Appears in the 40th Annual International Symposium on Microarchitecture (MICRO)
Chicago, Illinois, December, 2007

Argus: Low-Cost, Comprehensive Error Detection in Simple Cores

Albert Meixner
Dept. of Computer Science
Duke University
albert@cs.duke.edu

Michael E. Bauer
Dept. of ECE
Duke University
michael.bauer@duke.edu

Daniel J. Sorin
Dept. of ECE
Duke University
sorin@ee.duke.edu

Abstract

We have developed Argus, a novel approach for providing low-cost, comprehensive error detection for simple cores. The key to Argus is that the operation of a von Neumann core consists of four fundamental tasks—control flow, dataflow, computation, and memory access—that can be checked separately. We prove that Argus can detect any error by observing whether any of these tasks are performed incorrectly. We describe a prototype implementation, Argus-1, based on a single-issue, 4-stage, in-order processor to illustrate the potential of our approach. Experiments show that Argus-1 detects transient and permanent errors in simple cores with much lower impact on performance (<4% average overhead) and chip area (<17% overhead) than previous techniques.

1. Introduction

Technological trends are leading to more hardware errors, due to both transient and permanent physical phenomena [28, 7]. The first and most important step in tolerating these errors is detecting them. Once an error is detected, the system can avoid silent data corruption and potentially recover to a pre-error state and resume execution. In this work, we focus on relatively simple cores, rather than speculative out-of-order cores. Simple cores are becoming more attractive due to their low power consumption, particularly for multicore chips, embedded applications, and applications for which throughput is more important than latency. For example, the UltraSPARC T1 (Niagara) chip [10] contains 8 simple cores, the Cray MTA [3] consists of simple multithreaded cores, and the Silicon Packet Processor in CISCO's CRS-1 router [4] has 188 Tensilica Xtensa single-issue, in-order cores. For many applications of simple cores, reliability is important but it must not cost much in terms of hardware and power.

Core error detection can be achieved by simply replicating each core (dual modular redundancy), but this option is extremely expensive. Even if providing the required number of transistors is technologically feasible, DMR incurs a large opportunity cost and approximately doubles core power consumption. Other detection schemes, such as DIVA [1, 31] and redundant multithreading [23, 21, 16], exist, but all of them are either incomplete or expensive, in terms of area or performance, for simple cores.

Our goal is to provide a low-cost, low-power mechanism for comprehensively detecting transient and permanent errors in a simple microprocessor core. After detecting an error, the core can recover to a pre-fault state using a checkpoint recovery mechanism [27]. Instead of low-level checking of each core component, our scheme, Argus, uses run-time checking (dynamic verification) of the following four invariants that guarantee the core is operating correctly:

- Control Flow: An error-free core must continue to make forward progress through the control flow graph specified in the program binary.
- Computation: An error-free core must correctly perform computations (additions, shifts, etc.).
- Dataflow: An error-free core must preserve the dataflow graph specified in the program binary.
- Memory: An error-free core must interact correctly with the memory system, and the memory system must not be corrupted.

Argus checks these four invariants by integrating existing mechanisms for runtime control flow checking [5, 9, 30], computation checking [24, 19, 20, 17], dataflow checking [15], and memory checking. We have proven that checking these four invariants is sufficient for detecting all possible single errors in an idealized core that has no I/O, exceptions, or interrupt handling.

To evaluate its hardware cost and error coverage, we have incorporated Argus error detection into the OpenRISC 1200 (OR1200) core [11], synthesized the Verilog, and laid out the design. Our results show that this implementation, called Argus-1, adds less than 17% to the core area (and less than 11% to the total chip area, including caches) and increases runtime by 3.2-3.9% on average. The implementations of Argus-1's four checkers do not detect all possible errors, due to cost constraints, but they still detect over 98% of all unmasked injected errors.

In the remainder of the paper, we first present an overview of Argus (Section 2) and the Argus-1 implementation (Section 3). In Section 4, we provide an experimental evaluation of Argus-1. In Section 5, we compare Argus to related work, and we conclude in Section 6. The appendices contain a proof of Argus's completeness and a proof of the equivalence of Argus-1's checkers and ideal checkers.

2. Argus Overview

The key insight exploited by Argus is that, at a high level, von Neumann processor cores perform only four basic activities: choosing the sequence of instructions to execute, performing the computation specified by each instruction, passing the result of each instruction to its data-dependent instructions, and interacting with memory. We prove in Appendix A that by checking that these activities are performed correctly, Argus can detect all possible core errors, except errors in the parts of the core that handle I/O, exceptions, and interrupts. This set of activities is quite similar to the set of activities checked by DIVA [2], but Argus's approach to checking them is fundamentally different.

We consider Argus to be a high-level error detection scheme, because these four activities are not specific to any particular micro-architecture, design, or implementation; they are present in any von Neumann processor. Strictly speaking, memory access is not a fundamental task but a form of dataflow. We chose to consider memory separately, because it differs significantly from dataflow between registers in ways that would make it difficult to implement a combined checker for register and memory dataflow.

In the rest of this section, we discuss the requirements of the four invariant checkers. Formal definitions of the invariants are given in Appendix A.

Control Flow Checking. A control flow checker [5, 9, 30] periodically verifies that the runtime execution path is valid with respect to the static control flow graph (CFG) of the program binary. If the static and dynamic CFGs conflict, an error has been detected. Unlike many control flow checkers, we also consider liveness to be a part of control flow correctness.

When used in isolation, a control flow checker detects errors in fetch logic, branch destination computation, and PC update logic. However, without dataflow and computation checking, a control flow checker cannot detect when an error causes the core to choose the *wrong* one of two possible data-dependent branch destinations. Argus's control flow checker interacts with the other checkers to detect these error scenarios.

Dataflow Checking. A dataflow checker [15] ensures that the static dataflow graph (DFG) of the program binary matches the dataflow graph reconstructed at runtime and that the values traversing the DFG are not corrupted. In isolation, a dataflow checker detects errors in many activities, including: fetch, decode, register rename, register read/write, and instruction scheduling (ROB, reservation stations, etc.).

Computation Checking. A computation checker detects errors in functional units. Checker implementations vary between units. Some checkers require a fully replicated functional unit, but many utilize knowledge about the initial result to simplify the redundant computation. Sellers et al.'s book [24] provides an excellent survey of existing checkers for adders, multipliers, dividers, bit-wise logic units, etc.

Memory Checking. A minimal memory checker must be able to detect data corruption in the memory system as well as errors that cause the wrong data word to be accessed. In more complex memory systems that support multiple outstanding requests and potentially multiple cores, faults can also manifest themselves as incorrect orderings of memory accesses. We do not consider this type of error, because there are only a few unlikely scenarios for ordering errors in simple cores. An example of a complex memory checker that could be used with Argus can be found in prior work [14].

3. Argus-1 Implementation

In this section, we describe an implementation of Argus, called Argus-1, that illustrates the engineering tradeoffs between checker costs and error coverage. Although perfect checkers can be designed, their costs are not always worth their additional error coverage, as compared to near-perfect checkers. In Appendix B, we prove that Argus-1 detects the same errors as an ideal Argus implementation, except for false negatives due to finite-sized checksums and memory ordering errors.

To obtain realistic information about the costs and complexity associated with implementing Argus-1, we have built Argus-1 in Verilog and incorporated it into the OR1200 processor core [11].

3.1. Baseline OpenRISC processor

The OR1200 processor core is a 32-bit scalar (1-wide), in-order RISC core with a 4-stage pipeline and 32 general purpose registers. It has an instruction cache and data cache, which we assume in this paper to both be 8KB. We study both direct-mapped and 2-way associative caches, with LRU replacement for the 2-way.

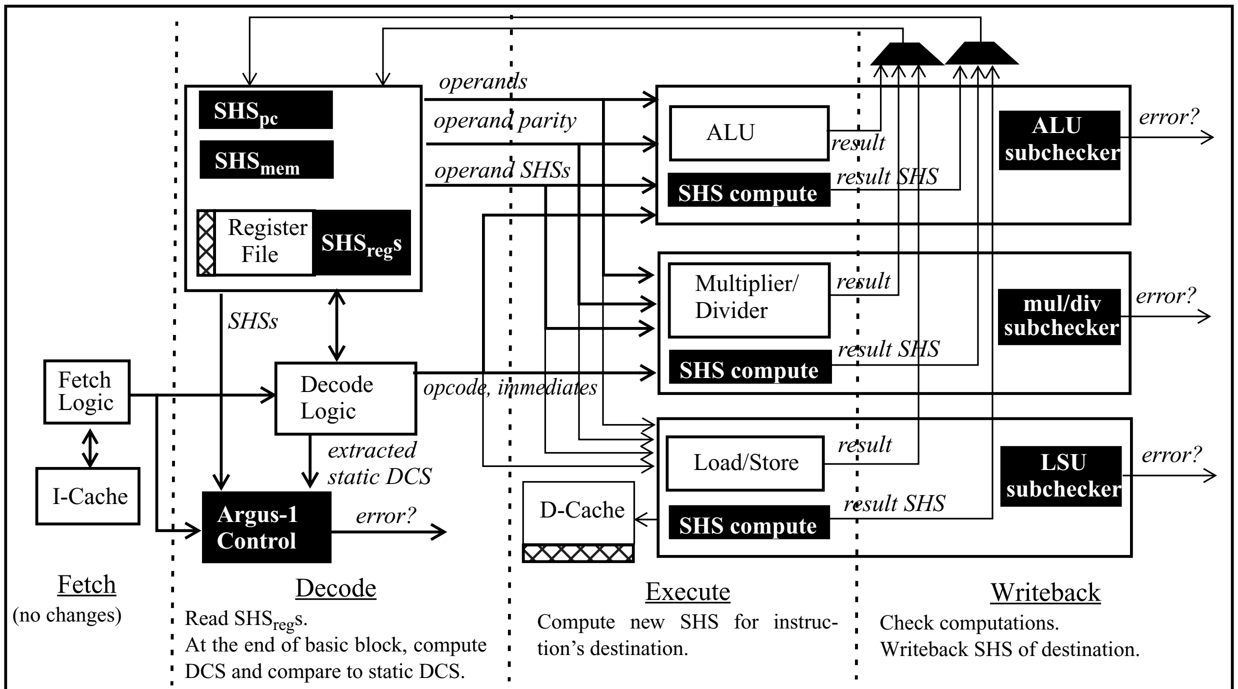

Figure 1. Summary of Argus-1 Implementation.
Dark boxes represent hardware added for Argus-1. The hashed boxes represent parity bits.

The data cache is write-back, write-allocate, and it blocks on misses. The OR1200 core has an integer ALU, a non-pipelined integer multiplier/divider, and a load/store unit, but no floating point hardware. There is a single branch delay slot and no branch penalty, so no branch prediction is necessary. This core represents the low-end of the simple cores that are expected to be used, perhaps in conjunction with a small number of superscalar cores, in multicore chips.

We illustrate in Figure 1 how Argus-1 is integrated into the OR1200 core, and we will discuss each of the Argus-1 additions in the rest of this section. Note that the sizes of the structures in the figure are not to scale; Argus-1 hardware comprises less than 17% of the core area (and less than 11% of the total chip area), as we show in Section 4.3.

3.2. Control Flow and Dataflow Checkers

Argus-1's control flow and data flow checkers are based on our earlier work on Dynamic Dataflow Verification (DDFV) [15]. DDFV was developed for dynamically scheduled superscalar cores and does not exercise its full potential when used with OR1200's simpler dataflow, but it is still necessary to check that the simple dataflow is error-free.

DDFV detects errors in the core's dataflow by comparing the static DFG specified in the program to the dynamic DFG within the processor during execution. Both DFGs are represented using constant-size signatures. The static signatures are computed at compile time and embedded in the program such that they can be read by the processor and compared to the dynamic signatures computed at runtime. To avoid problems with data-dependent branches, which dynamically alter the DFG, DDFV performs checks at the granularity of basic blocks for which the correct DFG is known at compile time. An incorrect DFG can go undetected if it maps to the same signature as the correct DFG (aliasing). The chance of aliasing can be arbitrarily reduced by increasing signature sizes.

DDFV implicitly checks the control flow within each basic block, but it cannot detect errors in control flow between basic blocks. To provide full control flow checking, Argus-1 adds a mechanism on top of DDFV that checks whether transitions between blocks are performed correctly.

3.2.1 Unifying Control Flow & Dataflow Checking

Argus-1 uses a basic block's dataflow signature as both (a) a representation of the block's internal dataflow, needed by the dataflow checker and (b) a unique, address-independent block identifier necessary for full control flow checking. We refer to this single signature as the *Dataflow and Control Signature (DCS)*.

What enables Argus-1 to use the DCS for both dataflow and control flow checking is the way DCSs are embedded and used. Unlike DDFV [15], which embeds a single signature into each block, Argus-1 embeds into each basic block the DCS of each of its legal successor blocks. At runtime, the control flow checker decides, based on information received from the computation checker, which of the legal successors will be executed next and passes the corresponding DCS to the dataflow checker. If, due to an error, the wrong successor or any other illegal block is executed next, the DCS computed by the dataflow checker will not match the DCS anticipated by the control flow checker and, barring aliasing, an error will be detected.

Figure 2 illustrates how Argus-1 embeds the DCS (in a Signature instruction, which is a NOP) for basic blocks with one or two successors. A basic block that ends with a conditional branch (e.g., BB1) contains the DCS of both its branch target (BB3) and its fall-through block (BB2). At runtime, during execution of BB1, the control flow checker will choose one of the two DCSs based on the direction the branch takes. For a block with a single successor (e.g., BB2 or BB3), only a single DCS is embedded.

```
BB1:   add r1, r2, r3
       sub r4, r1, r2
       Signature {BB2,BB3}
       beq BB3
BB2:   load r6, 0(r4)
       mul r7, r6, r6
       Signature {BB4}
       jmp BB4
BB3:   or r8, r6, r9
       Signature {BB4}
BB4:   and r10, r8, r6
       ...
```

Figure 2. DCS Embedding

3.2.2 Implementation Details

DCS Computation. We compute the DCS similar to the dataflow signature in DDFV [15]. A *state history signature (SHS)* is maintained for each architectural location; thus, we have an SHS for each register (SHS_{reg}), the program counter (SHS_{pc}), and memory (SHS_{mem}). An SHS for a given location represents the creation history of that location's current state. This history depends on the histories of the data and operations that were involved in creating the current state, but not on the data operand values themselves. For example, when *add r1, r2, r3* gets executed, then the new value of SHS_{r1} depends on the values of SHS_{r2} and SHS_{r3} and the fact that the operation was an addition, but not on any of the register values. Each SHS is reset to a location-specific initial value at the beginning of each basic block. The DCS of a basic block is a function of all of the SHSs after the last instruction in the block commits.

The two non-register SHSs (SHS_{pc} and SHS_{mem}), which are also located at the register file, are necessary to track the output of instructions that do not have register outputs. Jumps and branches write their result SHSs to SHS_{pc}. A store, however, cannot simply write its output SHS to SHS_{mem}, because it would overwrite the histories of all prior stores in the same basic block. Instead, SHS_{mem} is updated by computing a hash of the prior SHS_{mem} and the store's output SHS. SHS_{mem} does not track dataflow through memory[1], but ensures that operands are delivered correctly to the memory system. Memory updates and reads themselves are checked by the memory checker (Section 3.4).

SHSs accompany their corresponding data throughout the processor. Each register has an attached SHS_{reg}, and SHS wires or latches are attached to every part of the datapath that carries operands, including the operand bypass network. Each functional unit contains an SHS computation unit that computes the new output SHS as a function of the instruction's operands' SHSs and an ID of the operation type. The instruction's result SHS is then written back to the register file or SHS_{mem} or SHS_{pc}, depending on the type of instruction.

In Argus-1, all signatures (SHSs and DCS) are 5 bits wide. The 5-bit signature size is the smallest that allows a unique initial value for each of the OR1200's 32 registers, and it is also a convenient size for purposes of embedding the DCS in the binary (discussed later). History updates are computed using CRC5 as a hash function. Unlike our original dataflow checker [15], Argus-1 does not compute the DCS incrementally; instead, the SHS_{reg}s (160 bits total) are organized as one wide register, such that all SHS_{reg}s can be read or reset to initial values in parallel. The DCS computation is performed by first running the SHSs through a hard-wired bit permutation and then sending them through an XOR tree that computes the final 5-bit DCS. The SHS bits are permuted to make the DCS not only dependent on the set of SHSs in the register file, but also on the assignment of SHSs to registers.

Signature Embedding. Signature instructions embedded into the program cause performance degradation because they increase cache pressure and consume processor cycles. To reduce these effects, we minimized the number of embedded Signature instructions by storing DCS bits in unused instruction bits, which are common in fixed-size RISC instruction formats. Actual

1. Tracking dataflow through memory with signatures would likely require an SHS for every memory location.

Signature instructions (NOPs) are embedded only in basic blocks with insufficient unused bits.

The DCSs are added to basic blocks in three distinct phases as part of program compilation and linking. In the first phase, empty Signature instructions are added to basic blocks with insufficient unused bits to embed the DCSs. In the second phase, the DCSs of all blocks are computed. In the third phase, the legal successor blocks are determined and the DCSs are embedded into the binary. In Argus-1 this process is part of the compiler tool-chain. Instead, it could also be performed by a static binary rewriter or dynamic compilation. Argus-1 does not support execution of unprotected code, but necessary modifications would be minor.

Indirect Branches. Indirect branches complicate control flow checking, because of the difficulty in determining the legal successor blocks. Indirect branches are usually the result of *switch* statements, function pointers, or function returns. To minimize cost, Argus-1 expects the DCS to be stored in the 5 most significant bits of the register containing the branch target address. This solution is sub-optimal in that it reduces the range of addressable targets, but it is necessary to minimize Argus-1's cost. For switch statements and function pointers, the target DCSs are embedded into all entries in the jump table and function address constants in the binary. The DCS for a function return is provided by writing it into the link register when the function is called. To facilitate this, basic blocks that end in a function call contain two DCSs, although they have only one legal successor: one for the first block in the called function and one for the block specified in the link register (link DCS). Because the DCS is tied to the link register, it is automatically saved and restored during nested function calls along with the link address.

Data Value Correctness. Beyond checking the shape of the dataflow graph, the dataflow checker also has to ensure that data values are transmitted correctly. To detect errors in data values, Argus-1 adds a parity bit to each register and each part of the datapath that carries an operand or instruction result.

Checking Liveness. There are many techniques for checking liveness, including simple watchdog units [13]. Our watchdog has a 6-bit counter. At every clock cycle, the counter is reset if the pipeline is not stalled, and it is incremented when the pipeline is stalled. When the counter saturates (i.e., after 63 consecutive stall cycles), the watchdog indicates an error. To bound the time between control flow checks, Argus-1 also requires a fixed limit on the size of basic blocks.

3.3. Computation Checker

The computation checker consists of several functional unit (FU) sub-checkers. We illustrate in Figure 3 how Argus-1 is integrated into a FU. For each FU, we have a sub-checker and a SHS computation unit. The opcode, from which the operation ID is derived, is distributed in a way that makes it impossible for a single fault to cause the same incorrect opcode to be delivered to both the FU and sub-checker while the correct opcode is delivered to the SHS computation unit (see Figure 3). In the remainder of the section, we describe the sub-checker for each FU in Argus-1.

3.3.1 ALU Sub-Checker

The primary component in an ALU is the carry look-ahead adder, and there have been many prior schemes for detecting errors in these adders. We previously developed a low-cost adder sub-checker [33], which has about the same delay as a carry-lookahead adder and roughly the same area as a ripple-carry adder. In Argus-1, we slightly enhance the adder sub-checker such that it can also check the bitwise logical operations. The enhanced sub-checker emulates the logical operations by multiplexing the appropriate inputs. For example, a full adder acts as an XOR if we tie its carry-in to 0.

We check the ALU's shift and sign-extension units using a single unit that performs a right-shift followed by a sign-extension (RSSE). This RSSE organization also enables us to check the alignment and sign-extension of sub-word loads and stores. To check a right-shift of *A*, the RSSE replays the shift (and sign-extension, if it was an arithmetic shift) and compares to the ALU's result. To check a left-shift of *A*, the RSSE shifts the ALU's result back to the right and then compares to *A* (while masking out those bits that had been left-shifted off the end of the word). The RSSE checks sign-extension instructions by shifting them zero bits to the right and replaying them with the sign extender.

3.3.2 Multiplier/Divider Sub-Checker

Multiplication is checked using a well-known modulo arithmetic approach [24]. Assume the two inputs are *A* and *B,* and the modulus is *M*. The sub-checker

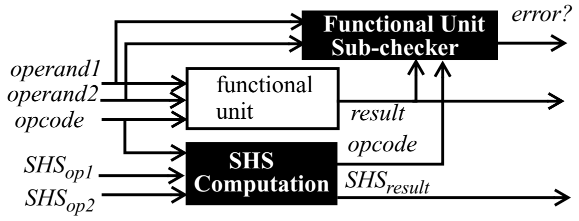

Figure 3. Functional Unit with Error Detection.

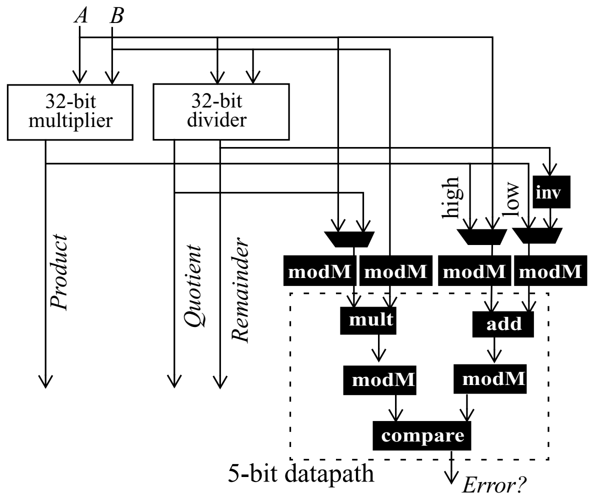

Figure 4. Multiplier/Divider Sub-Checker

verifies that *[(A mod M)*(B mod M)] mod M = Product mod M.* For small *M*, the sub-checker multiplier is much smaller than the original multiplier. To facilitate efficient modulo computation in hardware, we choose *M* to be a Mersenne number (2^5-1=31).

Argus-1 checks division with the same hardware. If *A/B=Quotient+Remainder/B*, then *B*Quotient=A-Remainder*. We use the same modulo arithmetic to check that *[(B mod M)*(Quotient mod M)] mod M= [(A mod M)-(Remainder mod M)] mod M*. To check division and multiplication with the same logic requires muxes for choosing the inputs to the modulo units and logic to negate the remainder. We illustrate this combined sub-checker in Figure 4.

Modulo checkers have a small probability of aliasing, in which a faulty computation is undetected. This probability can be made arbitrarily small by increasing *M*, at the cost of a larger multiplier in the sub-checker.

3.4. Memory Checker

The memory hierarchy is composed of the core's load-store unit, caches, and main memory. Argus-1 detects address computation errors using an adder checker like the one described in Section 3.3. Argus-1 also detects errors in data re-alignment in byte and half-word (16-bit) loads and stores using the RSSE sub-checker. In the OR1200 core, the ALU and the load-store unit are not used simultaneously, so there is no contention for the ALU checker.

To detect data corruption, Argus-1 adds parity to each word in the data cache and memory (assuming ECC is not already present). Argus-1 does *not* add parity to the instruction cache, because errors in instructions will cause incorrect control flow and/or dataflow, which will be detected during the DCS comparison.

Argus-1 also protects against errors that cause a load or store to access the wrong word despite providing a correct address. To detect these errors, Argus-1 embeds the physical address along with the data in the caches and memory. If the core wants to store value *D* to address *A*, it actually stores the value $D_A = D\ XOR\ A$ (into address *A*). Parity is computed over *D* and stored along with the data. When the core loads from address *A*, it takes the value obtained from memory, D_A', and XORs it with *A* to obtain *D*'. In the error-free case, *D'* will equal *D* (the value that the core wanted to store to address *A*). If a single-bit error occurs in either the address or data, then *D'* will not equal *D* and their parities will differ, indicating an error.[2]

Argus-1 cannot detect errors that cause cache accesses to be silently not performed. These errors are equivalent to memory ordering violations, in that they cause loads to get values from the wrong stores. In the OR1200 core, there are two concrete scenarios for such errors: an access that misses in a cache even though it should have hit and a store issued to the cache that does not perform the actual write. The former error scenario can be addressed using redundant tag comparisons and parity on the tags. The latter error scenario could be detected by following each store with a load to the same address. We decided against this option due to power consumption and performance considerations.

4. Experimental Evaluation

The goals of our evaluation are to confirm Argus-1's error detection coverage (Section 4.1) and error detection latency (Section 4.2), and to determine its area and performance overheads (Sections 4.3-4.4).

4.1. Error Detection Coverage

An Argus implementation with perfect checkers can detect all possible single-error scenarios (and many multiple-error scenarios) in the non-exceptional part of the core. However, due to physical constraints, the Argus-1 implementation has sacrificed some small amount of error coverage. Argus-1 cannot detect the following errors:

- Some memory access errors (see Section 3.4)
- Errors that are hidden by DCS aliasing
- Errors that are hidden by aliasing in the modulo-based multiplication/division sub-checker

2. The I/O controller removes the embedded address from the data sent to I/O devices. Partial stores can use the same techniques used in other systems with per-word EDC, usually read-modify-write.

TABLE 1. Error Injection Results

Error Type	unmasked, undetected (silent corruption)	unmasked, detected	masked, undetected	masked, detected (DME)
transient	0.76%	37.4%	38.2%	23.7%
permanent	0.46%	37.6%	38.2%	23.7%

- Errors in unprotected areas: Argus-1 only covers the portion of the core associated with user-level sequential program execution.
- Some multiple-error scenarios: Argus-1 cannot detect when one error causes the core to execute incorrectly and another error in the corresponding checker logic prevents detection.

We performed error injection experiments to empirically test Argus-1's error detection coverage. The experiments were performed while the core was running a "stress-test" microbenchmark that involves a broad range of registers and instruction types. It would have been difficult to test Argus-1 using benchmark code, because many benchmarks have frequently executed inner loops that use only a handful of registers and a small subset of the instruction set. We injected single transient and permanent bit-inversion errors in all portions of the microprocessor core, including the features added for Argus-1. From among the roughly 40,000 total gates, we randomly sampled 5,000 gate outputs on which to inject bit flips. We did not inject errors in the caches, but we did inject errors in the core's interface to memory. Because most transients are normally masked, when we activate a transient error, we wait until either it affects the OR1200's architectural state or until a fixed amount of time has elapsed (in which case we consider the error to be masked). For a permanent error, we consider it to be masked if it has no impact before the fixed amount of time elapses.

For each error injection experiment, we classified its result along two axes. First, was the error detected? Second, was the error masked? The errors that we want most to avoid are unmasked, undetected errors, which represent silent data corruptions. In Table 1, we show the results of these experiments.

4.1.1 Unmasked Errors

Most importantly, we observe that silent data corruptions are extremely rare, compared to detected errors. Of the unmasked transient and permanent errors, Argus-1 detects 98.0% and 98.8%, respectively. Examining these few undetected errors in detail, we noticed that the majority of them are in gates that affect multiple bits in the datapath; these errors flip an even number of bits and are thus undetectable using parity. Some other undetected errors are due to aliasing in the multiplier modulo checker and DCS.

We examined which parts of Argus-1 were responsible for detecting each error. The computation checkers were responsible for 45% of detected errors. The next largest contributor to error coverage was parity (on operands, registers, and load values), which caught 36% of detected errors. The DCS comparison caught 16%, and the watchdog caught 3%. These results confirm that a composition of all checkers is necessary in order to achieve good coverage.

4.1.2 Masked Errors

Before we discuss the results, we first note that the results are identical for transient and permanent errors, because of the way that we inject them (i.e., we activate a transient until it shows up or until the experiment ends, in which case it is equivalent to a masked permanent error). We observe that a large fraction of injected errors are masked, which is not surprising [32]. All errors in Argus-1 hardware are masked, because they have no impact on the OR1200 core's execution. One other class of masked errors is the set of errors that impact only the most significant 32 bits of the multiplier's 64-bit result. These bits are accessed only by the multiply-accumulate instruction, which is not used by any of our benchmarks. Therefore, we did not include it in our "stress-test" microbenchmark.

Of the masked errors, Argus-1 detects 38.3%. A detected masked error (DME) leads to a recovery and, for a transient error, is not necessary; however, detecting permanent errors is important, so that we can potentially take action to address them. Many of these DMEs are in Argus-1 hardware itself. DMEs only affect performance, and we would rather incur some DMEs than silent data corruptions. To confirm that Argus-1 never incurs "false positives" (i.e., detects errors that did not occur), we also performed experiments in which we injected no errors. Argus-1 never reported an error in these experiments.

4.2. Error Detection Latency

Argus-1 detects most errors soon after they occur. Errors that affect control flow are either detected at the end of the current or subsequent basic block, depending on whether the error affects intra-block or inter-block control flow, respectively. Errors in computation (ALU, multiplier/divider) are detected in the cycle after the erroneous computation. Errors in dataflow are

TABLE 2. Area Overhead. Areas in mm^2.

	OR1200	With Argus-1	Overhead
core	6.58	7.67	16.6%
I-cache: 1-way	2.14	2.14	0%
2-way	2.42	2.42	
D-cache: 1-way	2.14	2.24	4.9%
2-way	2.42	2.54	5.1%
total: 1-way	10.86	12.05	10.9%
2-way	11.42	12.63	10.6%

detected at the end of the current basic block. Errors in interactions with memory are detected either the cycle after the error (for incorrect address computation in the load-store unit) or when a load accesses a cache block whose parity signifies an error. The latter case has an arbitrary long error detection latency, which is common to all EDC based schemes. Detection latency can be bounded by using cache and DRAM scrubbing, but will still be much higher than Argus-1's detection latencies for other errors. Long latencies can be circumvented by using error correcting codes (ECC) instead of simple error detecting codes (EDC).

4.3. Area Overhead

A primary motivation for Argus is to use less area than existing schemes, such as core replication and DIVA. In particular, we wanted to devise a scheme that used a small fraction of the area of even a simple core.

Most of Argus-1's area is used for dataflow and control flow checking. This area overhead consists of: widening all datapaths and registers to accommodate one parity bit and 5 SHS bits per datum; CRC logic and an XOR tree to compute updated SHSs and the final DCS; and logic to extract the static DCS from the code. The various computation checkers are the second major area contributor. The remainder of Argus-1's area is for control logic, the watchdog timer, etc.

We used our CAD tools, Synopsys Design Compiler and Cadence Silicon Ensemble, to floorplan and layout the core, both with and without Argus-1. We did not include the debugging hardware or the TLB. Our CAD tools use the publicly available VTVT 0.25μm standard cell library [29]. We present our results in Table 2. We first observe that the unmodified OR1200 core uses 6.59mm^2 of area (2.565mm by 2.565mm), and the core with Argus-1 uses 16.6% more area.

To determine total chip area overhead, not just core area overhead, we used Cacti 3.0 [8] to calculate the area of the caches. Both the instruction and data caches are 8KB. Argus-1's data cache adds area for parity, but

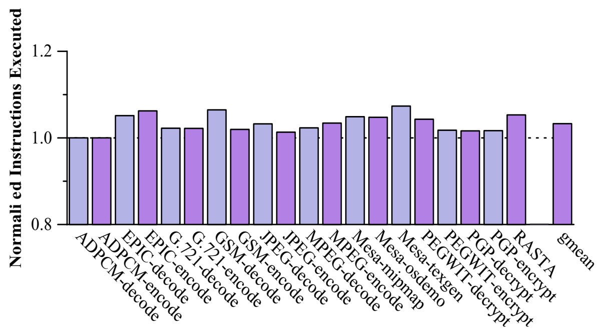

Figure 5. Dynamic Instruction Overhead

recall from Section 3.4 that Argus-1 does not need to add parity to the instruction cache. If we compare Argus-1 to an unmodified OR1200 chip (with no error detection on the core or caches), then Argus-1 consumes only 10-11% more area.

The low area overhead of Argus-1 suggests that it has a fairly low power overhead, but we do not have reliable power analysis at this time. We plan to quantify Argus-1's power overhead in the future.

4.4. Performance Overhead

Argus-1's error detection hardware does not cause any pipeline stalls or delay instruction retirement, because Argus-1 is designed to invoke backward error recovery [27] once an error is detected. Our CAD tools also showed no increase in any critical paths due to Argus-1 logic and thus we do not expect changes in clock cycle time. Hence, Argus-1's only potential impact on core performance is due to having Signature instructions embedded in the instruction stream when insufficient unused bits are available to store the DCSs. Signature instructions consume instruction cache space as well as fetch and decode bandwidth. We use the OR1200 simulator and the MediaBench benchmark suite [12] to analyze how this impacts performance. We assume a memory configuration typical for an embedded system; the data and instruction cache are each 8KB, hits take 1 cycle, and misses take 20 cycles.

In Figure 5, we plot the dynamic instruction count overhead. On average, the overhead is 3.5%. Dynamic instruction count overhead is generally lower than static instruction count overhead (which is 7% on average), because the frequently executed inner loops of the benchmarks contain mostly arithmetic and logic operations with sufficient unused bits to embed DCSs. Initialization code as well as function prologues and epilogues—which consist mostly of loads, stores, and operations involving immediates—have few unused bits and therefore require the embedding of Signature

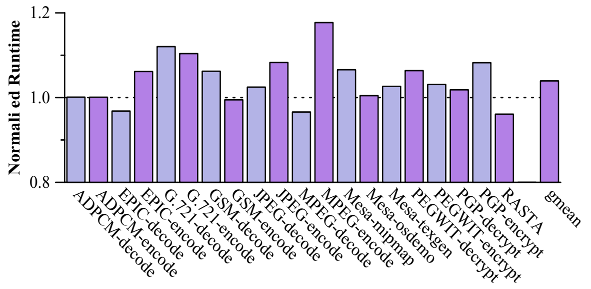

Figure 6. Runtime Overhead (1-way cache)

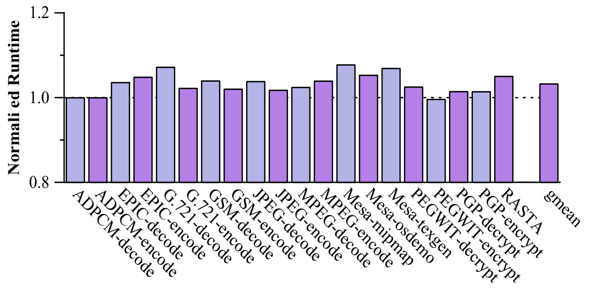

Figure 7. Runtime Overhead (2-way cache)

instructions to store DCSs, but are executed less frequently than the inner loop bodies.

In Figure 6 and Figure 7, we plot Argus-1's performance impact for a direct-mapped and 2-way instruction cache, respectively. On average, the runtime overheads are 3.9% and 3.2%. Argus-1's performance impact has two components: the cost of processing more dynamic instructions and an increased code footprint. The first component of the overhead depends linearly on the increase in the number of dynamic instructions, but runtime overhead from executing additional instructions is generally lower than the dynamic instruction count overhead itself, because all added Signature instructions (NOPs) execute in 1 cycle whereas an average instruction takes 1.1-1.7 cycles.

Argus-1's second performance component, due to the increased code footprint, is far less predictable and highly benchmark specific. The increased code footprint can increase instruction cache capacity misses, but the relationship is not linear. As a secondary effect, the insertion of Signature instructions also causes a re-alignment of basic blocks, which can randomly reduce or increase the number of conflict misses. The average effect of this re-alignment will be zero, but it can have tremendous impact on individual benchmarks, including speed-ups on several benchmarks with Argus-1 and a direct-mapped cache. The 2-way set-associative cache is less sensitive to re-alignments than the direct-mapped cache, which explains the lower variation in runtime overhead in Figure 7.

5. Related Work

There is a long history of research in error detection. Because we have already discussed checkers for control flow, dataflow, and computation, we do not mention them again in this section.

Redundant cores. Replicating a core provides a conceptually simple mechanism for detecting errors. This approach is, in terms of hardware and power, prohibitively expensive for commodity hardware.

DIVA. DIVA [1, 31, 2] checks similar invariants as Argus, but its approach is quite different. DIVA uses *C* simple checker cores to detect errors in a *C*-wide superscalar processor. DIVA is an excellent, low-cost design option for protecting large cores with simple decode logic. For example, a DIVA checker is only 6% of an Alpha 21264 core [31]. However, for simple, small cores, there is little opportunity to make the checker cores smaller than the cores they are checking, such that DIVA becomes almost indistinguishable from using redundant cores.

Redundant multithreading (RMT). There are many varieties of redundant multithreading schemes [23, 21, 16], but they all share the goal of using otherwise idle thread contexts to provide redundancy in SMT cores. RMT has three significant costs: the performance loss due to thread contention (estimated at 30% [16]), the opportunity cost of using threads for redundant computation instead of useful work, and the energy consumed by the redundant threads. RMT also has the implicit cost of requiring an SMT core and cannot detect permanent errors in non-replicated units.

BulletProof. The BulletProof pipeline [25] uses built-in self-test to detect and diagnose (isolate) 89% of permanent faults, but it cannot detect transient errors. BulletProof increases the area of a 4-wide VLIW core (excluding caches) by 9.6%. This overhead is likely to be greater for a single-wide core because BIST tables and other checker hardware singletons cannot be amortized over multiple instances of the units they check. BulletProof has the advantage of lower performance impact, no required software changes, fewer possibilities for false negatives due to aliasing, and more precise diagnosis. Argus's main advantages are its ability to detect transient errors and lower detection latency.

Commercial Examples. There are fault-tolerant commercial microprocessors that both detect and correct core errors, and we discuss two representative examples. The LEON-FT microprocessor [6] and the IBM S/390 G5 microprocessor [26] use error codes for large storage structures. The LEON-FT uses triple modular redundancy (TMR) for each flip-flop, and has a total area overhead of roughly 100% [6]. The S/390 G5 microprocessor completely replicates large portions of the core, including the I-unit (fetch and decode) and E-unit (execution and register file), which adds considerable overhead.

Software redundancy. Software replication of instructions can be used to detect transient hardware errors [18, 22] at the cost of a 50% slowdown for an out-of-order processor [22] and high energy consumption. The performance loss for a simple in-order core would be roughly 100%, due to the lack of idle slots in which to execute redundant instructions.

Error detecting/correcting codes. Error codes are excellent at detecting errors in storage and messages. Certain codes can also check computations. However, error codes are not applicable to general logic.

6. Conclusions

The goal of this research was to develop low-cost error detection for simple processor cores. We expect simple cores to remain popular for many embedded applications as well as for multicore chips, and it is important that we can detect errors in these cores. We believe that Argus provides a viable, efficient solution to this problem. The key to Argus's efficiency is that it checks invariants instead of components. Invariant checking also makes it is easier for us to formally reason about Argus's error coverage capability.

The Argus-1 implementation shows the potential of the Argus approach. Its area and performance costs are quite low, particularly when compared to previous schemes for detecting errors in simple cores. We also believe that, with additional tuning of the implementation, we could further improve its error coverage and reduce its costs.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCR-0444516, the National Aeronautics and Space Administration under grant NNG04GQ06G, and an equipment donation from Intel Corporation. We thank Fred Bower, Derek Hower, Alvy Lebeck, Anita Lungu, and Bogdan Romanescu for feedback on this work. We thank Bogdan Romanescu and Heather Sarik for help with the experiments.

References

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Int'l Symp. on Microarchitecture*, Nov. 1999.

[2] T. M. Austin. DIVA: A Dynamic Approach to Microprocessor Verification. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[3] L. Carter, J. Feo, and A. Snavely. Performance and Programming Experience on the Tera MTA. In *Proc. of the SIAM Conf. on Parallel Processing*, Mar. 1999.

[4] Cisco Systems. Cisco Carrier Router System. http://www.cisco.com/application/pdf/en/us/guest/products/ps5763/c1031/cdcco% nt_0900aecd800f8118.pdf, Oct. 2006.

[5] X. Delord and G. Saucier. Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors. In *Proc. of Int'l Test Conf.*, 1991.

[6] J. Gaisler. A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, June 2002.

[7] Int'l Technology Roadmap for Semiconductors, 2003.

[8] N. P. Jouppi and S. J. Wilton. An Enhanced Access and Cycle Time Model for On-Chip Caches. DEC WRL Research Report 93/5, July 1994.

[9] S. Kim and A. K. Somani. On-Line Integrity Monitoring of Microprocessor Control Logic. In *Proc. of the Int'l Conf. on Computer Design*.

[10] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[11] D. Lampret. OpenRISC 1200 IP Core Specification, Rev. 0.7. http://www.opencores.org, Sept. 2001.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of the 30th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, p. 330–335, Dec. 1997.

[13] A. Mahmood and E. McCluskey. Watchdog Processors: Error Coverage and Overhead. In *Proc. of the 15th Int'l Symp. on Fault-Tolerant Computing Systems*, p. 214–219, 1985.

[14] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, June 2006.

[15] A. Meixner and D. J. Sorin. Error Detection Using Dynamic Dataflow Verification. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[16] S. S. Mukherjee et al. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, p. 99–110, May 2002.

[17] M. Nicolaidis. Efficient Implementations of Self-Checking Adders and ALUs. In *Proc. of the 23rd Int'l Symp. on Fault-Tolerant Computing Systems*, p. 586–595, June 1993.

[18] N. Oh et al. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Trans. on Reliability*, 51(1):63–74, Mar. 2002.

[19] J. H. Patel and L. Y. Fung. Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. *IEEE*

Trans. on Computers, C-31(7):589–595, July 1982.

[20] J. H. Patel and L. Y. Fung. Concurrent Error Detection in Multiply and Divide Arrays. *IEEE Trans. on Computers*, C-32(4), Apr. 1983.

[21] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proc. of the 27th Annual Int'l Symp. on Computer Architecture*, p. 25–36, June 2000.

[22] G. A. Reis et al. SWIFT: Software Implemented Fault Tolerance. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, p. 243–254, Mar. 2005.

[23] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of the 29th Int'l Symp. on Fault-Tolerant Computing Systems*, p. 84–91, June 1999.

[24] F. F. Sellers et al. *Error Detecting Logic for Digital Computers*. McGraw Hill Book Company, 1968.

[25] S. Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proc. of the Twelfth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[26] T. J. Slegel et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, p. 12–23, March/April 1999.

[27] D. J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, May 2002.

[28] J. Srinivasan et al. The Impact of Technology Scaling on Lifetime Reliability. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, June 2004.

[29] J. B. Sulistyo et al. Developing Standard Cells for TSMC 0.25um Technology under MOSIS DEEP Rules. Technical Report VISC-2003-01, Dept. of Electrical and Computer Engineering, Virginia Tech, Nov. 2003.

[30] N. J. Warter and W.-M. W. Hwu. A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking. In *Proc. of the 20th Int'l Symp. on Fault-Tolerant Computing Systems*, p. 442–449, June 1990.

[31] C. Weaver and T. Austin. A Fault Tolerant Approach to Microprocessor Design. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, July 2001.

[32] C. Weaver et al. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proc. of the 31st Annual Int'l Symp. on Computer Architecture*, p. 264–275, June 2004.

[33] M. Yilmaz, A. Meixner, S. Ozev, and D. J. Sorin. Lazy Error Detection for Microprocessor Functional Units. In *Proc. of the IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, Sept. 2007.

Appendix A: Proof of Argus Completeness

We consider program execution on an abstract von Neumann machine with a finite set of registers R and memory locations M, no I/O, and no interrupts or exceptions. This machine executes a program, which is a sequence of instructions. The machine's ISA maps each instruction to a specification that defines an n-tuple of input addresses, an n-tuple of output addresses, and one function f for each output. Immediate values are part of the function definitions. The abstract machine executes one instruction per timestep.

Note 1. For simplicity, we look at the program as the linear sequence of instructions after all data-dependent branches have been resolved. In reality this sequence is not known *a priori*, but we can use it to construct the equally abstract correct execution. Any physical checker or processor must determine the correct sequence using state information and the program code before extracting instruction specifications from it.

We represent program execution with a graph that describes the machine state and executed instruction at each timestep. It has the following vertices:

- A *state vertex* s_t for each timestep t for each register and memory location. R_t and M_t are the sets of register and memory vertices at timestep t. The processor state P_t is the union of R_t and M_t.
- A subgraph per instruction that has vertices for each input and each output of the instruction ($I_{t,n}$,$O_{t,n}$).

Each vertex v_t in the graph is annotated with a value $V(v_t)$ and an address $A(v_t)$. $V(v_t)$ represents the data stored in the corresponding location or instruction input or output at timestep t. The address of a storage location is a constant that is unique to the location. The addresses of instruction inputs and outputs are part of the instruction specification. Register input and output addresses are specified as constants. Addresses of memory inputs and outputs are functions of register input values.

Given an initial assignment of values to vertices at *t=0* (*initial state*) and a program, we can construct the unique value assignments for a *correct execution*. Because there is no I/O, interrupts or exceptions, all values depend only on initial state and the program. The value assignments for timesteps *t>0* are derived by iterating over timesteps using the following algorithm.

For every timestep t we first select the t^{th} instruction in the program sequence and determine its specification. Based on the specification, we add the following edges to represent data propagation (*data propagation edges*) from timestep t to t+1:

- An edge to each instruction register input $I_{t,n}$ from the state vertex s_t with the same address in P_t
- An edge from each instruction register output $O_{t,n}$ to the state vertex with the same address in P_{t+1}
- An *identity edge* between each vertex in P_t and the vertex with the same address in P_{t+1}if the vertex in P_{t+1} is not connected to an instruction output. These edges represent unmodified state.

Values are then assigned based on the edges; each instruction input is assigned the value of the vertex s_t to which it is connected, each instruction output $O_{t,n}$ is assigned $f_{t,n}(V(I_t))$, and all vertices in P_{t+1} are assigned the value of the vertices to which they are connected. Then memory addresses are computed using the input and output address functions. Edges for memory inputs and outputs are added in the same way as edges for registers, and values are propagated to and from memory vertices.

This algorithm uniquely defines value assignments for all vertices in the graph. Using the value assignments and data propagation edges, we can now define the conditions monitored by the checkers.

Control Flow Checker (CFC). The tth instruction is executed (i.e., the machine is live) and its specification is identical to the specification of the tth instruction in the program sequence according to the ISA.

Data Flow Checker (DFC). Dataflow checking is split into two separate conditions; one to ensure that the edges are correct (i.e., dataflow graph has the correct shape) and one to ensure that values are propagated correctly across these edges.

Shape (DFC_S). For all instructions, each register input is connected to a vertex in R_t and each register output is connected to a vertex in R_{t+1}. There is an edge to each vertex in R_{t+1} that is not connected to an instruction output; this edge's other end is a vertex in R_t. For any data propagation edge, the addresses of the two register vertices it connects are identical.

Value (DFC_V). Any two vertices connected by a data propagation edge are assigned the same value.

Memory Flow Checker (MFC). MFC checks all the same conditions as DFC, except that they apply to memory rather than registers. MFC_S further checks that memory address functions are evaluated correctly.

Computation Checker (CC). The value assigned to the nth output of the tth instruction is equal to $f_{t,n}(V(I_t))$.

Proof: Every execution that meets all checker conditions assigns the same state values as the correct execution.

We prove this by induction.

Base case ($t = 0$): For purpose of the proof, we assume that the *initial state* (value assignments at t=0) is checked using an external checksum mechanism (e.g., a checksum over all the initialized program data and register values that can be compared to the program binary). In practice this requirement is satisfied by having initial EDC values for each register and memory location.

Induction step ($t \rightarrow t+1$): Assume that all values at timestep t match the correct execution. CFC ensures that the specification of the tth instruction matches the instruction specification in the program and thus the correct execution. DFC_S ensures that the instruction register inputs are connected to the vertices with the same addresses in P_t and that the register outputs are connected to the vertices with the same addresses in P_{t+1}. As addresses are unique and specifications identical, the resulting edges must be the same as those in the correct execution. Because the values at time t are identical to the correct execution and the instruction inputs are connected to the same vertices in both executions, DFC_V ensures that the values of the input vertices are also the same. MFC_S ensures that, given identical register input values, the memory input and output addresses are also identical in both executions. Based on these addresses, MFC_S and MFC_V ensure that memory input values are identical in both executions in the same way DFC_S and DFC_V do for register input values. As a consequence of all input values being identical, CC ensures that the values of all output vertices are identical to the correct execution.

We can now show that the value of each vertex in the execution at t+1 is the same as the value in the correct execution. Each vertex is either connected to an instruction output or not:

Case 1: Vertex connected to instruction output. We showed above that both the value of each output and all edges are the same in both executions. Thus, the vertex must be connected to an output in both executions and, because of DFC_V and MFC_V, its value must match the output's value. Because the output has the same value in both executions, the vertices must also have the same value.

Case 2: Vertex not connected to instruction output.

Because the vertex is not connected to an instruction, there is an implicit edge to the vertex with the same address in P_t, and because edges are the same in both executions, this is also the case in the correct execution. By the induction assumption, the vertex has the same value in both executions at timestep t and therefore (by DFC_V or MFC_V) must also have the same value in both executions at timestep t+1.

Thus, in both cases every vertex has the same value in both executions at timestep t+1.

End of Proof

Note 2. In a real implementation of a processor, each timestep corresponds to committing an instruction. P_t is the architected state before committing the tth instruction. As long as all checkers ensure the conditions for updates of architected state described above, it does not matter if they do not detect incorrect microarchitectural state (buffers, latches, etc.). Any non-masked error in non-architectural state will appear as a violation of these conditions for architected state.

Definition: Block-based checking. In a block-based execution, register and memory vertices exist only for timesteps that mark the end of a block of instructions. Block-based execution can trivially be derived from regular execution by removing all intermediate register and memory vertices and connecting their inputs transitively to the next instruction input or state vertex on a block boundary. The major differences between these block-based executions and regular executions are that there are now multiple instructions between state vertices and instruction inputs can be directly connected to instruction outputs. For these edges, the requirements for equal addresses and data of connected vertices still apply, but we further require that an input can only be connected to the output with the highest timestep t out of all outputs with the same address and timesteps smaller than the input's timestep. This requirement must be added to the conditions for DFC_S and MFC_S.

With these modifications, we use the same inductive proof as before to show that the values assigned to all state vertices at the end of each block are the same for the correct execution and an execution that meets all checker conditions. CFC, DFC_S, and MFC_S still ensure that instruction specifications and input/output edges for the block are correct. In case 1, we still argue that the final instruction output must be correct because all inputs must be correct by recursively applying the argument to inputs connected to other instruction outputs. The argument in case 2 is unchanged.

Appendix B: Proof of Argus-1

Proof: Argus checkers, except MFC_S, check all conditions of ideal checkers in block-based execution (under the assumption of error-free checking mechanisms and aliasing-free checksums/signatures):

CFC. The watchdog component of CFC ensures that only a finite amount of time passes between executed instructions. CFC further enforces a finite bound on the number of instructions in a block. Thus, each block is reached and executed within finite time.

Within each block, CFC uses a signature (DCS) that describes the sequence of instruction specifications in that block. If any of the instruction specifications within the block do not match the program, the computed DCS will differ from the DCS that the program specifies for that block. This discrepancy will cause an error to be detected, if CFC selected the correct block DCS from the program (recall Note 1).

Before executing each new block, CFC selects the DCS based on the current processor state. The state before executing the first instruction of a basic block is known to be error-free (otherwise the error would have been detected and the execution terminated). Thus, the error-free checker uses error-free state to decide on the next DCS to use, which must hence be the correct one.

DFC_V. Value checking is performed by assigning a checksum (EDC) to each vertex (i.e., storage location). These checksums are part of the checker mechanism and therefore presumed to be error-free. If a data propagation edge exists between two vertices, their EDCs must be identical (as EDC propagation is error-free). Thus, if the values of the vertices do not match, an error will be raised, because the (equal) EDCs cannot match the (unequal) values.

DFC_S. Dataflow checking assigns an SHS to each vertex and a DCS, which contains the SHSs for all registers, to each block. Ideal SHSs uniquely describe the subgraph derived from recursively following all incoming data propagation edges to the register and state vertices at the beginning of the block. The DCS therefore describes all data propagation edges in the block. The static DCS embedded in the program describes the subgraph in the correct execution. In the correct execution, all conditions of DFC_S are satisfied by construction. All DFC_S conditions describe edge properties or address assignments. Addresses are static for state vertices and checked by CFC for instructions and thus known to be error-free. Thus, to violate any of the conditions, at least one edge in the block must differ from the correct execution. But if any one edge differs from the corresponding edge in the correct execution, then the computed DCS, which fully describes all edges, must also differ from the DCS embedded in the binary and an error will be detected.

Like CFC, DFC_S is dependent on picking the correct signatures. In Argus-1, this is assured by CFC, which chooses the signatures for both DFC_S and itself.

MFC_V. Same as DFC_V.

MFC_S. MFC_S directly checks address computation and annotates each vertex with its address (by embedding it into the value/checksum) and will therefore detect address errors in the same way as data errors. However, it cannot detect absence of edges.

CC. Our computation checkers differ slightly from each other, but in principle they all perform a redundant (and presumed error-free) computation of each output from the input values and compare it to the corresponding observed output. If the observed output does not match the correct function value, it will also differ from the redundantly computed output and an error will be detected.

End of Proof