Releasing Efficient Beta Cores to Market Early

Sangeetha Sudhakrishnan, Rigo Dicochea, and Jose Renau
Dept. of Computer Engineering, University of California Santa Cruz
sangeetha, rigo, renau@soe.ucsc.edu
http://masc.soe.ucsc.edu

ABSTRACT

Verification of modern processors is an expensive, time consuming, and challenging task. Although it is estimated that over half of total design time is spent on verification, we often find processors with bugs released into the market. This paper proposes an architecture that tolerates, not just the typically infrequent bugs found in current processors, but a significantly larger set of bugs. The objective is to allow for a much quicker time to market. We propose an architecture built around Beta Cores, which are cores partially verified. Our proposal intelligently activates and deactivates a simple single issue in-order Checker Core to verify a buggy superscalar out-of-order Beta Core.

Our Beta Core Solution (BCS), which includes the Beta Core, the Checker Core, and the logic to detect potentially buggy situations consumes just 5% more power than the stand-alone Beta Core. We also show that performance is only slightly diminished with an average slowdown of 1.6%. By leveraging program signatures, our BCS only needs a simple in-order Checker Core, at half the frequency, to verify a complex 4 issue out-of-order Beta Core. The BCS architecture allows for a decrease in verification effort and thus a quicker time to market.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Reliability, Verification

Keywords

Verification, Microprocessors, Defects

ISCA'11, June 4–8, 2011, San Jose, California, USA.

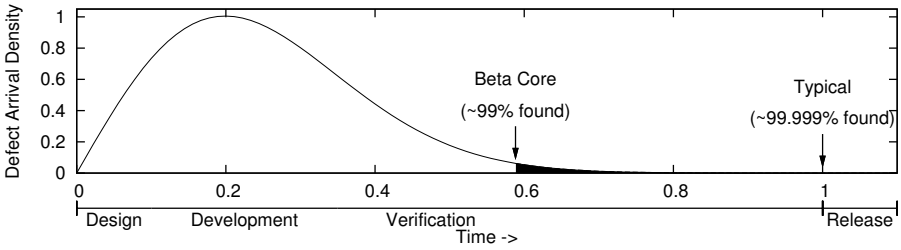
1. INTRODUCTION

Processor design is an extremely difficult and time consuming task. Hundreds of engineers are involved in the design and verification of a new processor. This process typically lasts between 3 and 6 years. Although significant effort is devoted to designing flawless processors, the biggest companies, like Intel and AMD, are unable to release bug free processors. This fact is documented in the numerous erratas [1–5] detailing complex situations that generate incorrect or buggy results. Bugs are inevitable, but tolerating even more bugs can potentially accelerate the time to market for future systems. We go one step beyond current proposals, instead of tolerating just a few bugs, we propose an architecture that efficiently tolerates a partially verified processor. We call this architecture the Beta Core Solution (BCS).

Recent research work [6–9] aimed at tolerating processor bugs can be broadly classified into 2 major categories: always re-execute and a priori bug detection. Always re-execute architectures, re-execute all instructions before they are allowed to commit or retire from the ROB. A complex core is coupled with another core that always checks the results [6, 7]. Always re-execute solutions require a correct Checker Core to verify a faster but potentially buggy core. The general solution to mitigate the Checker Core requirements is to allow the buggy or more complex core to run ahead and provide prefetching information to the Checker Core. Always re-execute architectures provide protection against unknown bugs, but at a high power cost because a non trivial checker is always active.

A priori bug detection systems tackle the issue in a fundamentally different manner. A priori bug detection architectures only re-execute instructions based upon prior knowledge of an existing bug. An a priori bug detection approach taps control signals and creates signatures [8, 9] used to detect previously specified bugs. They create a signature using several control signals from the processor. Their solution hashes most of the processor control signals and compares the results with a predefined "bug database". A signature hit means that the processor can potentially generate incorrect results. This triggers a recovery mechanism that can potentially have several orders of magnitude slowdown. Equally problematic, these systems require a database with all known potential bugs. This process involves the designers, which makes it difficult for systems in which new bugs are constantly detected. Contrary to always re-execute solutions, a priori bug systems have a low overhead, but they can not protect from unknown bugs.

A contribution of this work is an incorporation of the strengths of two techniques, "a priori bug systems" and "always re-execute systems", into a robust unified architecture. Our Beta Core Solution (BCS) is not just an integration, it requires a substantial redesign. An a priori bug detection system does not provide any benefits when integrated with an always re-execute system because the



always re-execute is already able to verify all instructions. BCS uses a unique signature mechanism for re-executing the frequent bugs in the Checker Core, and the infrequent bugs with an a priori system.

BCS provides a novel method to dynamically create signatures. We comprise our signatures of enough critical information to detect bugs but not contain so much information that they become too large. An overly detailed signature will have the Checker Core frequently active, a simple signature, like using program phases, can not capture all bugs. Our evaluation shows that our signature construction satisfies both constraints.

Conceptually, already executed code does not need to be constantly re-verified, our proposed signature system avoids re- checks. Our signature incorporates program phases with processor timing information. The signature is compared to a database of verified signatures. When a new signature is created, the Checker Core verifies the Beta Core's retiring instruction bundle before it is allowed to retire from the Beta Core. If the results from the Beta Core and Checker Core match, the instruction bundle retires and the new signature is added to the good signature database. If we find a signature that already exists, we retire the instructions without calling the Checker Core. If there is a mis-match between cores, the Checker Core flushes the Beta Core and executes several instructions before copying the new context to the Beta Core.

The previous approach works for most bugs, but there could still be potentially difficult to catch bugs that are missed. These infrequent and difficult bugs are captured with a bad database methodology defined in a signature. We have a database of good signatures to detect already verified code and we have a set of bad signatures for known bugs in the processor. The bad signature detection methodology is equivalent to a priori bug detection work.

In addition to the good and bad signature logic, we also propose a new modified register file for the Checker Core so that it can efficiently start for frequent bug phases. The Checker Core's Architectural Register File (ARF) is updated at the same time the ARF in the Beta Core is updated. When instructions retire from the Beta Core, the ARF from the Beta Core and the Checker Core are updated.

BCS is able to utilize a simple in-order Checker Core, as opposed to a complex out-of-order Checker Core, as a result of its low activity. If the Checker Core were active 100% of the time, this would necessitate the use of a complex checker as a means to keep up with it's partner Beta Core. When the Checker Core is inactive the Beta Core executes at full speed without the Checker Core constraining it's performance. The proposed Checker Core offers two major advantages, it is architecturally simplistic and consumes minimal power. As the evaluation shows, a single issue in-order core, at half the frequency of the Beta Core is able to verify a 4 issue out-of-order core. Since the simple Checker Core cycles at half the frequency, we can use a low power ASIC flow, clock gating, and several trade-offs to further reduce power consumption.

To estimate the power and performance overhead, we synthesized a 4 issue out-of-order core (IVM [11]) and a simple in-order core (Rachael [12]) with STMicro 90nm technology. We did not use a low power ASIC flow or clock gating for the simple core, thus demonstrating our results are actually worst case power consumption. The synthesis results show that the Checker Core consumes only 50mW, while the Complex Core consumes 2.8W. Checking the signature tables adds approximately another 50mW overhead. This means that the proposed BCS solution increases power consumption by less than 4%. Equally important, the Checker Core requires less than 5% of the area used by the Beta Core.

The rest of the paper is organized as follows. Section 2 covers related work; Section 3 describes the proposed Beta Core Solution and the required hardware to build signatures; Section 4 characterizes the setup of our evaluation; Section 5 presents and analyzes some of our results from simulation and implementation of the proposed architecture; and Section 6 concludes.

2. BACKGROUND INFORMATION

2.1 Defect Arrival Model

Figure 1 shows a typical defect arrival or detection rate for a generic processor. The figure uses the SLIM [10] model which is commonly used by software and hardware industries to estimate the number of pending or still unknown defects. Initially, very few bugs are detected, as more lines of code are added, additional bugs are created and corrected. For a period of time, most bugs are eliminated, but some bugs still exist. There are always hard to find bugs which require a significant amount of time to detect and/or correct. Solutions with a priori bug detection mechanisms protect from infrequent bugs, but they can not tolerate significantly buggy cores because of the unbound performance impact. As a result, they are more of a safety net than a technique to release a processor earlier to market. Our Beta Core Solution releases partially verified processors.

Beta Cores have a higher bug frequency than current released cores which could be considered the equivalent to a final release in software. Figure 1 shows that by tolerating 1% of the bugs, the processor could be released significantly earlier. This is not an exact value because the SLIM model needs to be adjusted per core, but it clearly shows the potential.

2.2 Prior Work

There are several architectures [6,7,13–18] that propose a buggy core coupled with a Checker Core. In every case, except for Paceline [7], the Checker Core is always active. This always on property

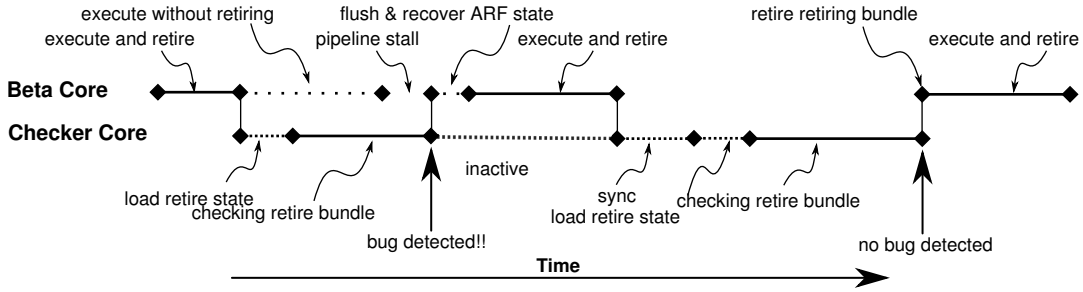


Figure 2: Sample BCS execution timeline

is good for verification, but it adds significant performance pressure to the Checker Core and power overhead to the system.

DIVA [13] is the seminal work proposing an architecture capable of tolerating buggy cores. The authors propose an always re-execute architecture which verifies all retiring instructions with custom hardware. Although DIVA checks instructions before they retire, it is different from our BCS solution and modern always re-execute solutions. DIVA's checking hardware is not a complete core and therefore does not allow for decoupled execution. The checking hardware is customized logic comprised of two parallel pipelines, one for verification of all functional computations and the other for memory verification. The checking component can essentially be seen as additional pipeline stages required for instruction retirement. The checker hardware is a 4-way in-order unit which cycles at the same frequency as the processor. Our checker is a 1-way in-order core, which cycles at half the frequency of the Beta Core, the low activity rate for our Checker Core supports this implementation.

DIVA's checker hardware is coupled with the main processor architecture, as such, it could prove more tedious from a verification perspective. The tight coupling of the main core with the verification logic requires a higher level of core modifications. To maintain information used by the checking logic, the main core design is complicated. This intrusive design specification could result in substantially more design and verification time. DIVA was also designed for low bug frequency, whereas BCS is aimed at high frequency bug systems. If the DIVA Checker is active 100% of the time, it has 120 times slowdown on average [19]. BCS has an average slowdown of 4 times when active 100% of the time. Power analysis was not provided for DIVA, however, the impact of a 4-way multi-stage checker unit could prove substantial. Our evaluation provides details for power and synthesis flow.

More recent proposals use full cores to verify the execution of retiring instructions. This "checker core" is decoupled from the potentially buggy core allowing for a smaller performance impact on the potentially buggy core. Recent works [6, 7] have been designed to tolerate cores with high bug frequency. In comparison, DIVA was designed for infrequent bugs. In this work, we focus on the more modern re-execution models because we target high bug frequency systems.

Paceline [7] proposes an architecture similar to Tandem [6] but allows the checker to be deactivated. Their deactivation is at coarse level, and it's main goal is to catch faults generated from an overclocked core. If it is not overclocked there is no need to have it checked.

In Tandem [6], a 4 issue in-order core was able to check a 4 issue out-of-order core. The out-of-order core was able to provide prefetched information from the caches and the branch predictors. In Slipstream [15], the leader executes only a subset of the instruction stream. The leader core executes speculatively, only what it thinks is necessary for forward progress. The checker, executes the same instruction stream non speculatively. The main focus of this work was to increase speed of execution, but the biggest difference with our work is that both cores need to be fully verified. In all the above works, the checker is always active, which is a big difference from what this work proposes. Another main difference with all existing work is that all of them propose architectures capable of tolerating infrequent bugs, unlike our solution where we propose an architecture able to tolerate frequent bugs. This idea of a higher tolerance to buggy processors is vital if designers seek to decrease the time to market through a reduction in verification time.

Another work [20] proposes an architecture aimed at tolerating bugs yet to be discovered. But, their approach is dependent upon the quality of signals built during verification time. The authors state that the dependency can be overcome by creating additional guardians. Additionally, the core's minimum functionality has to be 100% validated. Since we remove the dependency on signals, our approach is more robust.

3. THE BETA CORE SOLUTION

At a high level, the proposed architecture couples a simple in-order processor, Checker Core, with a partially verified complex out-of-order processor, Beta Core, as a method to capture design bugs. The Checker Core can verify the instructions before they retire from the Beta Core. To reduce overheads and to allow a much simpler Checker Core, a signature database determines whether the Checker Core needs to verify the Beta Core. Utilizing the signature information we can determine whether or not a bundle of instructions should be allowed to retire without verification. We refer to the instructions that can be retired in the current clock cycle as the retiring bundle.

Some bugs prevent the processor from retiring any instructions. To prevent stalls in the system, we implement a deadlock detector in the Beta Core. If no instruction retires for an extended period of time, we flush the Beta Core pipeline and transfer the context to the Checker Core to execute additional instructions before transferring context back to the Beta Core. The Checker Core must execute the additional instructions before control is transferred back to the Beta Core because defects have a high temporal locality.

3.1 Beta and Checker Core Coordination

Figure 2 shows a sample flow of the BCS architecture. We see the Beta Core executing instructions, then activate the Checker Core when a new signature is detected. The Checker Core must then be synchronized with the Beta Core. The Checker Core loads the retirement state and verifies the current retiring bundle. When

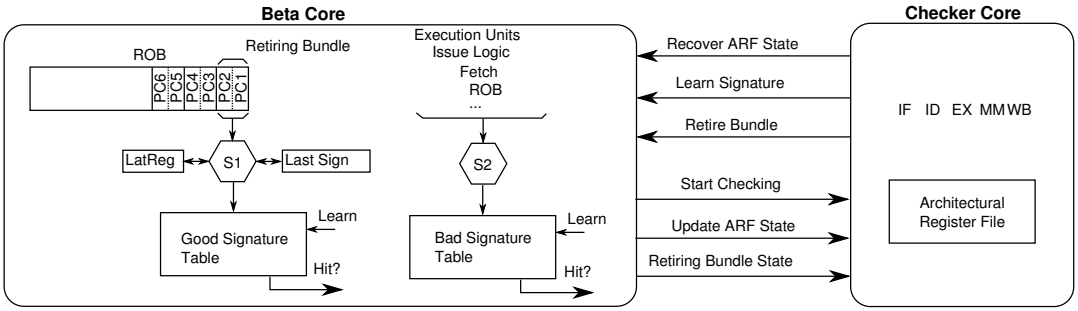


Figure 3: Beta Core Solution integrating an out-of-order Beta Core and a simple in-order Checker Core. LatReg is the latency register indicating the number of cycles since the last retirement. S1 and S2 are the signature generators for the good and bad signatures respectively.

the Checker Core detects a bug, it notifies the Beta Core. The Beta Core must flush it's pipeline and recover the ARF state from the Checker Core. After a few cycles, the Checker Core is inactive and the Beta Core resumes normal execution.

In the second portion of Figure 2 we see a scenario that occurs when the Checker Core does not detect a bug, the Checker Core is activated, it synchronizes, and loads the retirement state. After this it checks the retirement bundle, but sees that no bug is detected. The Beta Core is now allowed to retire the instructions it previously fetched and executed.

Figure 3 shows the overall proposed architecture. The communication between the Beta Core and the Checker Core is performed in the following manner. When a Start Checking trigger is received from the Beta Core, the Checker synchronizes itself with the Beta Core. Once synchronization is completed, the Beta Core sends the Retiring Bundle State to the Checker. The Checker reads the retiring bundle state and simultaneously starts to execute the bundle of instructions. If the bundle of instructions generate a correct result, the Checker Core notifies the Beta Core to learn the signature (Learn Signature) and to retire the bundle (Retire Bundle). If there is a mismatch, the Beta Core is flushed and the Checker executes a fixed number of additional instructions (Checker Active Instruction) to remove potential bugs related to consecutive instructions. The Checker then copies the architectural register state to the Beta Core (Recover ARF State). Once the transfer is completed, the Beta Core starts to execute as usual and the Checker Core becomes inactive.

3.2 Signature Composition

A key component of the proposed architecture is the utilization of signatures. We have a good signature table and a bad signature table. While the good signature table targets frequent bugs, the bad signature table provides a safety net by protecting against all the infrequent bugs and the bugs not captured by the good signature table.

The good signature is dynamically built to detect already verified components from the Beta Core. The bad signature table is constructed in the same manner as the a priori bug detection work [8] which essentially creates a database of bad signatures created from designers after the processor has been fabricated. In this work, we focus on the good signature table, and as the evaluation will show we did not need to create bad signatures for any of the applications and bugs analyzed.

For comprehensive reasons, a complete signature might include all the control and data values from the Beta Core. However, the complexity associated with this type of signature proved to be unnecessarily high. This would also require an unreasonable amount of memory (several megabytes in our experiments) to store the signatures. The resulting database with too many new signatures requires the Checker Core to be frequently active.

Algorithm 3.1: CREATE GOOD SIGNATURE(*every cycle*)

if (rob_retire_rf)

then $\begin{cases} lat\_reg \leftarrow 0 \\ sign\_tmp \leftarrow hash\{retired\_inst, lat\_reg\} \\ signature \leftarrow \{sign\_tmp, last\_sign\} \\ last\_sign \leftarrow signature \end{cases}$

else $\begin{cases} lat\_reg \leftarrow lat\_reg + 1 \\ signature \leftarrow last\_sign \end{cases}$

return $(signature)$

Figure 4: The good signature algorithm.

Our solution divides the problem into two signatures: good and bad. The good signature covers the most frequent cases, but it does not guarantee results. The correctness can be guaranteed with a complex but infrequently used bad signature, which triggers the Checker Core.

Figure 4 provides an algorithmic explanation of the good signature creation. This algorithm corresponds to the S1 block from Figure 3. The proposed good signature includes retiring instruction information and timing references. Although it would improve the signature quality to incorporate the data inputs and outputs for each retiring instruction, this significantly increases the number of required signatures. Building the good signature is a two-step process, we first build a hash using data commonly available during retirement: destination physical register id, PC, re-order buffer id, load store queue entry id (mobid), and whether the instruction is a load or a store. In the second step we add the number of cycles since the last retiring bundle, and the signature of the last retired bundle. The reason for including the timing information is that many bugs are timing dependent, and without including timing in the signature we miss many bugs.

The Beta Core creates a good and bad signatures every cycle. A signature can be thought of as a set of signals which capture the current state of the processor. If the generated signature is a hit in the good signature table and a miss in the bad signature table, the

processor retires the bundle, updates the Checker Core architectural register file (Update ARF State), and proceeds as usual. Otherwise, the Beta Core stalls the retirement, triggers a Start Checking signal, and transfers the state of the instructions in the retiring bundle (Retiring Bundle State) to the Checker. This state transfer requires several cycles. We call this the Checker constant activate overhead.

An important decision we had to make was the optimal signature size. A very small signature reduces the hardware requirements needed to generate and store the signature, but it increases the probability of a false positive. A false positive will occur when a new signature is created for a situation that has previously been verified. It is important to notice that most false positives will not have a large negative effect because it is expected that most of the time the Beta Core will be correct.

To estimate the signature size, we set a requirement that a false positive should miss a bug once every 30 years or more for typical applications. The SPEC application, with the highest number of signatures, generates a new signature every 40 cycles on average. Conservatively, we assume a 3GHz processor with an IPC 1 that has 1000 faults per second on average. These very conservative assumptions require 40 bits/signature.

$$40 = ceil(log_2(\frac{(30 \text{ years} * 3 \times 10^7 \ \frac{\text{sec}}{\text{year}} * 4 \times 10^9 \text{ Freq})}{4 \times 10^6 \text{ Cycles between faults}})) \quad (1)$$

The size of the signature table is determined by the size of each signature and the total number of signatures. It is only necessary to keep the tag of the signature in the table. For example, a signature table which contains 43 bit signatures and 8K entries(13 bits) would require 30,000 Bytes (8K*30/8).

Every time a new good signature is detected the Checker Core re-executes the same instruction as the Beta Core. If the retiring instructions, for which the signature is being created, finish correctly on the Checker Core the new signature is saved. Minimizing the overall number of new signatures while at the same time capturing all correct states is the goal. Our results show that a 16K cache suffices and yields good results.

3.3 Timing Relevance

Bugs which manifest based on the current state of the processor, or occur as a result of other instructions in concurrent execution, are referred to as timing related bugs. Bugs can occur due to environment and not solely because the instruction has an inherit bug. For example, a cache miss or a full instruction buffer may invoke signals which cause the executing instructions to fail.

The timing of each bundle is captured through our LatReg which counts the number of cycles that have elapsed since the last bundle of instructions was retired. This timing information is equivalent to time stamping the newly fetched instruction from the fetch stage and through retirement, but this would require significantly more resources.

The previous signature is not perfect because it can not capture bugs like $2+2=3$ if we have previously executed the $3+3=6$ in the same instruction. Originally, we thought that this would have been a problem, but after analyzing all the IVM bugs and the hwBugHunt bugs [21] we found that this type of time-independence bug was not common. Most of the bugs happen when some "uncommon" condition happens like a buffer full when X happens.

As an example, the code snippet shown in Figure 5, demonstrates that these types of bugs cannot be captured without timing information. In this piece of code, we look for the most recent load to be forwarded. But due to the existence of the bug instead of the most recent load, the oldest load gets forwarded. The bug only affects the result when the load store queue provides incorrect data forwarding. We found that, for some loads, the first execution did not have forwarding, and successive executions did. Since forwarding changes the execution time, a signature with timing information helps to protect against this timing dependent bug.

Algorithm 3.2: TIMING BUG EXAMPLE()

```
tmp_ldq = ldq_tail_f − 1;
i = 0;

while (ldq_valid_f[tmp_ldq]) &&
      (ldq_robid_f[tmp_ldq] != finished1hint_in) &&
      (i <= DQ_SIZE)
{ i = i + 1;
  tmp_ldq = tmp_ldq − 1;
  ldq_mobid_fin1 <= tmp_ldq;

  if(BUG_EXISTS)
    tmp_ldq = ldq_tail_f + 1;
  else
    tmp_ldq = ldq_tail_f − 1;
```

Figure 5: IVM bug not found without timing information.

Since some bugs can be missed with the good signature, we use a bad signature to capture more complex and difficult to detect bugs. The bad signature solution follows the previously proposed Phoenix approach. We can not use the same approach for the good signature because it would require too many signatures.

3.4 Checker Core Synchronization Overhead

This paper proposes active and inactive states for the Checker Core as a method to leverage the fact that it's a simple in-order core. This approach can only work if the Checker Core can be kept in an inactive state for a significant portion of the time, and if the activation overhead is kept in check. This section objective is to explain the sources and techniques to mitigate switching overhead.

There are several steps before the Checker Core starts to verify the retiring bundle of instructions. First, the Checker Core is activated. Second, the Beta Core then loads the architectural state to the Checker Core. Third, the Beta Core transfers the retiring bundle state. The final step occurs when when the Checker Core starts to execute and verify the retiring bundle of instructions.

Loading the ARF from the Beta Core to Checker Core requires a transfer of all the architectural registers (approximately 64 registers). Since the Checker Core operates at half the frequency and we can use the 3 ports from the register file (1 WR and 2 RD/WR ports), it requires approximately 44 cycles ($44 \approx 2 * \frac{64}{3}$). We performed some simulations based on the signature generated with a 16K entry good signature table, and we observed under 100 cycles between Checker Core activations on average. This means that for an ideally fast Checker Core, the overhead will be over 30%. Clearly this is not acceptable.

To mitigate the overhead, we propose several techniques: First, we propose to update the Checker Core architectural state while the Beta Core retires instructions. After a retirement takes place in the Beta Core, the results are sent to the Checker Core for writes into its register file. This eliminates some of the activation over-

Structure	Beta Core and Baseline	Checker Core
Out-of-Order	Yes, 4 issue	No, single issue
Frequency	3GHz	1.5GHz
Issue Window/LSQ/ROB	16-entry/64-entry/256-entry	NA
McFarling Predictor	16K/16K/16K entries	NA
RAS/BTB	32/4K 4-way	NA
L1 Data	32 KB, 2-way, 32 B line, 2 clk	16 KB, 2-way, 32 B line, 1 clk
L1 Inst	32 KB, 2-way, 64 B line, 2 clk	16 KB, 2-way, 64 B line, 1 clk
L2 Data shared	1024 KB, 8-way, 64 Byte line, 4ns hit, 60ns miss	
Good Signature Table	16K entries x 40 bits entry	
Deadlock Detector	333 ns	

Table 1: Beta Core, Checker Core, and shared simulation parameters.

head associated with preparing the Checker Core for execution. As our results demonstrate it is more efficient to keep the architectural states of both cores in sync. This functionality is already provided by several ROBs that update the local Architectural Register File (ARF) as the instructions retire. Our proposed system allows for the Beta Core to update the local ARF and the Checker Core ARF.

We also propose to keep the Beta Core executing instructions while the Checker Core verifies the retiring bundle, we just need to block the retiring. This works extremely well as the most common case is for the Checker Core to not find a bug in the Beta Core. In such cases, the Beta Core can "prepare" work and accumulate instructions ready for being retired.

Finally, we allow the Checker Core to start executing the retiring instructions while we transfer the retiring bundle state. Even a simple 5 stage processor needs 5 cycles from fetch to retirement. These cycles can be used to overlap the state transfer. The resulting system can hide most of the overhead. The evaluation will show the overall impact.

4. SIMULATION SETUP

The goal of this work is to reduce the amount of time necessary to verify and release a processor. To validate this work, we used a modern high performance processor to implement our architecture. We used IVM, the Illinois Verilog Model, a 4 issue out-of-order Alpha processor. The processor has features like speculative instruction scheduling, memory dependence prediction, and a complex branch predictor. The simulation parameters are detailed in Table 1.

For RTL synthesis of IVM we used Synopsys Design Compiler C-2009.06 with a 90nm STMicro process node. The front-end flow was derived from Synopsys Reference Methodology as defined in the solvnet portal. It is important to note that we did not use any low power methodologies, in which case our synthesis output is actually worst case results. We used an Alpha EV6-like processor for the signature and bug detection evaluation. The Illinois Verilog Model (IVM) [11] implements a subset of the Alpha EV6 architecture, it was designed originally at the University of Illinois for fault-tolerant research. It contains over 30K HDL statements. The processor is not complete, and it has several bugs. For example, it can not execute most of the SPEC CFP2000 applications because it does not support floating point operations. Also, it fails to completely execute the SPEC benchmarks because of several bugs. Very few benchmarks can execute over 100K instructions without requiring a pipeline flush. In this regard IVM actually serves as an ideal candidate for our work. It provides an architecture with complex features, but also not thoroughly verified.

We use Rachael [12] to characterize the Checker Core. Rachael is a derivation of the Leon2 [22] processor, both are 5 stage in-order SPARC32 ISA processors.

Design Error Category
Wrong signal source
Conceptual error
Case statement
Wrong constant
Incorrect logical expression
If statement
Wrong operator

Table 2: Bugs introduced to the Beta Core.

For the evaluation we used a subset of SPEC CINT2000 (bzip2, crafty, gcc, mcf, perlbmk, vpr, gap, twolf, eon) on the simulated Verilog design. SPEC CFP2000 is not evaluated because IVM lacks support for floating point operations. The missing integer benchmarks were excluded because IVM had a very high failure rate for these applications, thus the Beta Core had bugs every few cycles, and therefore the Checker Core is always active. In practice this is not very realistic, this is equivalent to executing all instructions on the Checker Core. Although we do not present these results, our signature detected and corrected all bugs. We did evaluate the performance impact when the Checker Core is active 100% of the time, this diminished performance by roughly 4 times.

Modeling at the Register Transfer Level (RTL) is a slow process, to speedup simulation we used SimPoint 3.2 [23]. Since RTL is several orders of magnitude slower than traditional architectural simulators ($\approx 0.02 KIPS$), we used simulation intervals of 20M instructions with 2 intervals per benchmark. We warmed up the RTL with the architectural simulator state. As a result, the RTL simulation has a correctly trained state before each simulation point begins.

5. EVALUATION

In this section, we present the overall results of our proposed Beta Core Solution (BCS). The architectural parameters are shown in Section 4. We evaluate our BCS architecture against a baseline architecture and an "always re-execute" system. The baseline architecture is a 4-issue out-of-order core. The always re-execute implementation is the same as the BCS architecture, however, the

Checker Core is kept active 100% of the time. We demonstrate the advantages of dynamically turning on the Checker Core when needed, rather than keeping it on at all times.

Figure 6 shows the performance slowdown for the proposed BCS architecture compared against a previously proposed a priori bug detection architecture and an always re-execute architecture. As Figure 6 shows, the Beta Core Solution tolerates buggy cores as well as the always re-execute proposal, and it limits the maximum performance penalty for the worst case situations. It is able to do so with a simple in-order core cycling at half the frequency, which has significant power saving advantages.

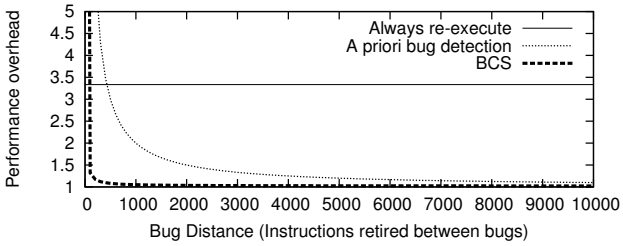


Figure 6: Slowdown for different techniques to tolerate bugs

5.1 Overall Results

Our results show that the proposed BCS captures all frequent bugs with our good signature table. To evaluate this effectiveness, we inserted bugs from two sources: hwBugHunt and the bugs already existing in IVM.

The bugs inserted from hwBugHunt differ in classification and provide a a wide range of plausible bugs. Table 2 classifies the types of inserted bugs.

IVM has several bugs because it was designed, mainly by one student, in one year as a university project [24]. On average a bug occurs every 200 retired instructions. This is due to non exhaustive verification. Hence, we consider the IVM processor to be an ideal candidate for a Beta Core.

The proposed solution detected all the bugs found during RTL simulation. We added a bug from each of the categories listed in hwBugHunt [21]. Our good signature found all the bugs. We did not use the safety net provided by the bad signature table for any of the bugs.

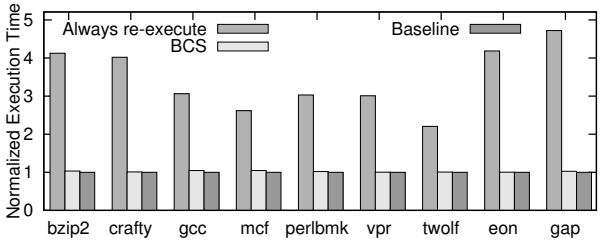


Figure 7: Execution time for a plain Checker Core and an always re-execute Checker Core.

In Figure 7, we show the normalized execution time to the baseline. Our methodology is labeled as BCS and the first bar is the performance for the always re-execute. Clearly, our methodology, is able to keep up with the 4 issue out-of-order in terms of performance with a slowdown of an average 1.6%.

The always re-execute methodology has a high execution time because the Checker Core is running at half the frequency of the 4 issue out-of-order. For our methodology, instead of keeping the Checker Core active at all times, we activate it only when needed. On average, the Checker Core needed to be turned on only for about 3.6% of the 9 benchmarks listed.

The benchmark requiring the least Checker Core activity is eon, the most is gcc. Applications that have many phases will have many new signatures. Table 3 shows that the highest slowdown has a corresponding high activity rate. Similarly, eon has the least slowdown and it induces the least Checker Core activity.

The worst case in terms of average number of instructions between new signatures ($\frac{\#Inst}{\#Sign}$) is gcc with just 5.4 instructions. Nevertheless, it is active only 6.4% of the time. Using the 1.2 IPC and the active time from gcc, we get around 18 cycles/sign which is clearly higher than the gcc's 5.4 $\frac{\#Inst}{\#Sign}$. The reason is again the new signature clustering. Intuitively, whenever a program reaches a new phase, it generates many back to back new signatures, which trigger a Checker Core activation.

5.2 Synthesis Results

We synthesized the 4 issue out-of-order Beta Core (IVM [11]) and a Checker Core (Rachael [12]) with STMicro 90nm technology to study the power and area overhead. The synthesis results show that the Checker Core consumes only 50mW while the Beta Core consumes 2.8W. For our signature caches, we obtained a 50mW output from our 90nm memory compiler. Therefore, we can safely state BCS increases power by 4%, and the Checker Core uses less than 5% of area used by Beta Core.

5.3 Overheads Characterization

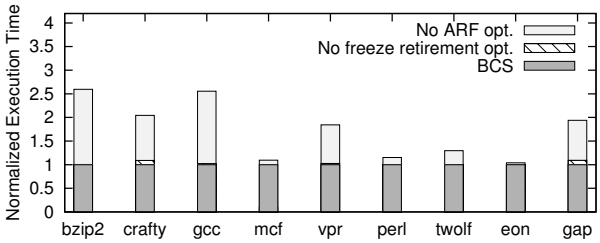


Figure 8: Performance impact of the proposed optimization techniques (freeze, adaptive, and ARF transfer)

Figure 8 shows the effect each overhead has on the execution time. There are several overheads before the Checker Core can begin to verify the retiring bundle of instructions. Loading the architectural register file, transferring the bundle of retiring instructions.

Optimization 1: Transfer ARF State After the Checker Core is active, we need load it with the current architectural state. To load the architectural register file after the Checker Core is active would require about 44 cycles (Section 3.4). Clearly, this is expensive, so we propose to update the Checker Core ARF, while the Beta Core updates its ARF. Figure 8 shows the breakdown without our optimization.

Optimization 2: Freeze Only ROB When the Checker Core is verifying the retiring bundle, we allow the Beta Core to continue executing instructions and just block it from retiring more instructions, until the Checker Core has verified the current retiring bundle. During this phase, the Beta Core can accumulate instructions and once the verification is done, it can retire the instructions. To

Benchmark	bzip2	crafty	gap	gcc	mcf	perlbmk	vpr	twolf	eon	average
IPC	2.20	1.83	2.44	1.20	0.150	1.07	1.141	0.855	2.045	
Active Time (%)	2.7	3.9	4.6	6.4	4.9	3.3	1.2	4.9	0.064	3.6
$\frac{\#Inst}{\#Sign}$	18.8	9.4	8.2	5.4	15.8	6.9	23.2	119.0	17.7	24.9

Table 3: Main results showing the percentage of time with the Checker Core active (% Active Time) and the distance in retired instructions between new signatures.

Benchmark	∞ entry	64K entry	32K entry	16K entry	8K entry	4K entry	2K entry	1K entry
bzip2	99.5	98.5	97.7	95.7	93.3	89.4	82.8	74.1
crafty	98.7	95.4	92.8	88.5	82.1	72.3	59.9	44.8
gcc	95.2	90.3	86.4	80.1	70.9	59.8	48.3	37.4
mcf	99.5	97.6	96.2	94.1	91.2	86.7	79.8	71.5
perlbmk	98.7	94.4	90.2	84.0	74.6	60.3	44.9	31.0
vpr	99.7	98.8	97.8	96.0	93.0	87.4	77.7	65.8
twolf	99.5	98.3	97.8	93.0	83.0	77.4	65.7	55.4
eon	99.7	98.4	97.4	94.0	74.0	60.4	45.7	40.8
gap	99.2	95.3	91.7	85.6	75.8	62.8	47.1	32.3
average	98.6	95.8	93.3	89.1	83.0	74.1	62.9	51.0

Table 4: Signature table hit rate (%) for several table sizes.

study the effect of this optimization, instead of freezing only the ROB, we freeze the Beta Core while the Checker Core is checking the correctness of the retiring bundle.

Figure 8 studies the impact each of these optimizations has on execution time normalized to the proposed BCS execution time. The label marked "no freeze" is where we we stall the Beta Core while the Checker Core verifies the retiring bundle. Here we see that it in some benchmarks it adds additional execution time, thereby decreasing the performance of the Beta Core. The transferARF optimization has a bigger impact. When we removed that, the Checker Core was required to transfer the previous correct ARF state to itself before verifying the bundle of verifying instructions.

5.4 Signature Characterization

A key characteristic of the proposed architecture is the use of signatures. This section starts showing the cache miss rates for different signature table sizes, and continues showing problems associated with simpler program phase signatures and more complex control signal based signatures.

Table 4 shows the cache hit rates for several good signature tables. An unlimited table has 98.6 average hit rate. From a high level first approximation, this means that a program with an IPC of 1 needs a signature every 71 cycles. In reality it is much larger because the signatures are not randomly distributed. They have a log-normal distribution with most of the signatures having a 1 or 2 cycles distance. We performed simulations and 22% of the signatures have a distance of 1 cycle. This means that whenever a new signature is found, there is a 22% chance that the following cycle will be a new signature too. Table 4 also provides insights on table size requirements. Tables smaller than 4K entries start to decrease the cache hit rate faster. As expected the larger the table the better. Since it will be difficult to cycle a 32K entry table in a couple of cycles, we decided with a 16K table for this work.

What if the signatures used all the control signals?
We also performed experiments to understand the effect of using all control signals to compute the signature. The problem with this method is that the signatures become too large and would require extremely large caches. We found that the signatures had a max hit rate of 1.9% for twolf, a minimum hit rate of 0.1% for gap and perlbmk, while the average hit rate was .5%. This type of signature also forces the Checker Core to be constantly active due to the large number of new signatures.

What if the signature just used the program phases?
In our experiments, we have observed that tracking all the basic blocks is not sufficient to build signatures to detect all the bugs. We missed several bugs when only the PC was used. The reason is because, the basic blocks alone are not sufficient to capture all the state information. We find that in the buggy IVM [11], we miss a bug every 925 retired instructions when only the basic block information is used. Thus it is clear that in order to capture all the bugs in the system, we need more than just the program phase information.

What if the signature does not include timing information?
We removed the timing information from the signature, and the IVM processor missed bugs in all the benchmarks. For benchmarks like crafty we missed up to 25% of the bugs. Clearly, the timing information is necessary.

6. CONCLUSIONS

Ideally, processor verification would be simple and capture all bugs before a processor is released into the market. Since this is clearly not the case, researchers have proposed architectures to tolerate infrequent bugs [8, 13]. This work goes a step beyond these approaches and proposes a design to efficiently tolerate, not only infrequent bugs, but frequent bugs found in partially verified processors like IVM. The proposed solution allows for the release of Beta Cores earlier to market.

This paper makes several contributions; It proposed the Beta Core Architecture, which included a novel signature generation that avoided redundant checks. This allowed for our simple half frequency in-order Checker Core to run alongside a Beta Core correcting hard to find bugs while incurring a minimal amount of overhead. We also proposed a simplistic Checker/Beta Core interaction as a method to limit the complexity, power overhead, and area of the Checker Core. Our signature creation methodology integrated

ideas of program phases with control signals to avoid overly complex large signatures.

We proposed a novel signature composition to capture frequent bugs with a good signature table. The evaluation shows that a traditional signature with most control signals may work for the bad signature table, but it does not work for the good signature table due to a very high (99.5%) miss rate. Such a high miss rate keeps the checker frequently active. The evaluation also shows that remembering previously verified basic blocks does not work either because it misses too many bugs (1 bug every 925 instruction). Our signature methodology produces a hit rate of over 90% in the good table. Furthermore, it does not miss any bugs inherit in IVM or inserted from hwBugHunt.

Power is a major design constraint in current systems. The BCS solution is effective because the power consumption is kept in check with a 5% overhead using detailed synthesis results. It is important to note that our Checker Core was not subjected to any power optimizations. It does enter states of inactivity, however, our evaluation considers it to be on at all times but not necessarily performing corrections on the Beta Core.

Processor verification is an arduous necessary task, the goal of this paper is to reduce this time. In an effort to shorten the release to market time frame, while at the same time not compromise the integrity of the processor, BCS proves to be a valuable architecture. Our results showed that our checker was active only 3.5% of the time on average with just a 1.6% overall system slowdown.

7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants 0546819, 0720913, and 0751222; OpenSPARC Center of Excellence at UCSC; Special Research Grant from the University of California, Santa Cruz; gifts from SUN, nVIDIA, Altera, Xilinx, ChipEDA, and AMD. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] Intel Corportation, "Intel xeon processor, specification update doc. no.249678-056," Dec 2006.

[2] Intel Corportation, "Intel core 2 extreme processor x6800 and intel core 2 duo desktop processor e6000 and e4000, specification update doc. no.313279-024," Feb 2008.

[3] Intel Corportation, "Intel pentium m processor, specification update doc. no.252665-033," Jan 2008.

[4] Intel Corportation, "Intel pentium 4 processor, specification update doc. no.249199-069," May 2007.

[5] AMD, "Revision Guide for AMD Family 10h Processors," 2011.

[6] F. Mesa-Martinez and J. Renau, "Effective optimistic-checker tandem core design through architectural pruning," in MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, Dec. 2007, pp. 236–248, IEEE Computer Society.

[7] B. Greskamp and J. Torrellas, "Paceline: Improving single-thread perfomance in nanoscale cmps through core overclocking," in the 16th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, Sep. 2007, IEEE Computer Society.

[8] S. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," Microarchitecture, IEEE/ACM International Symposium on, vol. 0, pp. 26–37, Dec. 2006.

[9] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation," in 41st Annual International Symposium on Microarchitecture (MICRO-41), Nov. 2008.

[10] S. Fenstermaker, D. George, A. Kahng, St. Mantik, and B. Thielges, "METRICS: A System Architecture for Design Process Optimization," in International Conference on Design Automation, Jun. 2000, pp. 705–710.

[11] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in International Conference on Dependable Systems and Networks. Jun. 2004, IEEE Computer Society.

[12] M. Cowell and A. Postula, "Rachael SPARC: An Open Source 32-bit Microprocessor Core for SoCs," in Proceedings of the 2006 9th EUROMICRO Conference on Digital System Design, Aug. 2006, pp. 415–422.

[13] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in the 32th International Symposium on Microarchitecture, Nov. 1999, pp. 196–207.

[14] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," in International Symposium on Computer Architecture, San Diego, California, Jun. 2003, pp. 98–109.

[15] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in the 33th International Symposium on Microarchitecture, 2000, pp. 269–280.

[16] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," in ISCA, 2000, pp. 25–36.

[17] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in Symposium on Fault-Tolerant Computing, 1999, pp. 84–91.

[18] C. Weaver and T.M. Austin, "A Fault Tolerant Approach to Microprocessor Design," in DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS), Washington, DC, USA, Jul. 2001, pp. 411–420, IEEE Computer Society.

[19] Todd Austin, "Diva: A dynamic approach to microprocessor verification," Journal of Instruction-Level Parallelism, vol. 2, pp. 2000, May 2000.

[20] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians," in Proceedings of the conference on Design, automation and test in Europe, San Jose, CA, USA, Apr. 2007, DATE '07, pp. 743–748, EDA Consortium.

[21] S. Sudhakrishnan, L. Su, and J. Renau, "Processor verification with hwbughunt," in ISQED '08: Proceedings of the 9th international symposium on Quality Electronic Design, Washington, DC, USA, Mar. 2008, pp. 224–229, IEEE Computer Society.

[22] J. Gaisler, "The Leon-2 Processor," 2003.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 2002, vol. 30, pp. 45–57.

[24] C. Bazeghi, F.J. Mesa-Martínez, and J. Renau, "μComplexity: Estimating Processor Design Effort," in International Symposium on Microarchitecture, Nov. 2005.