beyond the occurrence of 1 fault per 20M transistors in our setup, while the performance cost of online fault detection is only 3%.

Acknowledgements. This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] A. R. Alameldeen *et al.*, "Energy-efficient cache design using variable-strength error-correcting codes," in *ISCA*, 2011.
[2] W. Bartlett and L. Spainhower, "Commercial fault tolerance: a tale of two systems," in *DSN*, 2004.
[3] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.
[4] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007.
[5] S. Borkar, N. Jouppi, and P. Stenstrom, "Microprocessors in the era of terascale integration," in *DATE*, 2007.
[6] F. Bower *et al.*, "Tolerating hard faults in microprocessor array structures," in *DSN*, 2004.
[7] K. Constantinides *et al.*, "Software-based online detection of hardware defects mechanisms, architectural support, and evaluation," in *MICRO*, 2007.
[8] S. Feng *et al.*, "Shoestring: probabilistic soft error reliability on the cheap," in *ASPLOS*, 2010.
[9] D. Fick *et al.*, "Vicis: a reliable network for unreliable silicon," in *DAC*, 2009.
[10] R. Golla and P. Jordan, "T4: a highly-threaded, server-on-a-chip with native support for heterogenous computing," in *Hot Chips*, Aug 2011.
[11] S. Gupta *et al.*, "StageWeb: interweaving pipeline stages into a wearout and variation tolerant CMP fabric," in *DSN*, 2010.
[12] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
[13] M. Li *et al.*, "Understanding the propagation of hard errors to software and implications for resilient systems design," in *ASPLOS*, 2008.
[14] A. Meixner, M. Bauer, and D. Sorin, "Argus: low-cost, comprehensive error detection in simple cores," in *MICRO*, 2007.
[15] A. Meixner and D. Sorin, "Detouring: translating software to circumvent hard faults in simple cores," in *DSN*, 2008.
[16] S. Nomura *et al.*, "Sampling + DMR: practical and low-overhead permanent fault detection," in *ISCA*, 2011.
[17] A. Pellegrini and V. Bertacco, "Application-aware diagnosis of runtime hardware faults," in *ICCAD*, 2010.
[18] A. Pellegrini, J. Greathouse, and V. Bertacco, "Viper: virtual pipelines for enhanced reliability," in *ISCA*, 2012.
[19] A. Pellegrini *et al.*, "CrashTest'ing SWAT: accurate, gate-level evaluation of symptom-based resiliency solutions," in *DATE*, 2012.
[20] M. Powell *et al.*, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *ISCA*, 2009.
[21] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: cost-effective architectural support for rollback recovery in shared-mem multiprocessors," in *ISCA*, 2002.
[22] B. Romanescu and D. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults," in *PACT*, 2008.
[23] S. Shamshiri and K.-T. Cheng, "Modeling yield, cost, and quality of a spare-enhanced multicore chip," *IEEE Trans. Computers*, vol. 60, no. 9, pp. 1246–1259, 2011.
[24] D. Sorin *et al.*, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *ISCA*, 2002.
[25] A. Strong *et al.*, *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley Press, 2009.
[26] M. Strus, R. Keller, and N. Barbosa, "Electrical reliability and breakdown mechanisms in single-walled carbon nanotubes," in *NANO*, 2011.
[27] S. Swanson *et al.*, "WaveScalar," in *MICRO*, 2003.
[28] N. Wang and S. Patel, "ReStore: symptom based soft error detection in microprocessors," in *DSN*, 2005.
[29] M. Woh *et al.*, "Low power interconnects for SIMD computers," in *DATE*, 2011.