Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware*

Smruti R. Sarangi, **Abhishek Tiwari**, and **Josep Torrellas**
University of Illinois at Urbana-Champaign
{sarangi,atiwari,torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

Although processor design verification consumes ever-increasing resources, many design defects still slip into production silicon. In a few cases, such bugs have caused expensive chip recalls. To truly improve productivity, hardware bugs should be handled like system software ones, with vendors periodically releasing patches to fix hardware in the field.

Based on an analysis of serious design defects in current AMD, Intel, IBM, and Motorola processors, this paper proposes and evaluates *Phoenix* — novel field-programmable on-chip hardware that detects and recovers from design defects. Phoenix taps key logic signals and, based on downloaded defect signatures, combines the signals into conditions that flag defects. On defect detection, Phoenix flushes the pipeline and either retries or invokes a customized recovery handler. Phoenix induces negligible slowdown, while adding only 0.05% area and 0.48% wire overheads. Phoenix detects all the serious defects that are triggered by concurrent control signals. Moreover, it recovers from most of them, and simplifies recovery for the rest. Finally, we present an algorithm to automatically size Phoenix for new processors.

1. Introduction

The complexity of today's high-performance processor designs has grown to a point where design verification is a major bottleneck [5, 7, 8]. Processor verification is a multi-year effort that is highly labor and compute intensive. It involves running formal verification tools and extensive test suites during both pre- and post-silicon stages, in a race to meet production shipment qualification. Overall, verification accounts for 50-70% of processor development time [4].

Unfortunately, even with all these resources, many defects still slip into production silicon. Perhaps the most notorious one is the Pentium floating-point division bug [11], which caused an error in the 9^{th} or 10^{th} decimal digit. It lead to a $500 million chip recall. A 1999 defect in the Pentium III [9] led original equipment manufacturers to temporarily stop shipping Intel servers. Problems in the cache and prefetch engine of the Pentium 4 [21] led to disabling prefetching in multiprocessor systems. More recently, defects led to a recall of Itanium 2 processors [10], incorrect results in the AMD Athlon-64 [27], and circuit errors in the IBM PPC 750GX [28]. The latter processor had some instructions that could not run at the rated 1 GHz, and IBM recommended running at 933 MHz. Less conspicuously, practically all processors in the field have several tens of known design defects, as reported by manufacturers in "errata" documents [1, 13, 15, 22].

There is every indication that this problem is about to get worse. Moore's law is enabling more design integration [18], increasing verification effort while hurting signal observability. Larger verification teams increase the risk of defects due to miscommunication. The ambiguities of the many new standards to support will also contribute to defects. All this suggests that, in addition to better verification tools, we need novel approaches to handle defects.

Ideally, we would like processor design defects to be treated like system software bugs. When the chip vendor discovers a new defect in a processor, it should be able to broadcast a "hardware patch" to all the chips in the field to fix them. Once installed, the patch should automatically detect when the defect is about to be exercised, and either avert it or repair its effect on the fly.

Such support would have two obvious benefits. First, it would enable in-the-field repair of incorrect behavior, avoiding erroneous execution and expensive chip recalls. Second, it would conceivably allow a company to release production silicon to market earlier, without so much in-house testing, thereby gaining a valuable edge over competitors. For example, the last ten weeks of testing could be saved, when the rate of detection of new bugs decreases to near zero [8].

Sadly, the state of the art is far from this vision. Perhaps the closest approaches are those of Itanium and Crusoe. In Itanium, some important functions such as TLB and FP unit control are supported in firmware, which is correctable with patches [17]. While this EPIC feature gives flexibility, it is slower than a hardware implementation. We would like the whole chip to be patchable and have no performance cost at all. Crusoe uses code translation and, therefore, can work around defects by changing the translation software [20]. However, most processors do not support this approach. Note also that patchable microcode such as IBM's Millicode [12] is not a solution either, since a complex design defect is not very correlated with the execution of any given instruction opcode.

As a step toward our vision, this paper makes three contributions. First, by analyzing the design defects in AMD, Intel, IBM, and Motorola processors [1, 13, 15, 22], we gain insight into how to detect and recover from them. Serious design defects are consistently concentrated in the core's periphery and cache hierarchy. Moreover, many of them can be detected before they have corrupted the system.

Second, we propose *Phoenix*, novel field-programmable on-chip hardware that detects and recovers from design defects. Phoenix taps key control signals and, using downloaded defect signatures, com-

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

Label	Processor	First - Last Version Considered	Features	Freq (MHz)	Area (mm2)	# Trans (Mill.)	# Design Defects
K6	AMD K6-2	Aug'98-July'99	4-issue core, SIMD support	550	81	9	18
P3	Intel Pentium III	May'99-Nov'04	On-chip L2 cache, SSE instructions, P6 core	1,200	79	44	92
Athlon	AMD Athlon (32 bit)	Aug'00-Oct'03	9 FUs, pipelined FPU	2,200	115	54	19
P4	Intel Pentium 4	Nov'00-Nov'04	Trace cache,hyperthreading,20-stage pipeline	3,400	131	55	99
Itan1	Itanium (3 MB L3)	June'01-May'03	EPIC arch, 10-stage pipeline, L3 cache	800	465	220	40
IBM-G3	IBM 750FX (G3)	Apr'02-Nov'04	2-issue RISC machine	1,000	37	30	27
Itan2	Itanium 2	July'02-Feb'05	Higher frequency, more scalable than Itan1	1,800	374	410	103
Mot-G4	Motorola MPC7457	Feb'03-Nov'04	Improved FPU, AltiVec insts, L3 cache cntrl.	1,333	98	58	32
P-M	Intel Pentium M	Mar'03-Dec'04	Similar to P3, emphasis on power efficiency	2,200	84	140	33
Athl64	AMD Athlon-64	Apr'03-June'04	Mem. contr. on chip, hypertransport, 64 bit	2,400	193	106	48

Table 1. Processor versions examined in this paper.

bines the signals into conditions that flag defects. When a defect is detected, Phoenix flushes the pipeline and either retries or invokes a customized recovery handler. We also present an algorithm to automatically size Phoenix for new processors.

Finally, we evaluate Phoenix. Phoenix induces negligible slowdown. Our design taps about 200 signals and adds only 0.05% area and 0.48% wire overheads to the chip. Phoenix detects all the serious defects that are triggered by concurrent control signals. Moreover, it recovers from most of them, while simplifying recovery for the rest. Given the very high cost of processor defects, we believe this coverage fully justifies Phoenix. Finally, our algorithm effectively sizes Phoenix for new processors of completely different types.

This paper is organized as follows. Section 2 characterizes design defects; Sections 3 and 4 present and argue for Phoenix; Section 5 evaluates it; and Section 6 describes related work.

2. Characterizing Processor Design Defects

Commercial microprocessors in the field are known to have design defects — popularly known as hardware bugs. When these bugs are exercised, they have serious consequences such as a processor crash, data corruption, I/O failure, wrong computation, or a processor hang. Moreover, as new generations of processor chips become more complicated, design defects are expected to become more prevalent [4, 18].

Processor manufacturers list these bugs in errata documents [1, 13, 15, 22] that are updated every few months, as defects are discovered and fixed — typically, with a silicon re-spin. For each defect, the document lists the condition that triggers it. A condition is a set of events, such as an L2 cache miss, a snoop request, and an I/O interrupt, with timing information. Sometimes, the document also gives a workaround to avoid exercising the defect. Workarounds often involve disabling hardware features such as the power control manager or multiprocessor support.

In Table 1, we take ten recent processors and list the version range considered, key architectural features, frequency, approximate area and number of transistors, and number of design defects reported. In the number of defects, we include all the different ones in all versions. While some may have been fixed in some revision, they still exist in the field.

From the table, we see that the number of design defects per processor ranges from 18 to 103. As an illustration, Table 2 describes three defects. The first one can occur if the L1 suffers a cache miss while the power manager is on and the L2 is being flushed. In this case, some L2 lines may get corrupted. The second defect involves four concurrent bus and cache conditions that cause a system deadlock. The third defect is a pipeline timing issue that induces incorrect results.

Defect	Proc.	Defect Description
Defect1	IBM-G3	If the L1 suffers a miss while the power manager is on and the processor is flushing its L2, some L2 lines may get corrupted. [Signal condition: L1WAITMISS & DPM (dynamic power management) & L2FLUSH].
Defect2	P4	If a cache hits on modified data (HITM) while a snoop is going on, and there are pending requests to defer the transaction and to re-initialize the bus, then the snoop is dropped, leading to a deadlock. [Signal condition: SNOOP & HITM & DEFER & BUSINIT].
Defect3	Athl64	When an Adjust after Multiply (AAM) instruction is followed by another AAM within three instructions, or is preceded by a DIV instruction by up to 6 instructions, the ALU produces incorrect results.

Table 2. Examples of processor defects.

Figure 1 classifies the defects into those appearing in non-critical structures (*NonCritical*) and the rest (*Critical*). NonCritical are defects in modules such as performance counters, error reporting registers, or breakpoint support. They also include Itanium firmware defects, which are correctable with patches.

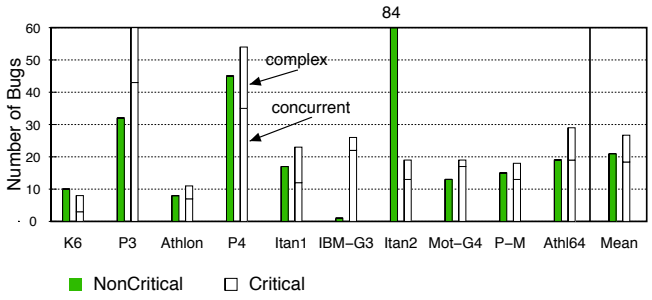


Figure 1. Classification of design defects in each processor.

Focusing on the Critical defects, we see that, on average, there are slightly over 25 of them per processor. We divide them into two groups. One group is those that are triggered by a relatively simple combination of *concurrent* signals. We call these defects *Concurrent*. *Defect1* and *Defect2* in Table 2 are Concurrent defects. The remaining Critical defects have more complex triggering conditions. They typically depend on some internal state and a sequence of events (e.g., when event E has occurred in the past, if signal S is asserted ...). We

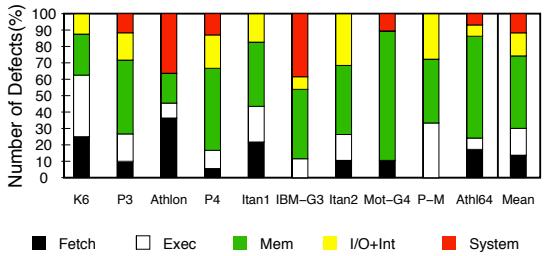


Figure 2. Classification of the Critical defects based on which module they are in.

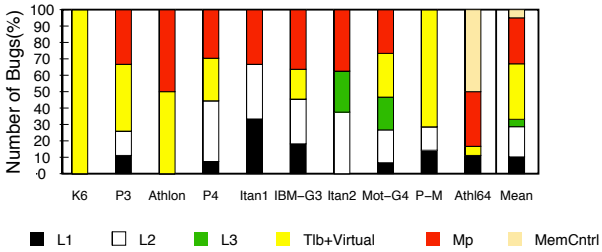


Figure 3. Classification of the Critical defects in the memory module.

call these defects *Complex*. *Defect3* in Table 2 is a Complex defect. From Figure 1, we see that on average 69% of the Critical defects are Concurrent.

2.1. Where Are the Critical Defects?

Figure 2 classifies the Critical defects according to the module in which they are: fetch unit (including L1 I-cache); execution unit; memory subsystem (rest of the caches, memory bus, and multiprocessor modules); I/O and interrupt subsystems (*I/O+Int*); and "system-related" units (*System*). The latter include the power and temperature management unit, the voltage/frequency scaling control unit, and the clock distribution network.

The largest fraction of defects occurs in the memory subsystem. Moreover, if we add up the *Mem*, *I/O+Int*, and *System* defects, we see that the defects in the "periphery" of the core dominate. This is because the inner core (*Fetch* and *Exec*) is usually more thoroughly tested and its components are typically reused across projects more.

Figure 3 classifies memory system defects based on where they are: data-L1, L2, L3, TLB plus virtual memory system, multiprocessor structures (*Mp*), and memory controller (*MemCntrl*). We see that, on average, TLB plus virtual memory and MP structures dominate. Overall, a defect-detection scheme largely focused on the inner-core such as DIVA [2] will not work well. Instead, our field-programmable scheme should draw most of its input signals from the periphery in general, and from the TLB plus virtual memory and multiprocessor structures in particular.

2.1.1. Analysis Across Generations

We gain additional insight by examining how the Critical defects change across generations of the same processor family. Figures 4 and 5 show the Intel Pentium and AMD processor families, respectively. The figures contain the relevant bars from Figure 2 in absolute terms. Figure 4 also includes bars for the Pentium and Pentium-II.

The defect profile from the Pentium to the Pentium 4 remains similar, both in absolute and relative terms. This is despite the fact that design teams reuse their know-how across generations, and that it can be shown that a given defect very rarely persists across generations. This suggests that when designers lay out the Phoenix hardware, they can use their experience from prior generations to decide how to distribute it in the new chip.

The Pentium-M in Figure 4 and the Athlon-64 in Figure 5 suggest that when a design is significantly simplified (the Pentium-M has no MP support) or enhanced (the Athlon-64 has an on-chip memory controller and more), the absolute number of defects can change

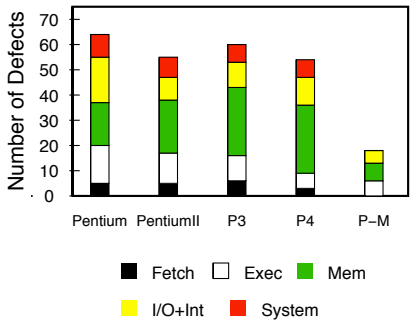


Figure 4. Critical defects across Pentium processors.

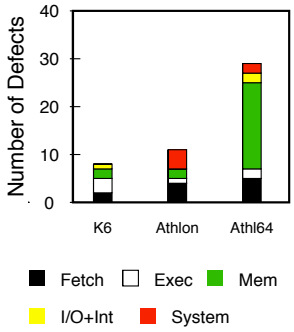


Figure 5. Critical defects across AMD processors.

a lot. Designers can use this knowledge to size the Phoenix hardware for a new chip.

2.2. How Do Defects Manifest & Can Be Worked Around?

To further understand the Critical defects, Figure 6 shows how they manifest. They corrupt caches or memory (*Corruption*), hang the processor (*Hang*), cause an I/O failure (*I/O-failure*), compute, load or store wrong data or addresses (*Wrong*), or have unpredictable effects (*Unpred*). The latter range from application error to processor shutdown, and are often caused by defects in the temperature or power manager. We observe that the manifestations are largely catastrophic.

Figure 7 shows the type of workarounds that the vendor proposes for these defects. Workarounds can be based on hardware (*HW*), compiler (*Compiler*), operating system (*OS*), firmware (*FW*), or be non-existent. Table 3 shows some examples.

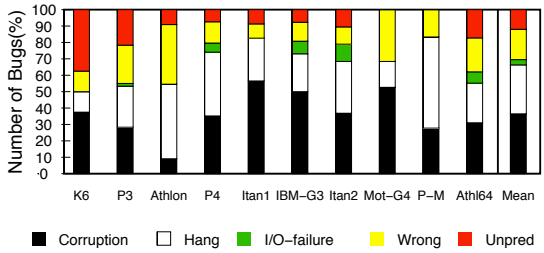


Figure 6. Classification of the Critical defects based on how they manifest.

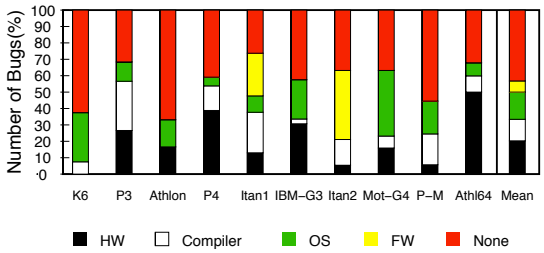


Figure 7. Classification of the workarounds for Critical defects.

Type	Process	Examples of Workarounds
Hardware	Change bios chip or modify motherboard	Disable certain feature (hyperthreading, MP, prefetching, ECC, memory scrubbing). Limit the voltage or frequency modes of different parts of the system (CPU, memory, I/O). Limit the settings for clock gating or memory refresh. Connect pins to certain logic values.
Compiler	Compiler changes	Add fences. Separate certain instruction types. Put no-ops. Avoid certain instruction combinations.
OS	OS patch	Use certain order to initialize memory or I/O devices. Process interrupts in a certain order. Add fences. Disable prefetching. Limit the settings for power, frequency, or paging (size and segmentation parameters).
Firmware	Firmware patch	Change firmware for FP ops, interrupt handling, ECC checking, page table walking, and TLB miss handling.

Table 3. Characterization of existing workarounds. They typically impair the system in some way.

Figure 7 shows that on average less than 60% of the Critical defects have workarounds. Of these, *HW* workarounds are mostly targeted to chip-set designers and equipment manufacturers, and typically impair the processor. *OS* and *Compiler* workarounds are targeted to OS and compiler developers, respectively. They range from minimal to significant software patches, often have as many versions as supported OS versions, and can lead to the removal of features. Finally, *FW* workarounds are found only for some Itanium defects. Note that the defects are *not* in the firmware — the workarounds are. They involve patches that the user can download. These are not patches that implement an operation differently; they typically disable functionality. Overall, except for *FW* workarounds, the user can do little to avoid these defects. Our goal is to extend a *FW*-like workaround procedure to all defects without any performance cost.

2.3. How Can Defects Be Detected and Avoided?

Finally, we examine some traits of the defects that can help us detect and avoid them. To detect a defect, we must identify the combination of signal conditions that can trigger it. This is reasonably easy to do for Concurrent defects. However, it is harder for Complex ones because they are caused by complex state and sequences of events. Therefore, in this paper, we focus on Concurrent defects, which are 69% of the Critical ones.

Among the Concurrent defects, we observe that there are those whose triggering condition can be detected before any damage is done. We call these defects *Pre*, for pre-detection. One example is *Defect1* in Table 2: if the processor is about to flush its L2 (L2FLUSH signal on) and the dynamic power manager is on (DPM signal on), there is a possibility that, in the middle of the flush operation, the L1 suffers a miss and *Defect1* is triggered. As another example, there are combinations of instructions that trigger a defect when they execute or commit, and such combinations can be detected earlier in the pipeline, before exercising the defect.

The other Concurrent defects have triggering conditions that cannot be detected before the damage is potentially done. We call these defects *Post*, for post-detection. An example is *Defect2* in Table 2: by the time we detect that the four conditions occur, the snoop may have been dropped.

Figure 8 breaks down Concurrent defects into *Pre* and *Post*. In nearly all cases, *Pre* dominate. On average, they account for 60% of the Concurrent defects. Typically, their triggering conditions are detected several cycles before the defects actually occur. Most of the *Post* defects are detected when they occur or at most a handful of cycles afterward.

To handle *Post* defects, we must understand the extent of the damage they cause. This is shown in Figure 9. *Local* are defects that affect only a very localized module. An example is the corruption of a threshold register in the power manager module. The other defects affect larger parts of the system: *Pipeline* affect the pipeline; *Cache* can additionally affect the caches; *Mem* can also affect the memory; finally, *I/O* can additionally affect the I/O subsystem. The Figure shows that most *Post* defects affect caches and/or memory. *Local* and *Pipeline* only account for 7% of the defects.

3. Phoenix: An Architecture for Defect Detection and Recovery

3.1. Proposed Solution

Based on the data analyzed, we propose to handle processor design defects by including novel field-programmable hardware called Phoenix in the processor chip. The goal is to patch buggy hardware in the field as we do today for software. Ideally, when a processor vendor discovers a new design defect, it will release a patch that chips in the field use to re-program their Phoenix hardware. Thereafter, every time that the defect is (or is about to be) exercised, Phoenix will detect it and hopefully avert it or help recover from it. Phoenix can be re-programmed multiple times.

Specifically, we program Phoenix to: (i) tap all the control signals that participate in triggering Concurrent bugs, and (ii) flag when the signals take the combinations of values that exercise Concurrent bugs.

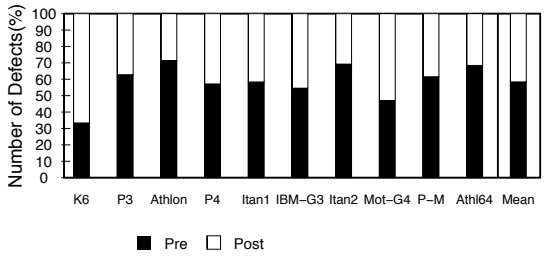


Figure 8. Classification of the Concurrent defects based on when they can be detected.

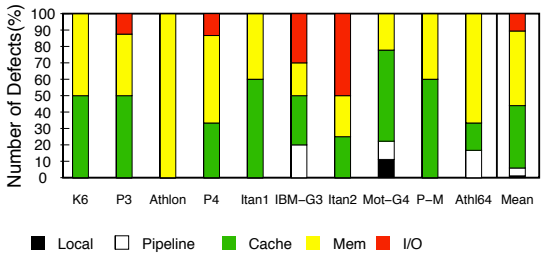


Figure 9. Classification of the *Post* defects based on the extent of the damage caused.

On detection of one such combination, the pipeline is flushed and a supervisor recovery handler is invoked with low overhead. The recovery action depends on the type of defect. If the defect is *Pre*, it has not been exercised yet. Phoenix averts the defect by having flushed the pipeline and, possibly, by temporarily disabling one of the signals that contributes to the defect. For example, in *Defect1* of Table 2, the power manager is temporarily turned off while L2 is being flushed. As the pipeline refills, the defect will very likely be averted. The reason is that these defects require a very specific interleaving of events, and flushing and re-filling the pipeline (with possibly a disabled signal) changes this interleaving. Indeed, pipeline flushing is used by the Pentium Pro to eliminate corner cases after manipulating control registers [14], and was used by the IBM 3081 to work around design bugs.

While we expect this to suffice, if the defect re-occurs, Phoenix flushes the pipeline again and emulates the subsequent few instructions in software. In this case, event interleaving changes considerably.

If the defect is *Local Post* or *Pipeline Post*, the defect has affected a limited section of the processor. Consequently, we follow the same procedure as in *Pre* defects. In the case of *Local Post*, the corrupted module (e.g., the power manager module) is also reset.

For the rest of *Post* defects, a pipeline flush is insufficient, since caches, memory, or I/O may already be corrupted. In this case, a recovery handler customized to that defect is executed (if provided by the vendor) or an exception is passed to the OS. Some form of recovery may be possible because the defect has *just occurred*, and corruption may not have propagated widely. A more generic (and expensive) recovery approach would be to rely on a checkpointing scheme to roll back execution to a prior checkpoint. The scope of the rollback would depend on the type of defect (Figure 9). *Cache* defects require cache state rollback, like in thread-level speculation systems; *Mem* defects require memory rollback, such as e.g., [25, 26]; *I/O* defects need I/O rollback, such as e.g., [23]. How to tune these schemes for the very short detection latency and low frequency of *Post* defects is beyond our scope.

Overall, by tapping the necessary signals, we argue that Phoenix can detect all Concurrent defects (69% of Critical). Moreover, by flushing the pipeline and possibly disabling a signal or emulating instructions in software, Phoenix can recover from the Concurrent *Pre*, *Local Post*, and *Pipeline Post* ones (63% of Concurrent). For the rest of Concurrent defects, Phoenix may simplify recovery by immediately invoking a customized recovery handler. Section 3.3 discusses what it would take for Phoenix to additionally detect the remaining Critical defects, namely the Complex ones.

Note that Phoenix's coverage of design defects is lower than the coverage of hard or soft faults by certain reliability schemes — often over ninety five percent. However, given the very high cost of design defect debugging, we claim that Phoenix is very cost-effective (Section 4).

3.2. Detailed Design

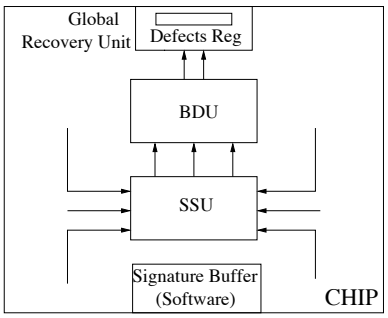
Conceptually, Phoenix consists of four units (Figure 10(a)). The *Signature Buffer* is a software structure that contains information to program Phoenix. The *Signal Selection Unit (SSU)* is programmable hardware that selects the logic signals to be monitored. The *Bug Detection Unit (BDU)* is programmable hardware that combines the selected signals into the logic expressions that flag when defects occur. Finally, the *Global Recovery Unit* takes the outputs of the BDU and, if any is asserted, initiates recovery.

Since centralized SSU and BDU units are not scalable, we develop a distributed design. We logically divide the chip into several *Subsystems*, such as the fetch unit, data cache, or memory controller, and assign one SSU and one BDU to each of them. Then, rather than directly connecting all SSUs to all BDUs, we logically divide the chip into *Neighborhoods* of several subsystems each. Each neighborhood has one *Hub*, which collects signals from the neighborhood's SSUs and passes them to other hubs, and brings in signals from other hubs into the neighborhood's BDUs.

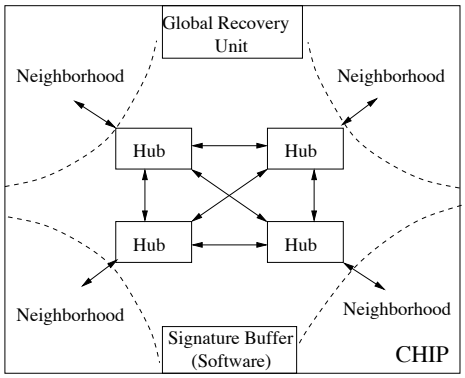
Figure 10(b) shows the chip with one hub per neighborhood. Neighborhoods are selected based on the chip floor-plan. They contain subsystems that are both physically close and functionally related. As an example, Figure 10(c) lists the four neighborhoods used for the Pentium 4. Figure 10(d) shows one neighborhood. It has one SSU-BDU pair per subsystem, each one supplying signals to and taking signals from the neighborhood hub. Figure 10(e) lists all the subsystems that we consider, although not all chips have all the subsystems. We now describe each Phoenix component in turn.

Signature Buffer. This is a software structure where a supervisor process stores a Defect Signature downloaded from the processor vendor. A signature is a bit-vector with as many bits as there are programmable transistors in the Phoenix hardware. The OS reads the signature and writes it to memory-mapped locations that re-program the SSUs, hubs, and BDUs. If more defects are discovered, an augmented signature is downloaded and used to re-program Phoenix. From our evaluation of the hardware in Section 5, we estimate that the size of a signature needs to be no larger than 1Kbyte.

Signal Selection Unit (SSU). The SSU is a field-programmable switch (Figure 11(a)). Its inputs are all the control signals generated by the subsystem that, in the designers' judgment, could possibly help



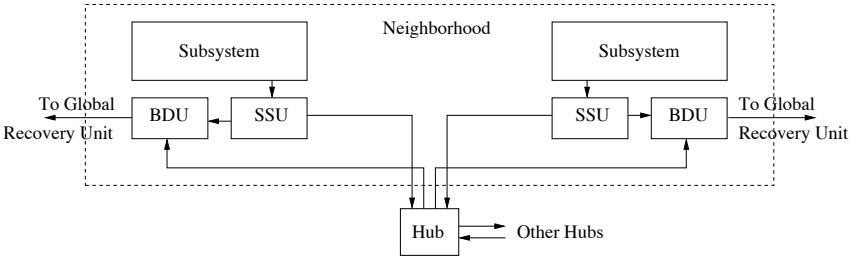
(a) Conceptual Design



(b) Implementable Design

1: L1 Data Cache, L2 Cache, I/O, Virtual Memory, Fetch Unit, MP & Bus
2: FP ALU, Instruction Cache, Decode Unit
3: Scheduler, Interrupt, Status, Ld/St Queue, Int ALU, Register File
4: System Control (controllers for power, temp, and voltage)

(c) The Four Neighborhoods in the Pentium 4



(d) A Neighborhood

Instruction Cache	FP ALU	Virtual Memory
Fetch Unit	Ld/St Queue	I/O Controller
Decode Unit	L1 Data Cache	Interrupt Handler
Scheduler	L2 Cache	System Control
Register File	L3 Cache	MP and Bus Unit
Integer ALU	Mem. Controller	Processor Status Manager

(e) All the Subsystems Considered

Figure 10. Phoenix design: conceptual (a) and implementable ((b)-(e)).

flag any yet unknown defect condition. For example, for the L2 cache subsystem, they could include the signals for cache hit, access, read or write, snoop, invalidation, lock, flush, unaligned access, and access size. At the intersection of each SSU's input and output line there is a programmable pass transistor. Each output line can be connected to at most one of the inputs.

Hub. A hub is a field-programmable switch that takes signals from the neighborhood SSUs and from the other hubs, and has outputs going to neighborhood BDUs and to other hubs (Figure 11(b)).

Bug Detection Unit (BDU). The BDU is a Field-Programmable Logic Array (FPLA) (Figure 11(c)). Its inputs are some of the outputs of the local SSU (i.e., the SSU in the same subsystem) and some of the outputs of the neighborhood hub. The reason for the latter is that defect conditions sometimes combine signals from multiple subsystems. The BDU outputs are the defect conditions that this particular BDU flags.

Since signals are routed across the chip, they may take varying numbers of cycles to reach their destination BDU. However, a BDU must capture a snapshot of signals corresponding to a given cycle. To accomplish this, we insert delay compensation buffers at every BDU input. They can be programmed to delay a signal for a number of cycles (Figure 11(d)).

Global Recovery Unit. The outputs of the BDUs are routed to the Global Recovery Unit (Figure 10(a)). Each of them is connected to one bit of the *Defects* register. When one of these bits gets set, the hardware automatically flushes the pipeline and jumps to the entry point of the recovery handler.

Recovery Handler. The Phoenix recovery handler is a set of supervisor software routines downloaded from the vendor that are stored in memory. Figure 11(e) shows the recovery handler algorithm. Depending on which bit in the Defects register was set, a different action is taken.

For *Pre* or *Pipeline Post* defects, the handler attempts to further change the processor state by possibly turning off one of the signals contributing to the defect condition. Then, it sets a timer and resumes execution. If the timer expires before the same BDU output is asserted again, the defect has been averted, and the handler turns the corresponding signal on again. Otherwise, the handler is retried from the beginning. In this case, however, the handler emulates in software a set of instructions to completely change the interleaving.

For *Local Post* defects, the handler performs the same operations except that it also resets the corrupted module.

For the rest of *Post* defects, if the appropriate checkpointing scheme is supported, the handler triggers a roll back to a previous checkpoint and resumes execution. Otherwise, it sends an exception to the OS or, if a specialized recovery handler was provided by the vendor, it invokes it.

3.3. Handling Complex Defects

Enhancing Phoenix to also detect Complex defects would have to address two issues. First, the triggers for these defects are described as sequences of events (*Defect3* of Table 2 is a simple example). Consequently, Phoenix would have to include hardware to capture the sequences, most likely using string matching algorithms. These algorithms typically combine multiple FSMs to match a string. Therefore, Phoenix would have to include FSMs that are programmed dynami-

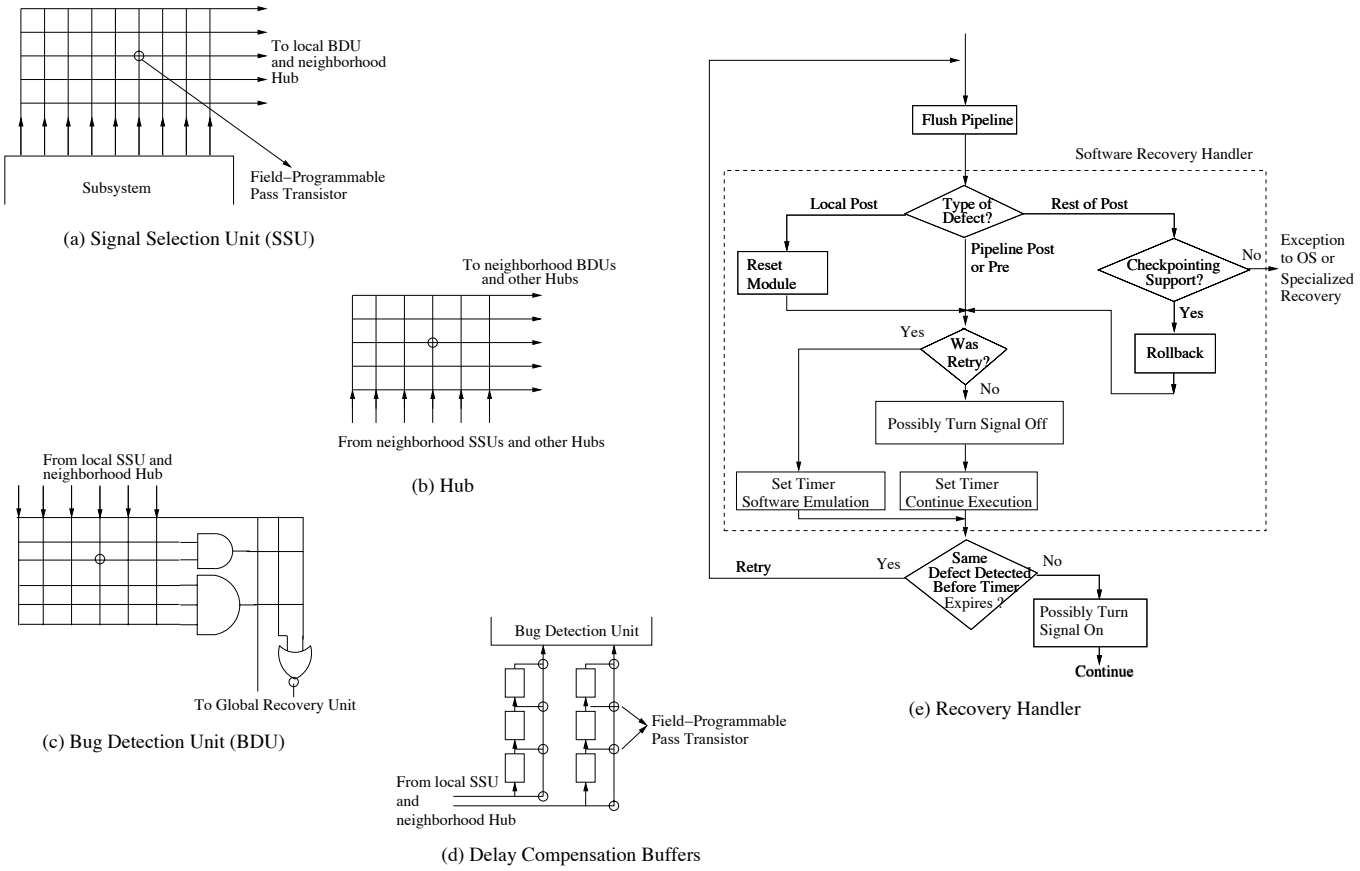


Figure 11. Phoenix components.

cally. This can done by using lookup tables to store the state transitions for the FSMs.

The second, harder problem is the way in which the signals of Complex bugs are specified. They are given as a set of microarchitectural events like in Concurrent defects in only a few cases. In some cases, they are instead given as one event and some unspecified "internal boundary condition". Moreover, in most cases, they are given as obscure internal conditions, often corresponding to RTL-level signals. These signals are both hard to tap and very numerous. For example, one Complex defect starts with a prefetch that brings erroneous data into the cache, setting some internal error state but not notifying the OS because the program correctness has not been affected. If the data is later read, no error is reported (this constitutes the Complex defect) because it appears that the existing internal error state inhibits any error reporting. This internal error state is given in terms of RTL level-like signals. Overall, handling Complex defects involves an unappealing quantum loss in cost-effectiveness.

3.4. Sizing Phoenix Hardware

As engineers design a new processor, they do not know what design defects it will have and, therefore, how to size its Phoenix hardware. However, it is at this time that they need to choose the signals to tap, and the size and spatial distribution of the Phoenix logic structures, wires, and delay buffers.

To solve this problem, we propose to use the errata documents of the 10 processors analyzed in Section 2 to determine, for these processors, what signals to tap, how to combine them into defect conditions, and how to route them to minimize logic, wires, and buffers. From these "train-set" processors we can extract parameterized rules of the Phoenix hardware required, which we can then apply to new processors. In Section 5, we apply the rules to 5 completely different processors.

To understand the procedure, we note that Phoenix taps two types of signals: Generic and Specific. *Generic* signals are microarchitectural events largely common to all processors, such as cache miss, bus transaction, or interrupt. In practice, their number is largely bounded by very low hundreds. *Specific* signals are specific to a processor, such as hyperthreading enabled or thermal duty cycle settings. They are special pins or registers in the processor, and can be compiled from the processor's manual.

From the train-set processors, we generate two pieces of information. First, we compile the list of Generic signals that participate in Concurrent defects in any of our 10 processors. This list has ≈150 signals. Second, we generate rules of thumb that, given the number of signals tapped from a subsystem or from a neighborhood, provide the recommended sizes of the Phoenix logic structures, wires, and buffers for the subsystem or neighborhood, respectively. We generate these rules by generating scatter plots of the hardware needed by all subsystems and neighborhoods of the train-set processors and taking the envelope lines of the plots. The rules for our 10 processors are shown in Table 4, where s is the number of signals tapped in a subsystem, and n_A and n_B are the total number of signals tapped in neighborhoods A and B, respectively.

With this information, Table 5 shows the 3-step *Phoenix* algorithm, used to size the Phoenix hardware for a new processor. In the

Wires from local SSU to neighborhood Hub: $s/4$
Wires from local SSU to local BDU: $s/4$
Wires from neighborhood Hub to local BDU: $s/3$
Local BDU outputs: $s/3$
Wires from Hub in neighborhood A to Hub in neighborhood B: $(n_A + n_B)/20$
One-bit buffers in each BDU input: Maximum number of cycles to traverse the chip

Table 4. Rules of thumb to size the Phoenix hardware. They are based on our 10-processor train set.

Phoenix Algorithm
1. Generate the list of signals to tap
1.1. Take the list of Generic signals from train set
1.2. Compile the list of Specific signals from manual
1.3. Combine the two lists
2. Place the SSU–BDU pair of each subsystem, group subsystems into 3 – 6 neighborhoods, place the hub of each neighborhood
3. Size the Phoenix logic structures, wires, and buffers based on the rules of thumb

Table 5. *Phoenix* algorithm to size the Phoenix hardware for a new processor.

first step, we generate the list of signals that Phoenix needs to tap. They are the sum of the Generic signals (obtained from the train-set processors) and the Specific ones for this processor (taken from the processor's manual).

In the second step, we decide the locations of the SSU-BDU pairs and hubs in the chip floor-plan[1]. For this, we identify the subsystems of Figure 10(e) in the chip floor-plan. For each subsystem, we place its SSU-BDU pair at a boundary of the subsystem, close to adjoining subsystems. Then, we apply the k-means clustering algorithm to automatically group subsystems into 3-6 neighborhoods. Typically, subsystems that are related to each other are already laid out close by and, therefore, end up in the same neighborhood. In practice, however, what subsystems end up in what neighborhoods is *not* very important. In each neighborhood, we place a hub approximately in the centroid of all the SSU-BDU pairs.

Finally, in the third step, we use the rules of thumb of Table 4 and the number of tapped signals per subsystem and neighborhood (from steps one and two) to size and lay out the Phoenix logic structures, wires, and buffers.

4. Usefulness of Phoenix

This section addresses several questions regarding the usefulness of Phoenix.

Why We Believe that Phoenix Detects All Concurrent Defects. Phoenix uses control signal conditions to detect defects. The reason why we believe this approach works is the way errata documents are written. First, since the defects are meant for system designers, they are given as conditions that designers can manipulate — generic microarchitectural events such as "cache miss" or processor-specific ones such as "pin A20 asserted". Second, for a vast community of system software writers and other equipment manufacturers to avert the bug with a modest amount of effort, the conditions need to be simple. In fact, they are likely to be a superset of the real conditions. Consequently, if the defect is exercised, these simple signal conditions must be asserted. Concurrent defects are easier to detect because there is a time window when all participating signals are asserted.

There is evidence from another scheme [29] that capturing high-level signal conditions is effective. Such scheme uses signal conditions to activate a hardware module that breakpoints the processor (Section 6).

Why We Believe that Phoenix Recovers from the Concurrent *Pre*, *Local Post*, and *Pipeline Post* Defects. Design defects are subtle by definition and, therefore, exercised only with a very specific interleaving of events. With a different interleaving, the defect disappears. Flushing and re-filling the pipeline (especially if one participating signal can be disabled) is known to change event interleaving [14]. In the worst case, Phoenix emulates a set of instructions in software in the re-execution, which completely changes the interleaving. Therefore, this approach is effective to avert *Pre*, *Local Post* and *Pipeline Post* defects.

What Fraction of the Manufacturer Workarounds Is Still Needed. Each workaround corresponds to a single defect. Consequently, for each defect that Phoenix recovers from, there is one less workaround needed — although, as shown in Figure 7, over 40% of the defects have no known workaround to start with. Based on Phoenix's coverage, about 69 x 63 = 43% of the workarounds are unneeded. If, with specialized handlers, Phoenix can additionally recover from the remaining *Post* defects, then it could eliminate up to about 69% of the workarounds.

This has a very significant impact, as workarounds are very expensive. Moreover, they are quite *different* than Phoenix recovery handlers. First, hardware workarounds typically impair the system performance, while Phoenix recovery handlers do not. Second, software workarounds are likely to be OS-specific and, therefore, require multiple versions, while Phoenix handlers are not.

Why Existing Techniques such as Patchable Microcode or Opcode Traps Do Not Work. Popular past and current processors support microcode patching or opcode traps — e.g., IBM's zSeries [12] and Intel's Pentium 4 [16]. These techniques cannot be used to effectively detect the defects considered here. The reason is that each of our defects occurs when there is a subtle combination of events; such occurrence is not very correlated with the execution of any given instruction opcode. For example, *Defect2* in Table 2 appears when a cache hit occurs at the same time as a snoop, a request to defer the transaction, and a request to re-initialize the bus. It is unproductive to handle this defect by performing additional checks or trapping at every single load and store. Even if we did, it is unclear what change to the instruction microcode could avoid the defect. This is why vendors propose expensive workarounds for these defects.

Why Phoenix's Coverage Justifies its Cost. Phoenix detects 69% of defects and, of those, recovers from 63% and helps recover from the rest. This coverage of design defects is lower than the coverage of hard or soft faults attained by certain reliability schemes — often over ninety five percent. However, we argue that Phoenix is still very cost-effective. The reason is the high cost of debugging design defects. Indeed, while well-known redundancy-based techniques such as ECC or parity handle hard and soft faults well, they do not work for subtle design defects. Only laborious design verification can detect such defects. Those that are detected before releasing silicon may require 2-3

[1] We used this same procedure to place the SSU-BDU pairs and hubs in the train-set processors.

Proc.	# of Neigh.	# of Subsys.	SSU			Hub			BDU			Total Area (% Chip)
			Avg In	Avg Out	Avg, (Max) Pass Trans	Avg In	Avg Out	Avg, (Max) Pass Trans	Avg In	Avg Out	Avg, (Max) Pass Trans	
K6	3	15	10.2	5.1	84.4, (288.0)	22.9	27.2	680.1, (1165.9)	5.9	3.4	32.8, (112.0)	0.11
P3	3	16	15.6	7.8	197.6, (968.0)	37.4	44.3	1717.1, (2654.2)	9.1	5.2	76.8, (376.4)	0.06
Athlon	5	16	11.6	5.8	104.7, (364.5)	24.2	27.3	737.5, (1182.9)	6.8	3.9	40.7, (141.7)	0.03
P4	4	16	16.5	8.2	219.8, (968.0)	36.3	41.8	1902.3, (4647.5)	9.6	5.5	85.5, (376.4)	0.06
Itan1	4	17	10.5	5.3	92.2, (312.5)	24.6	28.3	748.5, (1230.7)	6.1	3.5	35.9, (121.5)	0.01
IBM-G3	4	16	11.6	5.8	115.2, (450.0)	25.6	29.4	855.0, (1640.0)	6.8	3.9	44.8, (175.0)	0.10
Itan2	4	17	11.4	5.7	106.3, (392.0)	26.7	30.7	902.4, (1609.5)	6.7	3.8	41.3, (152.4)	0.01
Mot-G4	6	17	12.2	6.1	121.4, (450.0)	25.9	28.7	814.4, (1429.7)	7.1	4.1	47.2, (175.0)	0.05
P-M	3	16	15.2	7.6	183.8, (684.5)	36.4	43.2	1655.2, (2658.1)	8.9	5.1	71.5, (266.2)	0.06
Athl64	5	17	12.9	6.4	127.6, (420.5)	28.5	32.1	1241.7, (4349.8)	7.5	4.3	49.6, (163.5)	0.03
A.Mean	4.1	16.3	12.8	6.4	135.3, (529.8)	28.8	33.3	1125.4, (2256.8)	7.4	4.3	52.6, (206.0)	0.05

Table 6. Characterization of the logic structures in Phoenix.

person-months each to debug, and may cause costly shipping delays; those that slip into production silicon require performance-impairing or software-intensive workarounds, or risk expensive recalls and security breaches. Either type may require a costly chip re-spin.

Consequently, even with a modest coverage, Phoenix substantially reduces costs. Alternately, Phoenix can enable earlier release of a processor to market, which gives a very significant edge to a company. Overall, therefore, given Phoenix's simplicity, it fully justifies its cost. Moreover, it will become more useful as chips become more complex and as processors change generations faster.

5. Evaluation of Phoenix

In this section, we address three key issues: Phoenix's hardware overhead, Phoenix's execution overhead and, most importantly, Phoenix's defect coverage for new processors.

5.1. Phoenix Hardware Overhead

To estimate Phoenix's hardware overhead, we apply the rules of thumb of Table 4 to each of the 10 processors and size the resulting hardware. While this approach may be a bit conservative because these rules were obtained from these same processors, we will see that the hardware overhead is so small anyway, that this matters very little. In the following, we examine the signal, area, wire, and buffering requirements.

5.1.1. Signals Tapped

Figure 12 shows the number of tapped signals, grouped according to the modules in Figure 2. There is a bar for each processor, which includes both its Specific signals and the Generic ones for the subsystems it has. The leftmost bar shows all the Generic signals, most of which are included in all processors. Overall, Phoenix taps 150-270 signals, of which around 150 are Generic. The processors with the most taps are the most complex ones, namely the Pentiums and the Athlon-64. Importantly, most of the taps come from the core's periphery.

5.1.2. Area Required

Table 6 characterizes the sizes of SSUs, hubs, and BDUs. Columns 2 and 3 list the number of neighborhoods and subsystems per chip. The values are 3-6 and 15-17, respectively. The subsequent columns show the average number of inputs, outputs, and pass transistors for the SSUs, hubs and BDUs. In parenthesis, we show the size of the largest such structures in the chip in number of pass transistors. We can see that these structures are small. Averaged across all processors, an SSU has 12.8 inputs and 6.4 outputs, a hub has 28.8 inputs and 33.3 outputs, and a BDU has 7.4 inputs and 4.3 outputs. The table also shows that even the largest hubs only have a few thousand pass transistors.

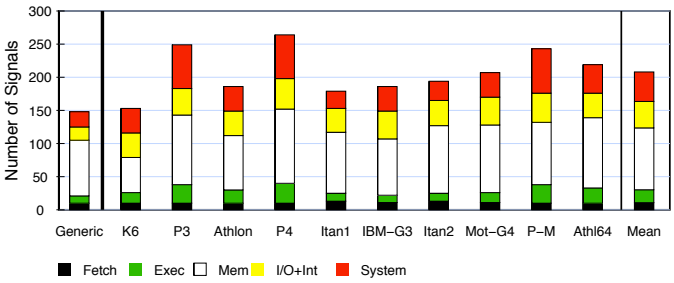


Figure 12. Classification of the signals tapped by Phoenix.

To estimate the area and delay of these structures, we use data from Khatri *et al.* [19]. Their PLA area estimates include the related overhead. The last column of Table 6 combines the area of all these structures for each processor, and shows the result as a fraction of the chip area. Overall, the area required is a negligible 0.05%. For the delay, we use Khatri *et al.*'s worst-case model and scale it for the appropriate technology using [6]. We find that the delays inside our structures are typically a fraction of a processor's cycle, and rarely go up to 2 cycles for the larger structures.

To size the logic to include in the BDUs, we examine all the defect-flagging logic functions. We generate them using sum-of-products logic. Figures 13 and 14 show the distribution of the number of minterms per defect-flagging function, and the number of inputs per minterm, respectively. We can see that over 90% of the functions only need a single minterm of 1-3 inputs. Considering that a BDU has on average 4.3 outputs (Table 6), we conclude that the amount of programmable logic in a BDU is very modest.

5.1.3. Wires Needed

Table 7 shows the number of wires added by Phoenix. Column 2 shows those added between neighborhoods, while Column 3 shows those added inside neighborhoods. The former are global wires; the latter can be considered intermediate-level [18]. On average, Phoenix adds 58 global wires and 115 wires within neighborhoods.

To assess the wiring impact of Phoenix, we use Rent's rule [30], which relates the number of wires connected to a structure with the

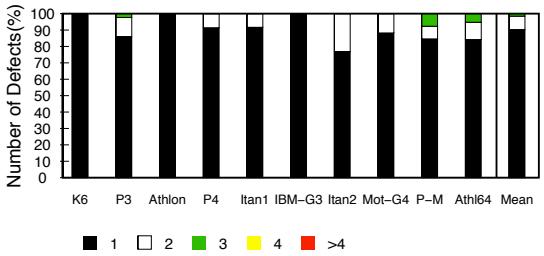


Figure 13. Distribution of the number of minterms per defect-flagging function.

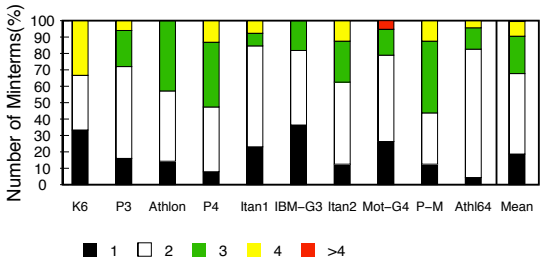


Figure 14. Distribution of the number of inputs per minterm.

number of transistors in the structure. The rule is $T = k \times N^p$, where T is the number of wires, N is the number of transistors, and k and p are constants. For each unmodified processor, we use Rent's rule to estimate the number of wires connected to each subsystem. We use the values of k and p given in Table II of [30]. Then, we compute the increase in the number of wires due to Phoenix (Column 4 of Table 7). The average increase is a very modest 0.48%.

To gain further insight, Columns 5 and 6 show the subsystem that produces the most signals for other subsystems and the one that consumes the most signals from other subsystems, respectively. Such subsystems are typically in the core's periphery, especially in the memory hierarchy. They tend to be the virtual memory, L2 cache, system control (i.e., power, temperature, or voltage control), interrupt handler, and processor status manager subsystems.

5.1.4. Compensation Buffers

To capture a snapshot of signals at a given clock cycle, Phoenix adds programmable 1-bit buffers to all BDU inputs. Given all the signals that contribute to a given BDU output, the one (i_{far}) that is routed from the farthest SSU should have all its delay buffers disabled; any other input should have as many buffers enabled as the difference in arrival times in cycles between i_{far} and its signal.

Using the floor-plan of each chip, we compute the worst-case latency of a Phoenix signal from when it is tapped by an SSU until it arrives at a BDU. For this computation, we use [19] to estimate the latency of crossing SSUs and hubs, and the ITRS [18] to estimate wire delays. Column 2 of Table 8 shows that this number ranges from 1 to 6 cycles[2]. This is the number of 1-bit buffers we add to every BDU input. Column 3 shows the resulting total number of 1-bit buffers added to the chip. This number represents very little storage.

Proc.	Number of Wires		Increase in # of Wires (%)	Highest Producer Subsystem (# Signals)	Highest Consumer Subsystem (# Signals)
	Total Between Neigh	Total Inside Neigh			
K6	28	85	0.80	L1 (2)	vmem (4)
P3	46	139	0.62	vmem (15)	vmem (22)
Athlon	66	103	0.48	vmem (3)	system (5)
P4	76	147	0.86	status (11)	L2 (14)
Itan1	48	98	0.15	L2 (5)	L2 (6)
IBM-G3	52	102	0.54	L2 (6)	system (7)
Itan2	54	106	0.18	interr (6)	interr (9)
Mot-G4	90	113	0.49	vmem (8)	mp&bus (7)
P-M	46	136	0.41	status (5)	interr (10)
Athl64	78	121	0.29	system (6)	memcontr (10)
A.Mean	58.4	115.0	0.48	–	–

Table 7. Characterization of the wiring required by Phoenix.

Proc.	Max Signal Latency in Cycles	Total 1-bit Buffers per Chip
K6	1	89
P3	2	290
Athlon	2	216
P4	6	924
Itan1	2	208
IBM-G3	1	108
Itan2	4	452
Mot-G4	2	240
P-M	5	705
Athl64	4	508

Table 8. Characterization of the buffers added by Phoenix.

5.2. Phoenix Execution Overhead

Phoenix has negligible execution overhead. To see why, consider first issues unrelated to exercising defects. Phoenix increases wire load by tapping signals. However, if designers see a benefit to Phoenix, they will make slight modifications to the circuits to ensure such load increase has no performance impact. Moreover, while large programmable logic is slow, we have seen that Phoenix uses very small and, therefore, fast structures (Section 5.1.2).

Even if defects are exercised, the overall overhead is negligible. The typical cost of correcting a *Pre* defect is to flush the pipeline, disable a signal, and set a timer. Even if we have one error every million cycles, the overhead is noise. In practice, defect activation is rarer — otherwise designers would have detected and fixed the defects.

5.3. Phoenix Defect Coverage for New Processors

We now consider the case where designers are building a new processor and want to use our algorithm of Table 5 to size Phoenix. A key question is what defect coverage should they expect? To answer this question, we consider a "test-set" group of five additional processors from *domains different* than the ten in the train set (Table 9). They include embedded, network, and multicore processors. To these processors, we apply the algorithm of Table 5.

After the Phoenix hardware is completely laid out for these five processors, we examine their errata documents for the first time, and try to program all their Concurrent defects into their Phoenix hardware. We are interested in the *Detection Coverage* and the *Recovery Coverage*. The former is the fraction of Critical defects in the pro-

[2]Our distinction between *Pre* and *Post* defects took into account these cycles.

Processor	Characteristic	Freq (MHz)
Ultra Sparc II	Advanced embedded processor	450
Intel IXP 1200	Network processor	232
Intel PXA 270	Advanced embedded processor	520
IBM PPC 970	Core for the IBM Power 4	2500
Pentium D	Chip multiprocessor	3400

Table 9. New processors analyzed.

cessor that are Concurrent *and* that we can program into the Phoenix hardware; the latter is the fraction of Critical defects that are Concurrent *Pre*, *Local Post*, or *Pipeline Post and* that we can program into Phoenix. The higher these coverages are, the more useful Phoenix is.

Figure 15 shows the results obtained. Each processor has three bars, from left to right: the number of Critical defects (both Concurrent and Complex) in the errata document (*Original*), the Concurrent defects that we program into the Phoenix hardware (*Detected*), and the Concurrent *Pre*, *Local Post*, or *Pipeline Post* defects that we program (*Recovered*). For each processor, the ratio between the second and first bars is the Detection Coverage, while the ratio between the third and first bars is the Recovery Coverage. In addition, the second and third bars have a number on top of them, which is the height of the bar as a percentage of the height of the same bar with unlimited Phoenix resources — both in terms of signals tapped and size of logic structures.

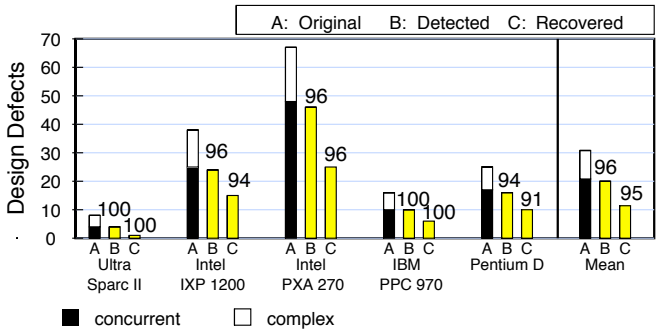


Figure 15. Using Phoenix in five additional processors from different domains.

The high numbers on top of the bars show that our algorithm of Table 5 appropriately sizes the Phoenix hardware on entirely new processors. Only an average of 4-5% of the relevant defects cannot be programmed due to lack of hardware resources.

If we consider the *Mean* bars and take the ratio between the second and first, and between the third and first, we obtain an average Detection Coverage equal to 65%, and an average Recovery Coverage equal to 39%, respectively. These numbers are remarkably similar to those obtained for the 10-processor train set (69% and 43%, respectively). They show that these different processors have an errata profile similar to that of the train set — about two-thirds of the defects are Concurrent and about two-thirds of them are *Pre*, *Local Post*, or *Pipeline Post*. Furthermore, they show the broad applicability and effectiveness of Phoenix for different types of processors.

5.3.1. Sufficient Size of the Train Set

Finally, we compute how many processors are needed in the train set to properly size Phoenix for new processors. We proceed by taking the processors in Table 1 in order and building the train set with the first processor only, then with the first two only, then the first three only, etc. For each train set, we build the corresponding list of Generic signals and rules of thumb as per Section 3.4 and, based on them, size the Phoenix hardware for the five new processors of Table 9. Finally, we attempt to program the defects of the new processors on their Phoenix hardware.

We record how the numbers on top of the two *Mean* bars of Figure 15 change with the train-set size, and show it in Figure 16. The figure shows that, as the size of the train set increases, such numbers increase, both for detection and recovery. Overall, we only need a seven-processor train set: the resulting Phoenix hardware is able to program all the defects that a ten-processor train set would enable to program. Importantly, the processors in the train set do not have to closely resemble the new processor.

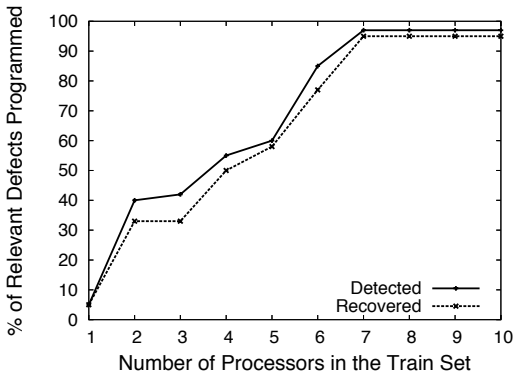


Figure 16. Impact of the train set size on the ability to program defects from the five new processors in Phoenix.

6. Related Work

Existing techniques for permanent defect detection/recovery. Itanium implements many functions in firmware, such as TLB and FP unit control, and interrupt and ECC handling [17]. While execution of such functions is slower, defects can be corrected with patches. Phoenix is more general in that patches can be applied to the whole chip. Moreover, Phoenix has negligible performance cost. Crusoe uses code translation and, therefore, "fixes" design defects by changing the translation software [20]. For the many processors without this support, this approach is not an alternative.

A related approach to fix defects is patchable microcode and opcode traps. This approach was popular in the past and is still used in processors such as IBM's zSeries (Millicode) [12] or Intel's Pentium 4 [16]. As indicated in Section 4, this approach is inappropriate for the design defects considered: a defect is not very correlated with the execution of any given instruction opcode. Trapping or performing additional checks at every instance of a given opcode would be unproductive.

Hardware workarounds proposed by vendors typically impair performance (e.g., chicken switches); Phoenix has negligible performance impact. Some processors have advanced monitoring capabilities such as trigger-based trace arrays or trap-on-event capabilities. For example, Pentium 4 can start tracing branches when an event happens, or trap when an address is accessed. These techniques are too limited to detect the defects detected by Phoenix. Finally, pipeline

flush and retry has been used to recover from design defects in the past (e.g., in IBM 3081).

Research proposals for permanent defect detection/recovery. DIVA adds a checker processor to a pipeline to re-execute instructions as they retire [2]. If the results differ, a fault has been detected. If the checker is assumed defect-free, it could be used to check for permanent defects. However, DIVA is not as suitable as Phoenix for design defects. First, DIVA focuses mostly on checking the pipeline, while most defects are in the core's periphery. Second, many defects cause incorrect side effects (e.g., cache corruption); simply re-executing the instruction will not help. Finally, instructions with I/O or multiprocessor side-effects cannot simply be re-executed.

Vermeulen *et al*. [29] describe a design automation compiler that takes a user's description of when a hardware breakpoint should happen, and generates an RTL breakpoint module. The user specifies the breakpoint in terms of a condition of signals. Such compiler could potentially be usable to generate RTL code for Phoenix modules.

Concurrently to our work, Narayanasamy *et al*. [24] analyze the design errors found in the AMD 64 and Pentium 4 processors. They discuss the applicability of techniques to patch hardware in the field (instruction editing, replay, checkpointing, and hypervisor support).

Analysis of design defects in processors. Avizienis and He [3] examine the Pentium II errata to see if it could be used in high-confidence systems. They propose a taxonomy of design defects. One main conclusion is that 50% of the defects are in the part of the processor that is not devoted to deliver performance, but to handle faults. We perform a much deeper analysis and examine ten more recent and complex processors. While some of our observations agree with theirs, we classify many of the defects in non-performance components as *NonCritical*.

7. Conclusion

Ideally, we would like hardware bugs to be handled like system software ones, with vendors releasing periodic patches. Costly workarounds and expensive chip recalls would be avoided and, conceivably, production silicon could be released to market earlier, giving a significant edge to the company. Toward this vision, this paper made three contributions.

First, we analyzed the design defects of AMD, Intel, IBM, and Motorola processors, to gain insight into how to detect and recover from them. Processors have on average about 30 Critical defects, consistently concentrated in the core's periphery and cache hierarchy.

Second, we proposed Phoenix, novel on-chip field-programmable hardware that detects and recovers from design defects. Phoenix taps key signals and, based on downloaded defect signatures, combines them into conditions that flag defects. On defect detection, Phoenix flushes the pipeline and either retries or invokes a customized recovery handler. We also presented an algorithm to automatically size Phoenix for new processors.

Third, we evaluated Phoenix. Phoenix induces negligible slowdown. Our design taps about 200 signals and adds only 0.05% area and 0.48% wire overheads to the chip. Of all the Critical defects, Phoenix currently detects the 69% that are triggered by concurrent signals (Concurrent). Of these, Phoenix easily recovers from the 63% that are *Pre* or *Local*/*Pipeline Post*, and can simplify recovery for the rest by immediately invoking a customized handler. We argue that this coverage fully justifies Phoenix, given the very high cost of processor defects — extensive pre-release debugging, costly shipping delays, chip re-spins, performance-impairing or software-intensive workarounds, expensive chip recalls and security breaches. Increasing the coverage further requires adding relatively expensive programmable FSMs to Phoenix to detect Complex defects. Lastly, our algorithm effectively sizes Phoenix for new processors of different types.

References

[1] AMD. Technical documentation. *http://www.amd.com*.

[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *International Symposium on Microarchitecture*, pages 196–207, November 1999.

[3] A. Avizienis and Y. He. Microprocessor entomology: A taxonomy of design faults in COTS microprocessors. In *Conference on Dependable Computing for Critical Applications*, November 1999.

[4] F. Bacchini, R. F. Damiano, B. Bentley, K. Baty, K. Normoyle, M. Ishii, and E. Yogev. Verification: What works and what doesn't. In *Design Automation Conference*, page 274, June 2004.

[5] B. Bentley and R. Gray. Validating the Intel Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[6] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[7] A. Carbine and D. Feltham. Pentium Pro processor design for test and debug. *IEEE Design & Test of Computers*, 15(3):77–82, 1998.

[8] A. Gluska. Coverage-oriented verification of Banias. In *Design Automation Conference*, pages 280–285, June 2003.

[9] M. Hachman. Boot-up bug discovered in Intel's desktop Coppermine chips. *http://www.my-esm.com*, December 1999.

[10] M. Hachman. Bug found in Intel's Itanium 2 could cause data loss. *http://www.extremetech.com*, May 2003.

[11] T. R. Halfhill. The truth behind the Pentium bug. *http://www.byte.com*, March 1995.

[12] L. C. Heller and M. S. Farrell. Millicode in an IBM zSeries processor. *IBM Journal of Research and Development*, May 2004.

[13] IBM. 750FX technical documentation. *http://www.ibm.com*.

[14] Intel. Pentium processor family developer's manual, volume 3: Architecture and programming manual. *http://www.intel.com*.

[15] Intel. Technical documentation. *http://www.intel.com*.

[16] Intel. IA-32 Intel Architecture software developer's manual volume 3B: System programming guide, part 2. *http://www.intel.com*, March 2006.

[17] Intel. Intel Itanium processor firmware specifications. *http://www.developer.intel.com/design/itanium/firmware.htm*, 2006.

[18] International Technology Roadmap for Semiconductors (ITRS). *http://www.itrs.net*, 2004.

[19] S. P. Khatri, R. K. Brayton, and A. Sangiovanni. Cross-talk immune VLSI design using a network of PLAs embedded in a regular layout fabric. In *International Conference on Computer Aided Design*, pages 412–418, November 2000.

[20] A. Klaiber. The technology behind Crusoe processors. *http://www.transmeta.com/about/press/white_papers.html*, January 2000.

[21] M. Magee. Intel's hidden Xeon, Pentium 4 bugs. *http://www.theinquirer.net*, August 2002.

[22] Motorola. MPC7457CE technical documentation. *http://www.freescale.com*.

[23] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *International Symposium on High-Performance Computer Architecture*, pages 203–214, February 2006.

[24] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *International Conference on Computer Design*, October 2006.

[25] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, May 2002.

[26] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, May 2002.

[27] Inquirer Staff. AMD Opteron bug can cause incorrect results. *http://www.theinquirer.net*, June 2004.

[28] Inquirer Staff. IBM Power PC 1GHz chip only runs properly at 933MHz. *http://www.theinquirer.net*, August 2004.

[29] B. Vermeulen, M. Z. Urfianto, and S. K. Goel. Automatic generation of breakpoint hardware for silicon debug. In *Design Automation Conference*, June 2004.

[30] P. Zarkesh-Ha, J. A. Davis, and J. D. Meindl. Prediction of net-length distribution for global interconnects in a heterogeneous system-on-a-chip. *IEEE Trans. Very Large Scale Integration Syst.*, 8:649–659, December 2000.