# Chico: An On-Chip Hardware Checker for Pipeline Control Logic

Andrew DeOrio, Adam Bauserman, Valeria Bertacco
Dept. of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
{awdeorio, adambb, valeria}@umich.edu

## ABSTRACT

The widening gap between CPU complexity and verification capability is becoming increasingly more salient. It is impossible to completely verify the functionality of a modern microprocessor before shipping, much less before tape out. Recent studies indicate that the majority of errors in these designs are centered on control and forwarding logic[9]. To address this problem, we present Chico, an efficient approach to on-chip hardware correctness that specifically targets escaped design errors in these high risk functional blocks. Our solution includes an on-chip checker block that monitors the correctness of potential data dependencies and program order of the executed instructions before they are allowed to commit. If this online checker detects a mismatch, the processor's exception handler is invoked, reconfiguring the system to a known-correct, formally-verified mode of operation which can correctly re-execute and commit the faulty instruction. The processor can then resume its normal, high throughput mode of operation. In our experimental setup, we have implemented Chico in an out-of-order processor design and evaluated its performance impact on 11 distinct buggy variants of the design running SPECint benchmarks. Our results indicate that Chico can overcome the errors present in these buggy designs at a minimal performance cost, ranging from less than 1% up to 4%. In addition, we evaluated Chico's area cost and found it to be an order of magnitude smaller than other popular solutions such as DIVA[10], with an area impact of less than 3% for our experimental processor. Our approach is novel in that it shows no appreciable performance degradation on a correct design, and it is a low complexity, area-frugal solution compared to previous work.

## 1. INTRODUCTION

The control logic portion of complex digital systems is notoriously difficult to test and debug. The growing complexity of large out-of-order cores further compounds this issue. For these reasons, the correct design of a pipeline's control logic is such a complex problem that it is often unachievable. As a result, this portion of the design is the culprit for more than half of the escaped bugs reported in modern processors' errata documents[8].

Simulation-based verification is the mainstream approach used in industry to validate a design and correct any design errors. However, due to the complexity of the design state space, only a small fraction of the design's functionality can be validated within the available development time window. Even so, bugs are difficult to diagnose and resolve once identified. Frequently, this requires analyzing extremely long simulation traces in order to identify the root cause of a problem. Formal verification techniques, while capable of checking the correctness of a design aspect under all possible execution situations, can only be deployed for very simple designs.

Recently, a few runtime verification solutions have been proposed by the research community [9, 10, 2]. A common trait of these solutions is the use of one or more on-chip checkers which can detect all or some functional errors by monitoring the processor's execution flow. If an error is detected, most solutions will provide a correction mechanism, which enables the processor to overcome the buggy configuration at a performance cost. Hence, the impact of an error in an aspect protected by an online verification mechanism is limited to a graceful performance degradation, rather than incorrect results or, possibly, a system crash. Thus, a key benefit of these runtime verification solutions is that they enable the verification team to focus its efforts on the design's execution scenarios that arise most frequently, and relieve the burden of striving to fully verify a design before its release to the market.

### 1.1 Contribution of This Work

In this paper we present an on-chip checker solution, called Chico, which specifically targets the most critical control aspects of out-of-order processor designs. Because of Chico's specialized focus, its checking and correction mechanisms require very few hardware resources, resulting in an extremely small area overhead, a full order of magnitude less than some previous approaches such as DIVA[10]. We made the choice to focus exclusively on control logic because of the large fraction of escaped bugs that originate in those blocks, and their critical impact on the system's correct operation.

Chico consists of a checker hardware block that monitors the flow of instructions executed by the processor, and checks that the control aspects of the execution are performed correctly: source value retrieval, data forwarding, branching selection, dynamic execution flow, deadlock, etc. Instructions are checked just before the commit stage, thus avoiding unnecessary expenditure of effort on errors that may appear during speculative operations. If an error is detected, Chico uses the processor's exception handling mechanism to prevent the problematic instruction from completing. It then forces the processor to re-configure into a barebone system by disabling most performance features (including pipelining, data forwarding, branch prediction, etc.) and re-executes the same instruction in this configuration. The barebone version of the design, also called *degraded mode*, is sufficiently simple that it can be formally verified at design time, thus we can guarantee correct completion of the instruction. Once the buggy state has been overcome by degraded mode, the normal mode of operation is resumed along with full-speed execution.

Our solution presents several novel advantages. First of all, Chico requires connections to just a handful of the design's signals, hence it can be easily integrated into a wide range of pipeline architectures. Additionally, our solution is free from false positives, triggering the degraded mode of operation exclusively when a bug arises. In contrast, other solutions described in the literature[9] allow false positives to trade accuracy (and thus performance) for area overhead.

## 2. RELATED WORK

Runtime verification solutions and the use of on-chip checkers have started to appear only in recent years, one of the first works in this domain being DIVA[10]. DIVA is a solution deploying a simple, but complete, processor core next to a complex core. The simple core re-executes and verifies the results of all instructions about to complete in the complex core. Chico differs from DIVA in that it does not need to recompute results and is therefore able to offer a significant reduction in area cost. Based on our experimental results, Chico's area is an order of magnitude smaller than DIVA. DIVA provides greater coverage than our design, but at the cost of significantly higher hardware overhead and prohibitive wire routing challenges, as it must connect to each stage of the complex core. Moreover, the DIVA checker occasionally causes the fast CPU core to stall, incurring a performance penalty even when no error has occurred. Our proposed method has no performance impact during normal operation, only when a faulty instruction is flagged and re-executed by the checker is any degradation realized. Additionally, the significant extra wiring required by DIVA to connect each processor stage to its checker counterpart is eliminated in the Chico solution. Consequently, Chico is not as susceptible to timing, wiring delay and routing problems. All of the this can be achieved with little sacrifice of error coverage, since control logic is the most frequent source of functional bugs. Finally, our Chico checker can be kept extremely simple because we rely on the main processor itself to recover from an error (by reconfiguring it to a barebone system). In contrast, the DIVA checker includes a complete processor implementation, since the checker is also in charge of the recovery.

Another runtime verification solution is Field Repairable Control Logic (FRCL) [9], a recently proposed method used to correct design faults. Similar to our approach, faulty instructions are executed in a degraded mode to correct errors. However, the error detection method is not as automated as with Chico. Instead, the values of control bits in the processor are matched at runtime against programmed "bug patterns." The method is quite flexible, though bugs need to be manually discovered and characterized by a verification engineer before a pattern can be created to correct them. Because FRCL observes only a small number of signals and has limited storage space for bug patterns, error conditions must often be over-specified. This leads to a high incidence of false positives which further increases as more patterns are added. Our method utilizes the same recovery mechanism as FRCL, but relies on a powerful, accurate and automated mechanism for bug detection.

In contrast to our checker which targets a specific class of errors, on-chip assertion processors [6] provide a more general solution, checking arbitrary properties on the chip at runtime. However, assertions must be selected on a per-design basis, thus the use of this solution requires a lot of engineering effort. Our proposed method targets a common class of design bugs, while requiring minimal effort for design time integration.

The use of hardware checkers has found an application supporting design-time formal verification. To this end, Bayazit and Malik have suggested a hybrid strategy [2] that combines hardware checkers with model checking. Their technique would allow an engineer to verify distributed properties too complex or distributed to be verified by a runtime checker, such as cache coherence in a multiprocessor system.

## 3. CHICO OVERVIEW

Chico is an on-chip runtime checker designed to verify correct operation of the forwarding and control logic blocks in out-of-order processors. The checker is embedded in the processor design before instructions are committed. Figure 1 shows a schematic of a generic out-of-order processor design augmented with Chico. The checker adds two extra pipeline stages to the system, a Setup and a Compare stage, in addition to an extra "golden" register file where only validated register values are stored.

Chico is placed between the execute and the commit stages, thus enabling complete recovery of the processor before any architectural state is affected or erroneous results can propagate outside the core. Each instruction passing through the pipeline is checked before it can move on to commit or write its result to memory or the register file. Chico performs a series of checks on each instruction. First, it verifies that the values of each source operand match the last committed value in the golden register file. Additionally, the PC of each instruction is checked against the following instruction's PC to ensure correct program order. Should a major error cause the core to hang (deadlock), failing to commit an instruction within a specified number of clock cycles, a watchdog timer will ensure forward progress. If a failure is detected, an exception is raised and the processor switches to a *degraded mode* which is a fully functional, but barebone version of the system, obtained by disabling all units or features which boost the performance of the system, including pipelining itself. The benefit of this mode of operation is that it is typically simple enough to be formally verified, trading performance for correctness. While in degraded mode, pipelining is disabled and the failed instruction is run through by itself, thus avoiding potentially negative interactions with other instructions that would be in flight at the same time. Because the system's functionality in degraded mode has been completely verified, the instruction will compute the correct result, although extra cycles will be required for it to commit. After the failed instruction passes the checker, the normal mode of operation is re-enabled.

### 3.1 Strengths and Limitations

Chico was designed to overcome and correct errors occurring as a result of functional design bugs in the data forwarding and control logic portions of the processor. Because of its design, it provides the added benefit of protecting the system against transient errors occurring in these units of the core. Chico also protects against permanent transistor

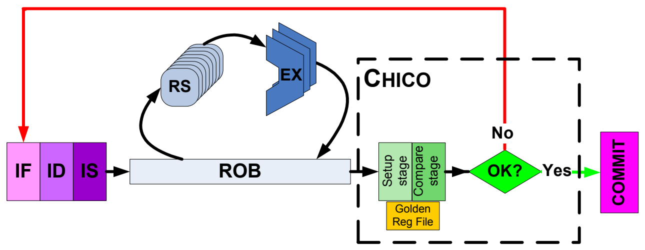

**Figure 1: Chico integrated in an out-of-order pipeline.** Chico checks each instruction before it proceeds to the commit stage. If the check fails, the system switches to degraded mode and re-executes the faulty instruction. The Chico hardware checker consists of two additional pipeline stages, Setup and Compare, and a golden register file to store the verified-correct register values.

failures in the control units, however protection against this latter type of error comes at a high performance cost because the recovery mode would be triggered very often. Finally, Chico can be used to shorten the time between first tapeout and customer release by facilitating the localization of control bugs on test chips between tapeout revisions.

Note, however, that the checker does not recompute the results of any instruction, it simply checks that the control information is correct. This is a consequence of assuming that core datapath components, such as adders, multipliers, logic functional units, etc., are correct. This assumption was made after observing that the verification of the processor's datapath benefits from a more mature methodology, and it is more successful in practice because it can rely on a well-defined specification and on several product generations delivering equivalent functional units. This observation is corroborated by the large majority of escaped bugs that are found in control logic blocks[9]. Because of our assumption we can maintain a very simple and lightweight design for our Chico checker, focusing exclusively on checking control logic activity. Note in addition that the Chico checker is only responsible for checking the correctness of the system's execution, but not recomputing the correct results in case of error – the degraded mode will take care of that. This aspect further simplifies the design of Chico, in fact, it has been argued on several occasions in the literature that checking a result is a simpler task than computing it[2].

The specific set of design properties we rely upon are listed below. Note that all of them are fairly straightforward and can be formally verified even in a complex design, since they involve few sequential elements. Arithmetic units such as multipliers can also be verified by checking their functional equivalence with a simple and correct implementation of the same function.

1. ALUs and other arithmetic units compute the correct result, for any given set of input operands.

2. The datapath blocks are correctly connected, that is, in absence of any dependency between instructions, execution proceeds correctly. This is equivalent to saying that the execution of individual instructions produces the correct result.

3. The memory subsystem works properly, that is, given a (data, address) pair, the memory is accessed at the proper address and the corresponding data is written or retrieved.

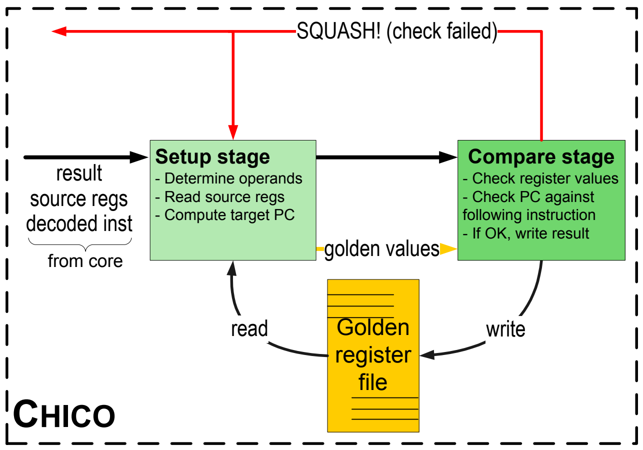

**Figure 2: Components of the Chico checker.** The checker is comprised of two pipeline stages, Setup and Compare, and an additional "golden" register file. The Setup stage reads the golden register file and determines which source registers need to be checked, while the Compare stage compares these values with the actual values. The Figure also shows the signals that are exchanged between the checker blocks.

Chico can also detect and correct permanent and transient faults that occur in covered blocks. Transient faults are dealt with by simply invoking the degraded mode of the system and recovering the instruction that suffered the fault. Permanent faults are different in that they cause errors which continue to manifest themselves even after re-executing the instruction in degraded mode. The risk is that the system may enter an infinite loop in which it keeps re-executing the same instruction with no forward progress. Hence, if an instruction is deemed to be erroneous even after being re-executed, Chico will raise an exception. The rationale behind this design choice was that if we let the instruction commit even when an error was detected after re-execution, the erroneous instruction would write an incorrect result with high probability, and this would undermine the integrity of the system.

## 3.2 Checker Design

Chico is divided into two pipelined stages, Setup and Compare, which are inserted in the pipeline directly before the commit stage. In addition, we inserted a small architected register file, which is read in the Setup stage and written in the Compare stage. Figure 2 shows a schematic of the checker blocks and the signals that they exchange. In this section, we present the architecture of the two stages and we show that this two-stage design is needed to check the correctness of the dynamic program flow.

### Checker Setup Stage

The Setup stage of the checker retrieves the source register values from the checker's private architected register file (the golden register file) using the decoded instruction's register indices. It would have been possible to design the checker so that the committed register state was accessed instead by indexing through a private, golden retirement rename table into the physical register file. Since a rename table would require nearly as much space as an architected register file, we opted for the register file. This approach is also safer, as it does not rely on the correctness of the retirement rename

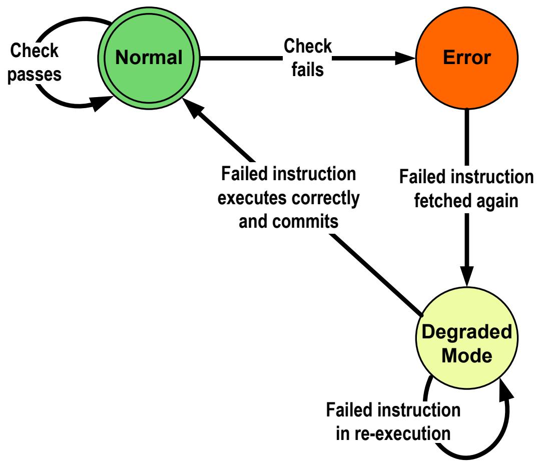
Check passes
Normal
Check fails
Error
Failed instruction executes correctly and commits
Failed instruction fetched again
Degraded Mode
Failed instruction in re-execution

**Figure 3: Pipeline control FSM.** This FSM resides in the Checker Setup Stage and controls the execution flow when a control error is detected in an instruction that is about to commit. It squashes the instructions in flight and takes control of fetching instructions until the problematic instruction has been re-executed.

table. Keeping a separate register file reduces the checker complexity and guards against possible errors.

The Setup stage includes logic to determine which operands must be checked. A multiplexer produces the expected PC of the following instruction, either the branch target (when the instruction is a taken branch) or the next PC (PC+4). The Setup stage is also home to a simple finite state machine (FSM) that controls the execution flow when an error is detected, squashing all instructions in the pipeline, passing the PC of the instruction to be re-run to the fetch stage and disabling any instruction fetching until the problematic instruction has left the out-of-order core (Figure 3).

**Checker Compare Stage**

The second checker stage is responsible for determining if the instruction executed with the correct source register values, if the program counter corresponding to the instruction was the correct one, and finally monitors the occurrence of deadlocks. The values of the source registers are compared against those retrieved in the golden register file. The comparison is only performed on the source registers specified by the instruction being checked. The PC of the instruction currently in the Compare stage is compared against the expected value computed in the previous stage. This verifies that branches have been resolved properly and no instructions have been skipped in the dynamic flow.

Note that we can only check the correctness of the instruction flow by comparing two successive instructions' program counters. We do this by comparing the PCs of the instruction in Setup with the one in Compare. It is possible that the Setup stage could be empty, due to an earlier stall or squash. In this case, we hold on to the instruction in Compare until a new instruction arrives in Setup, then perform the PC check and release the instruction in Compare.

If both the registers and PC checks are successful, the instruction's result is written to the golden register file. Otherwise, a squash signal is asserted, flushing the entire pipeline up to (and including) the instruction in the Compare stage. This signal also causes the pipeline control FSM in the Setup stage to activate the degraded mode (see Figure 3). The PC register in the fetch stage is reset to match that of the faulty instruction and the pipeline control FSM ensures that the instruction passes through the pipeline alone. When the re-executed instruction reaches the Setup stage again, the FSM transitions out of degraded mode, enabling the fetching of the next instruction and resuming normal execution.

## 3.3 Integrating the Checker

Integrating Chico into a pipeline is a relatively straightforward process. With reference to Figure 1, the checker is inserted between the reorder buffer and the commit stage (or between memory and writeback stages in an in-order pipeline). The interconnects in the pipeline must be augmented to pass along source register indices, decoded register and result values, and the PC. Often, this extra interconnect is already available in the processor's pipeline to support instruction replay after non-deterministic latencies, in which case the integration phase does not have any area impact on the design. Finally, the output of the checker indicating a failed assertion must be connected to the pipeline's exception handler. Note that integration into an in-order pipeline requires consideration of data forwarding paths since additional stages are being inserted.

## 4. VERIFICATION METHODOLOGY

The verification of a component designed to verify another component is a challenging task. We took an incremental approach to verifying Chico, with the ultimate goal of integrating it into a complex out-of-order processor. Our testing process was divided into three phases: unit-level testing, testing in a simple in-order pipeline and final integration. In addition, the degraded mode of execution must be formally verified to be fully correct.

For each checker stage, we created a set of directed tests to verify the basic functionality at the module level. The test development was independent of the checker implementation, and based only on its specification, so as to achieve maximum separation between implementation and testing. Most of the errors encountered during this early phase of testing were related to instruction semantics, particularly in determining which registers were actually accessed by each instruction, and hence should be checked by the checker.

In order to test our checker before integrating it into the out-of-order core, we inserted it in a simple 5-stage in-order pipeline. This was valuable in that it helped us work out any implementation-related bugs that might not have been caught with isolated testing. We randomly inverted control signals while running a large set of test programs, comparing the values written back on commit to those produced by a known-good processor model. An unexpected challenge arose here when we realized that adding two stages before memory and writeback would require extra forwarding logic. Conveniently, the checker helped in its own integration by pointing out errors in the new forwarding logic.

Final integration of Chico into the out-of-order processor design was relatively uneventful as a result of the thorough testing carried out beforehand. The pipeline's exception handling logic had to be slightly altered to handle Chico's squash signal, and the system's fetch enable was connected to the checker's pipeline control FSM.

The degraded mode of operation requires little alteration to the processor core into which Chico is being integrated. It is important to note that degraded mode does not require a new simplified version of the design. Many non-essential units, such as branch predictors and speculative execution units can be disabled with a variant of chicken bits, which are common in many design developments. Additional simplification is achieved by virtue of the single instruction execution, only one instruction is allowed in the pipeline at a time when in degraded mode. The whole ISA is verified, one instruction at a time, allowing formal tools to abstract away forwarding, squashing and out-of-order execution logic as well as greatly reducing the fraction of the design involved in each individual property proof. The combination of disabled logic blocks and unused logic due to single instruction execution make the degraded mode simple enough to be verified by traditional formal tools.

## 5. EXPERIMENTAL EVALUATION

We implemented and integrated the Chico checker in a out-of-order pipeline design, and evaluated its error-detection qualities by creating eleven variants of the processor, each with a different design error. All the errors were inspired by the bugs reported in errata documents of current processors in the market today. We measured the performance impact associated with using Chico by running three different types of testbenches on these design variants: pseudo-random testing, targeted direct test cases and the SPECint benchmark suite. These three types of tests enabled us to evaluate the performance of Chico in scenarios ranging from real-world applications to targeted extreme execution flows. In addition, we synthesized our design to estimate Chico's area impact.

A number of architectural parameters may impact the performance penalty of a chip utilizing our checker. Since the recovery mechanism invokes the built-in exception handler, squashing all instructions in flight in an out-of-order processor, the penalty due to a recovered error may vary. Pipeline depth is the most significant factor in determining this penalty. Dependencies between instructions, execution latencies (especially for loads and stores) and branch prediction accuracy all have an effect on the performance impact. We quantified the penalty for varying error rates using an RTL simulation of a moderately sized out-of-order processor. The out-of-order pipeline design used in our tests implements early branch resolution, meaning that branches are resolved in the core rather than on commit. Because of this, the penalty for mispredicted branches is not affected by the addition of Chico's two checker stages.

| Issue / retire capacity | single issue/retire |
|---:|---|
| Physical Registers | 64 |
| ROB entries | 32 |
| Reservation stations | 16 |
| Memory access | 4 load units |
| Memory latency | 8 cycles |
| Caches | 128 line I-cache and D-cache |
| Branch prediction | 64 entry hashing, 2-bit counter |

**Table 1:** Characteristics of the out-of-order testbench processor in which we integrated Chico.

### 5.1 Experimental Setup

Chico was developed in Verilog HDL and integrated into a 64-bit out-of-order pipeline which implements a subset of the Alpha ISA. The subset includes about 95% of the integer operations and all of the branching instructions. Floating point, byte manipulation and multimedia instructions were excluded. We made use of three different experimental setups to execute programs in the processor, a standard fetch for running targeted assembly test cases, a connection to a constrained random stimulus generator, and a parallel lock-step connection with an architectural simulator. The architectural simulator allowed us to simulate SPECint benchmark by emulating the instructions not supported by our ISA. The features of the out-of-order processor are shown in Table 1. Note that while here we evaluate Chico on a single issue/retire design, the checker could easily be extended to multiple issue/retire by adding a dependency check among instructions retired simultaneously.

In order to evaluate the bug-detection capabilities of our checker, we made use of a set of buggy processor designs implemented in Verilog. The bugs were created to mimic escaped bugs found in errata documents for modern processors, including x86, PowerPC and ARM[3, 4, 5]. The Verilog model of our out-of-order processor was manually modified, inserting specific bugs as necessary. These bugs are described in Table 2.

| Bug | Description |
|---|---|
| baseline | no bugs |
| two-stores | two consecutive stores cause incorrect address computation |
| two-branches | two consecutive branches corrupt program order |
| store-dep | store followed by a dependent instruction fails |
| mult-branch | multiply followed by a branch causes branch to resolve incorrectly |
| load-branch | conditional branch depending on a preceding load fails |
| rob-full | full reorder buffer causes disruption in program flow |
| rs-full | all reservation stations full causes a missed instruction |
| dep-instrs | two consecutive dependent instructions fail |
| zero-reg-CDB | forwarding from zero reg fails with simultaneous CDB broadcast |
| write-zero-reg | write to zero reg causes non zero values to be read |
| regA-regB-fwd | simultaneous regA and regB forwarding causes incorrect reg value |

**Table 2:** Bugs introduced into the out-of-order core

### 5.2 Simulated Workloads

We have evaluated the effectiveness and performance of Chico with three types of workloads: constrained random stimulus, directed assembly language test cases and SPECint benchmarks.

In the first setup, instruction streams generated by a closed loop constrained random test generator[8] were feeding the instruction bus of the processor. For this simulation we used our baseline out-of-order processor design without inserted bugs. The generator was configured to exercise the full ISA of our system, stressing in particular dependencies and data forwarding through memory. This gave us additional reassurance that the checker operated as designed.

We also utilized a set of directed tests in assembly language that were developed as part of the verification process during the initial development of the processor design. These test cases are described in Table 3. Each of the testbenches runs for a moderately high number of cycles and stress all the key features of the processor, particularly the control aspects. We found experimentally that this regression suite was capable to expose all the 11 bugs that were included in the design variants. These programs, while more suited to exposing bugs, might give pessimistic estimates of performance compared to real workloads. Because the test cases were designed to maximize the number of in-flight instructions, the penalty incurred on a checker recovery also tends to be maximized.

| Name | Description |
|---|---|
| bubblesort | bubble sort an array of numbers |
| combRec | recursively calculates combinations |
| dude | sorts, divides and square roots arrays |
| fib_rec | computes Fibonacci sequences recursively |
| objsort | sort items in a linked list |
| parsort | comparison sort an array of numbers |
| prime | finds all prime numbers less than X |
| powers | computes large powers |
| series | computes geometric and arithmetic series |

**Table 3:** Targeted test cases written in assembly language

To obtain results reflecting more accurately real-world execution flows, a set of 12 SPECint benchmarks were used in our third experimental setup. These provided us with an accurate estimate of the effect of the checker recovery on CPU performance. Because the benchmarks are prohibitively large, they could not be executed in their entirety using our RTL model. To derive a shorter trace that would be representative of each SPECint benchmark, we used SimPoint. The traces generated by SimPoint are derived from the actual benchmarks, but each contain less than 10k instructions. In effect, the traces fast-forward through the initialization of each benchmark, including only the core part of the program most indicative of overall performance. This method allowed us to extract realistic performance data while maintaining a practical simulation time. The benchmarks were compiled to target the full Alpha ISA, including a few instructions not handled by our processor. To solve this problem, we ran our model in lockstep with the SimpleScalar[1] simulator. SimpleScalar would simulate the SPECint benchmark and output a dynamic trace file which we fed to our RTL processor design. The trace file was consulted by the processor when it encountered instructions that were not implemented in our design. By doing so, we were able to follow the correct execution path and also receive the proper data from emulated system calls. This experiment provided us with a more realistic evaluation of the frequency of occurrence of escaped bugs during a typical workload execution and, consequently, of the performance penalty incurred by the use of Chico.

## 5.3 Results

When no buggy configuration arises, our method has a negligible impact on processor performance. The additional pipeline stages inserted for the Chico checker increase instruction latency, but this has no significant effect on the instructions-per-cycle (IPC) for large test programs. We purposely designed the processor to achieve this by freeing physical registers immediately after instructions leave the ROB, and by resolving branches during execution.

Table 4 reports our experimental results over the SPECint benchmarks. The first column indicates the testbench name, the second the number of dynamic instructions executed, the third is the relative IPC, that is, the ratio between the instructions-per-cycle when executing the testbench on a buggy design version and the IPC when executing on a correct design version which has not been equipped with Chico. Finally, the last two columns report the total number of recoveries triggered by Chico and the average number of penalty clock cycles incurred during each recovery. Note that the table reports results averaged over all the 11 buggy variants of the design. Figure 4 shows graphically the average number of clock cycles of penalty incurred with each Chico recovery, the same data reported in the last column of Table 4.

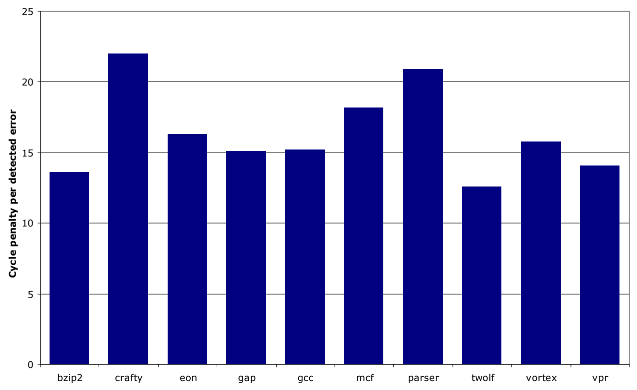

**Figure 4: Average cycle penalty** for each error recovery over all 11 buggy versions of the out-of-order pipeline equipped with Chico, while executing SPECint benchmarks.

When errors are present and caught by the checker, we still observe a fairly small impact on the overall IPC, particularly when running SPECint benchmarks. The slowdown in this case ranged from less than 1% to 4%.

The SPECint benchmarks provide the best workload for determining the checker's performance. In general, these programs exhibited a lower penalty per fault and degradation in the overall IPC (1% - 4%). However, the SPECint suite exposed significantly fewer bugs than the targeted test cases. Three programs, *gzip*, *mcf* and *perlbmk* were unable to expose any of the embedded bugs.

Figure 5 reports the same information as Figure 4 for the directed testbenches suite. It can be noticed that the penalty incurred is much higher in this suite compared to SPECint. The reason lies in the high density of in-flight instructions, which was achieved by developing this suite directly in as-

| Benchmark | #Instr | rel IPC | Recoveries | Penalty |
|---|---|---|---|---|
| bzip2 | 8613 | 0.967 | 77 | 13.6 |
| crafty | 7061 | 0.996 | 33 | 22.0 |
| eon | 6634 | 0.987 | 60 | 16.3 |
| gap | 7071 | 0.976 | 118 | 15.1 |
| gcc | 6534 | 0.975 | 37 | 15.2 |
| gzip | 5985 | 1 | 0 | N/A |
| mcf | 4771 | 1 | 0 | N/A |
| parser | 6148 | 0.987 | 76 | 20.9 |
| perlbmk | 5137 | 1 | 0 | N/A |
| twolf | 7821 | 0.992 | 48 | 12.6 |
| vortex | 5551 | 0.995 | 25 | 15.8 |
| vpr | 6927 | 0.993 | 39 | 14.1 |

**Table 4:** Performance summary for SPECint benchmarks, averaged over all buggy designs, relative to baseline design.

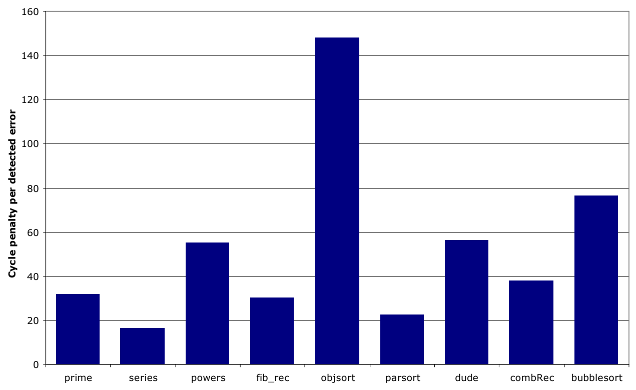

**Figure 5: Average cycle penalty** for each error recovery over all 11 buggy versions of the out-of-order pipeline design equipped with Chico, while executing each of the targeted test cases.

sembly. We also note that this suite was more effective at exposing bugs than the SPEC benchmarks. The *objsort* program is an example of nearly worst-case behavior, with an exceptionally high penalty for checker recovery. In this program, a large number of back-to-back dependent memory operations may be flushed from the pipeline whenever an error is encountered. The overall slowdown with directed test cases is greater than with the SPECint suite, with up to 30% performance impact in the worst case.

Simulation with the random stimulus generator reinforced our confidence in Chico's ability to differentiate bugs from correct operation. We ran nearly 12 million dynamic instructions using this setup, finding that Chico flagged no false positives. Interestingly, Chico exposed a few escaped bugs in our processor core which had not been caught during verification when it was originally designed by a team of graduate students. The results from our direct random stimulus tests reassured us that Chico was free from false positives and operated as designed.

## 5.4 Area Cost

We estimated the area overhead of our Chico implementation by synthesizing the experimental processor design with and without the inclusion of Chico. To this end, we used Synopsys Design Compiler targeting a 90nm TSMC library. Design Compiler was configured with timing as the primary objective and area as the second; a basic wire delay model from the library was used. The resulting checker area was 0.065mm$^2$, while the processor core occupied 2.291mm$^2$; thus Chico comprises 2.8% of the total area. Note that 2.8% is a very conservative area penalty due to the small size of the processor used for experimentation; a larger design, for example one that includes a floating point unit, a load-store queue, or other features, would further reduce Chico's relative size. Chico's total area can be broken down into its three components, with 5% comprised of the Setup stage, 2% of the Compare stage and 93% of the architected register file.

For comparison, we used the same methodology to estimate the area of a DIVA[10] solution in 90nm technology. Our DIVA implementation was a simple, single-issue 5 stage pipeline with a 0.5KB instruction cache and a 4KB data cache very similar to the one described in [10]. We synthesized the DIVA processor core with the same 90nm TSMC library as was used for Chico and approximated the cache areas with Cacti 4.2[7]. Note that this is a conservative estimate, as it is only the area of the DIVA checker processor itself and does not include any of the extremely complex interconnects necessary to connect each stage of the checker processor to its corresponding stage in the processor core. Our estimated area for a single-issue DIVA processor is 0.673mm$^2$, about 10 times the area of Chico.

## 6. CONCLUSIONS

We have presented Chico, an efficient approach to on-chip runtime verification that specifically targets functional errors in the control logic blocks of a processor. The area penalty of our Chico implementation is 10 times less than an approach based on a full checker processor. By taking advantage of a formally verified degraded mode, our solution recovers from functional errors in exchange for a small performance impact. Our solution is novel in that it targets a critical class of design bugs and optimizes its design based on the characteristics of this class of bugs. The error detection logic does not trigger false positives, resulting in zero performance degradation when no bugs are encountered. We show an average performance penalty ranging from less than 1% to 4% for the SPECint benchmarks running on a range of buggy processor designs. Our checker is effective in converting correctness concerns into performance issues, and it supports the verification team in managing the complexity of large out-of-order CPUs, increasing reliability and easing the verification process.

## 7. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[2] A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *International Conference on Computer-Aided Design*, pages 1052–1059, November 2005.

[3] DDJ Microprocessor Center. http://www.x86.org/.

[4] IBM Corporation. IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice, July 2005.

[5] Intel Corporation. Intel(R) StrongARM(R) SA-1100 Microprocessor Specification Update, February 2000.

[6] J. Nacif, F. de Paula, H. Foster, C. Coelho, and A. Fernandes. The Chip is Ready. Am I done? On-chip Verification using Assertion Processors. In *Symposium on Integrated Circuits and System Design*, pages 55–59, September 2004.

[7] D. Tarjan, S. Thoziyoor, and N. Jouppi. Cacti 4.0, June 2006.

[8] I. Wagner, V. Bertacco, and T. Austin. Stresstest: An automatic approach to test generation via activity monitors. In *Design Automation Conference*, June 2005.

[9] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field repairable control logic. In *Design Automation Conference*, pages 344–347, July 2006.

[10] C. Weaver, K. C. Barr, E. D. Marsman, D. Ernst, and T. Austin. Performance analysis using pipeline visualization. In *IEEE ISPASS*, pages 59–67, June 2001.