Duke University
Department of Electrical and
Computer Engineering

Senior Honors Thesis
Spring 2008

Proving the Completeness of Error Detection Mechanisms in Simple Core Chip Multiprocessors

Michael Edward Bauer
Advisor: Dr. Daniel J. Sorin
Submitted: April 22, 2008

Abstract

As transistor sizes have continued to scale down, the rate of both soft and hard errors occurring in processors has continued to grow. Ultimately these effects will come to dominate program execution, making it impossible to execute programs correctly without some form of error detection mechanism. Traditionally error detection mechanisms have relied on hardware replication or software redundancy to check for errors at a high cost in area, power, or performance. Recently, however new mechanisms have been developed to dynamically perform error detection in both single core and multicore processors with low area and performance overhead. This thesis proves the correctness and completeness of these error detection mechanisms, thereby demonstrating that they can comprehensively detect all errors occurring in chip multiprocessors composed of simple cores.

Contents

1 Introduction

The continual scaling of transistor sizes has resulted in transistors becoming more vulnerable to soft faults [1]. In addition to this, smaller transistors are more prone to suffer from hard faults either in time or during the manufacturing process [2]. The combination of these increasing fault rates has led both chip and software designers to search out mechanisms for performing error detection during run time.

In general, most error detection mechanisms have relied on replication of some form to detect errors. DIVA presents a hardware mechanism for using a much smaller and simpler core to check the execution of a complex out-of-order core, incurring a small area and power overhead [3][4]. Active-Redundant Simultaneous Multithreading (AR-SMT) takes the approach of duplicating threads to reduce area overhead, but still incurs a performance and a power penalty [5]. While these approaches have certainly addressed the issue of detecting errors, they have done so at a high cost of area, power, and performance.

In conjunction with the increased focus on fault tolerance, chip manufacturers have been using the growing number of transistors to put many simple cores on a chip instead of a few complex cores [6]. While error detection mechanisms with small overheads on complex, out-of-order cores can be built, very few are able to check simple cores without a high overhead. For example, DIVA has minimal area and power overhead when checking a large out-of-order core. However, for smaller cores, the verifying core will not be much smaller than the simple core itself, leading to an almost 100% area and power overhead. The difficulty of checking simple cores in the emerging multicore domain has led to the creation of new error detection mechanisms for simple cores.

Argus is a novel approach developed by Meixner for comprehensively detecting errors in simple cores with minimal area, power, and performance overhead [7]. Argus makes use of the invariants present in the execution of a program in the Von Neumann computing model to check for errors. One of the primary contribution of this thesis is not the actual implementation, but the proof of Argus' correctness and completeness. For this reason only the components necessary for proving the completeness of the Argus error detection mechanism will be discussed in this work. In addition to Argus, Meixner has also developed a method for the Dynamic Verification of Memory Consistency (DVMC) models [13]. This system will be employed to check the memory system in a chip multiprocessor. Proving that composition of Argus with DVMC is complete in its error detection coverage of a chip multiprocessor with simple cores is the other major contribution of this thesis. Since the focus is again on the proof, only the details of DVMC necessary for demonstrating that all errors in a chip multiprocessor can be detected will be discussed. The primary contribution of this thesis is the demonstration that both Argus and the composition of Argus with DVMC are complete in their error detection coverage.

In this thesis, the Von Neumann model of computation is first explained to gain some insight into the invariants that will be used to check the execution of simple cores. The checking mechanism for a single simple core will then be described and the proof demonstrating its correctness and completeness will be given. Another error detection mechanism necessary for checking memory in a chip multiprocessor environment is then explained as background for the last proof. Finally, a proof that the combination of a simple core checking mechanism and a memory checking mechanism can completely detect all errors in a chip multiprocessor will be given.

2 Von Neumann Program Execution Invariants

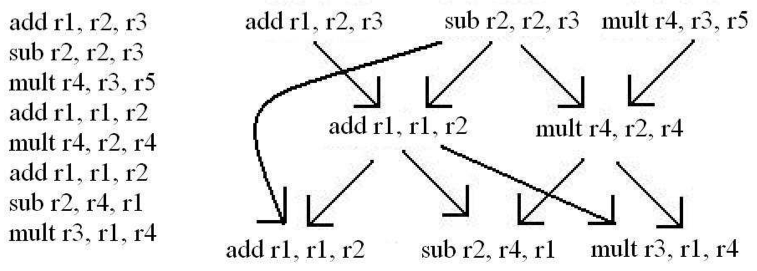
Over the course of the history of computing, the Von Neumann computer has emerged as the primary computing model in use today. Associated with the Von Neumann programming model are several invariants that are exploited by Argus to detect errors in simple cores. In order to understand how Argus makes use of these invariants, it is first necessary to define a Von Neumann programming model and its method of execution. The Von Neumann programming model consists of:

- A sequence of instructions that specify the program to be executed. Each instruction is assigned a number and each instruction appears to execute atomically.
- A program counter (PC) that specifies the current instruction that is being executed
- A machine (or computer) that is capable of executing the instructions specified in the instruction stream.
- A memory where the machine is capable of storing information and retrieving it at a later time

Having defined the Von Neumann programming model it is claimed without proof that the Von Neumann model consists completely of four invariants: dataflow, control flow, computation, and memory interaction. The proof that these four invariants completely define execution will be given later. Each of these four invariants is now discussed in detail.

2.1 Dataflow

Whenever a program is executed there are certain dependencies that exist in the program. In general there two types of dependencies: true dependencies and false dependencies [8]. True dependencies specify data dependencies where the value produced by one instruction is used as an input to a second instruction. False dependencies on the other hand are often system dependent such as two



```
add r1, r2, r3
sub r2, r2, r3
mult r4, r3, r5
add r1, r1, r2
mult r4, r2, r4
add r1, r1, r2
sub r2, r4, r1
mult r3, r1, r4
```

Figure 1: This figure gives an example of a short instruction stream and its associated static dataflow graph. True dependencies are represented by arrows between the different nodes.

instructions competing for the same ALU. Using these true dependencies the concept of a program dataflow graph can then be constructed [9].

A static dataflow graph is a directed acyclic graph that is constructed in the following manner:

- A node for every static instruction specified in the program
- A directed edge from one node to another indicating a true dependence between the two instructions

It is important to note that the complete static dataflow graph cannot be constructed in its entirety for almost all programs. This is a direct consequence of the existence of control flow instructions that can specify more than one PC following the completion of an instruction (i.e. a branch on equal instruction). Given a program's inputs, the complete dynamic dataflow graph can be created because all of these control dependencies (discussed in the next section) are resolved. However, with limited information only parts of the dynamic dataflow graph can be constructed. These smaller sections of the program's dataflow graph that are known without the program's input values are referred to as static dataflow blocks. An example of a static dataflow block and its corresponding static dataflow graph can be seen in Figure 1.

Returning to the model of Von Neumann execution, the dataflow invariant is the following: *for each static dataflow block the corresponding dataflow graph is fully defined and must be executed identically every time the program executes that particular block in order to ensure correct execution.* Given that this invariant must hold for every execution there is now a way of defining what the correct execution of static dataflow block is.

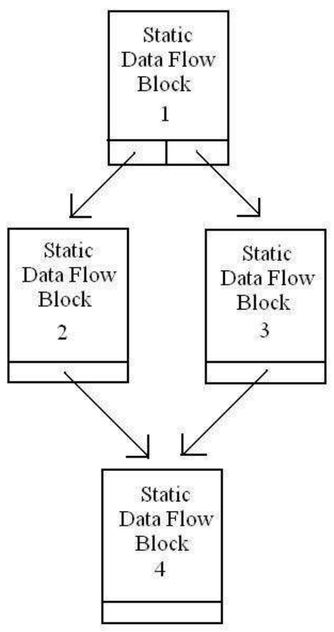


Figure 2: This figure demonstrates how control flow can specify the movement to multiple correct static dataflow blocks. In this example, block 1's set of all acceptable blocks consists of blocks 2 and 3. It should never be possible to move from block 1 to block 4.

2.2 Control Flow

As was noted in the previous section there exists a level of ambiguity with regard to generating a program's entire dynamic dataflow graph. This ambiguity arises as a result of control flow instructions that dictate how a program moves from one static dataflow graph to another. With most control flow instruction there are usually two different static dataflow blocks that are both considered to be correct destinations, as can be seen in Figure 2. In this case it is necessary to check that the program transitions from one static dataflow block to another acceptable static dataflow graph. It is important to note here that it is not necessary to check that the move is to THE correct block, but that the move is to a block in the set of acceptable blocks. Checking that the move is to the correct block within the set of all acceptable blocks is handled in the next section. The control flow invariant of Von Neumann execution is then that *all control flow statements must only result in moves to an acceptable static dataflow block.* In addition to this invariant, it is also necessary for the program to continue to make forward progress. The combination of the control flow invariant and forward progress ensures that the control flow of a program is correct.

2.3 Computation

The computation invariant of the Von Neumann programming model is much more intuitive. This invariant requires that *all computations and comparisons that constitute the execution of an instruction are carried out correctly.* It is important to note that this has ramifications for more than just basic arithmetic instructions. For example, in the case of a control flow instruction such as branch-on-equal, it is necessary that the comparison be evaluated correctly. This does ensure that the program transitions to the correct static dataflow block. In general, this invariant requires that any form of computation, either directly related to the program or indirectly related, must be executed correctly.

2.4 Memory

The last invariant of the Von Neumann programming model concerns the nature of memory accesses. This invariant holds that *each read to a specific address in memory will observe the value of the most recent write to that same memory address.* This invariant is necessary to ensure that the Von Neumann machine is capable of interacting with the memory system in a consistent manner. It is important to realize that this invariant also requires that memory state not be corrupted over time. Each read or write requires two different components of the access to be correct:

1. The memory address accessed is the one that is specified
2. The data intended to be read to or written from the specified memory address is actually read or written correctly

Both of these steps in executing a memory access must be performed correctly in order for the memory access to be considered correct. The exact reason for breaking down an access into two stages will become evident in the next section.

3 Checking Single Core Execution: Argus

In this section the checkers that are used by Argus to verify the invariants of a Von Neumann program execution are described [7]. Due to the limited scope of this thesis not all of the details regarding the actual implementation of the checkers will be described. Only those conceptual details that are necessary for proving the completeness of Argus will be mentioned. The differences between an ideal implementation of Argus and a first implementation of Argus (Argus-1) on a real OR1200 core [10] will also be described as it will be necessary for proving that Argus can be mapped onto a real world core.

3.1 Dataflow Verification

The key idea exploited by Argus for performing dataflow verification is that every execution of a program must execute the static dataflow blocks exactly as they are specified at compile time [11]. Since these dataflow blocks are known to be unique for this program, a signature can be used to represent the explicit static dataflow graph of each dataflow block. This is done by creating a State History Signature (SHS) for each architectural register in the core. For each instruction, the SHS of the destination register is then some function of the SHS values of each of the source registers and the type of operation. For example, for the instruction 'add r1,r2,r3', the SHS associated with register 1 would be updated as $SHS_{r1} = f(SHS_{r1}, SHS_{r2}, SHS_{r3}, \text{'add'})$. By applying this procedure over the course of an entire static dataflow block, Argus is then able to generate a signature representing the entire static dataflow block by computing a function of all the architectural registers' SHS's at compile time. During run time this procedure is repeated and the values are compared after the execution of each dataflow block. In addition to checking that the dataflow graph is structured correctly, an ideal implementation of Argus places error detecting codes (EDC) on every register to ensure that data is stored and transported correctly within the processor.

There are several noticeable differences between the ideal implementation of dataflow verification and the practical one. First, in an ideal implementation the SHS's could be stored in registers of infinite size. This would allow for registers that would never be prone to any signature aliasing. Instead, finite size registers are required for an actual hardware implementation. This leads to the use of hash functions that could potentially alias errors by mapping an erroneous value to the signature that is equivalent to the correct execution. In this case it is conceivable that an error might go undetected a very small probability of the time in a practical implementation. Lastly, it is infeasible to put error detection codes on every register in the system as this would result in significant area and power overhead.

3.2 Control Flow Verification

In order to check the control flow invariant of Von Neumann execution, Argus first creates a new SHS that is associated with the Program Counter, SHS_{PC}. This register holds a signature that is updated whenever a branch or a jump instruction is executed to reflect the new static dataflow block that was chosen. This value can then be checked after a finite number of static dataflow blocks have been executed in order to ensure that the control flow invariant of the program has been upheld.

In practice Argus actually combines dataflow and control flow checking by mapping the SHS for registers and the PC into a single Dataflow and Control Flow Signature (DCS). The details of this are beyond the scope of this paper, but it does allow checking at a basic block granularity with some control flow inside of the basic block instead of a static dataflow block granularity. For the remainder of this paper, all discussion will focus on the execution of basic blocks instead of static dataflow blocks.

Argus' actual implementation of control flow checking again suffers from the potential of error aliasing due to a finite register size for SHS_{PC}. This could lead to potential errors in control flow going undetected. Argus also implements a simple watch-dog timer in order to ensure that program is making forward progress which is another requirement of control flow. One potential downfall of this approach is that an error could occur in the watch-dog timer that causes it to stop monitoring program execution, allowing for the chance that the program incurs another error that halts forward progress.

3.3 Computation Verification

Checking computation is an extremely well studied field and Argus employs methods from [12] as well several other sources. The details of these checkers are beyond the scope of this paper. It is to be noted that because multiplication and division are checked using modulo arithmetic, there exists a small but non-zero probability that aliasing could occur in the checking of these operations. However, for the purposes of this thesis all computations are verified completely by the checkers in Argus.

3.4 Memory Verification

In order to verify a program's interaction with memory satisfies the Von Neumann memory invariant, an ideal version of Argus checks that the memory address that is to be accessed is correct and that data written to or read from the address is also handled correctly. To verify the first step, Argus maintains a SHS associated with memory, SHS_{mem}. This register records all of the addresses that are accessed using a signature similar to the other SHS registers. This signature is then included in the computation for the DCS for each basic block and compared with the signature generated at compile time. To verify the second step, an ideal implementation of Argus applies an EDC on every location in memory and uses the EDC to check for errors after reads. To verify that a value was written correctly, an ideal implementation of Argus executes a load to the same address that the data value was just stored to and compares the values to ensure that they are the same. By checking both of these sub-steps of a memory access, an ideal implementation of Argus is able to check the memory invariant of the Von Neumann programming model.

An actual implementation of Argus differs slightly from the above description. First, the SHS_{mem} register is again limited to finite length, introducing the small probability that aliasing will occur. Second, the ability to replay stores with a load to the same address is extremely costly in terms of performance.

For this reason, in most implementations (including Argus-1) this step will be omitted, creating the possibility that stores will silently not be written. Lastly, putting EDC on all memory locations does not guarantee that all data corruptions in memory will be detected. For example, if single error detection (parity) is placed on all locations, it is conceivable that a location receives two different bit errors over the course of a long period of time between accesses, leading to a silent error. While these are implementation details, they are still necessary for proving that an ideal implementation of Argus can be mapped onto a practical implementation and for knowing which holes exist in this mapping.

4 Proving the Completeness of Argus

In this section the completeness of Argus' error coverage is proved. The result and technique for this proof is one of the primary contributions of this thesis. The key insight is that the execution of a program in the Von Neumann programming model can be explicitly represented as a unique graph, and that each property of this graph is verified by one and only one invariant checked by Argus. After showing that an ideal implementation of Argus can completely detect all errors on simple core, it is then shown that the Argus-1 implementation with known holes described above is equivalent to Argus.

4.1 Argus Completeness

In this proof it is assumed that the system model consists of a set of finite registers which will be denoted by $\mathcal{R}$ and a finite set of memory locations denoted by $\mathcal{M}$ with no interrupts, exceptions, or I/O and purely physical memory system (e.g. no virtual address space). The program that is being executed is composed of a series of instructions that are executed sequentially, consistent with the Von Neumann model of execution. Instructions are defined as a pair of n-tuples and a function, with the first n-tuple specifying the input registers, the second n-tuple specifying the set of output registers, and the function specifying the operation to be performed on the input registers and written to the output registers.

A time step in this system is a single cycle. For purposes of this system model it is assumed that a single instruction is executed every cycle. The implementation of Argus is then able to check each instruction individually and does not have to be concerned with multiple instructions being in flight at the same time. While this assumption may appear at first unrealistic, consider that this is the appearance that all modern day processors present to the programmer as part of the Von Neumann model of execution.

Having defined the basic system model that is used to describe the program and its execution, it is now possible to construct a graph that represents the execution of a single program on the given system model. It is assumed that there

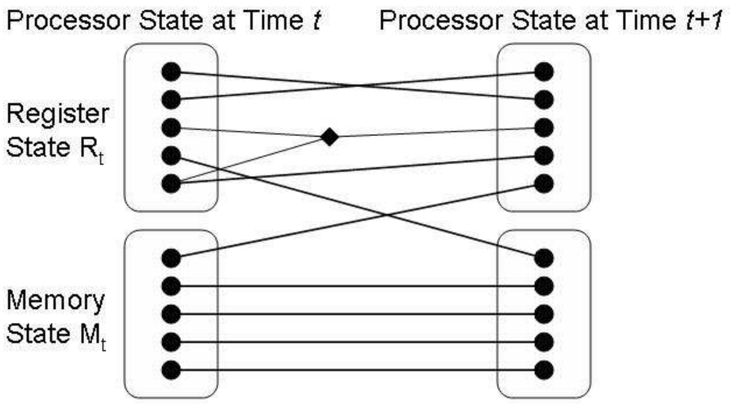


Figure 3: This figure gives an example of one step in an execution graph. Notice that in this example, multiple events are occuring as data is being both read from and written to memory. In addition to this a computation of some form is occuring as can be seen by the two edges meeting at the computation node in between the two different states.

is a single instruction stream with all data dependencies resolved. While these data dependencies actually are resolved at run time, it is easier to deal with the idea of a single abstract instruction stream, which is what is ultimately attained in any Von Neumann model of execution. The execution of the program on the system is then constructed as a graph mapping the state of the system at time t to time $t+1$. The graph can be defined as follows:

- For each time t there is a node representing the state of a register r_t or memory location m_t. $\mathcal{R}_t$ then represents the set of all registers at time t and $\mathcal{M}_t$ represents the state of all memory locations at time t. $\mathcal{S}_t = \mathcal{R}_t \cup \mathcal{R}_t$ represents the state of the entire system at time t. Each node posses two pieces of information: an address and a data value. The address associated with the node is unique to the node. The data associated with the node can be overwritten and also moved.
- A surjective mapping of edges from $\mathcal{S}_t$ to $\mathcal{S}_{t+1}$ that represents the execution of a single instruction in the system. It is important to note that although $|\mathcal{S}_t| = |\mathcal{S}_{t+1}|$ this mapping is not bijective because each register s_t can provide one or more inputs to the function of the instruction. Each node in $\mathcal{S}_{t+1}$ contains only one incoming edge. The algorithm for constructing these edges will be defined momentarily.

An example of the execution graph can be seen in Figure 3. The most important property of this graph is that it is unique to a correct program execution. Provided with an initially correct state at time $t = 0$ it is possible to iterate one step at a time and build the next state. Regardless of processor details, any machine that maintains this model of execution must generate this execution graph exactly in order to achieve a correct result. The algorithm for stepping from time step t to time step $t + 1$ is specified as follows:

1. Obtain the t^{th} instruction and decode its properties including its function, source registers, destination registers, and any memory locations with which it might interact

2. For each register in the input n-tuple, create an edge originating at that node and propagating to a functional node that performs the specified function on the n-tuple of inputs

3. For each register in the n-tuple of outputs, create an edge originating at the functional node and ending at the corresponding output register in S_{t+1}.

4. For all remaining nodes in S_{t+1} not already containing an incoming edge, create an *identity* edge that maps directly from the corresponding register in S_t. These edges handle the case of showing that some state remains the same during the course of a single instruction's execution.

Each edge is then labeled with weight equivalent to the value of the data stored in the node that the edge originates from. This data then flows along the edge and is written into the value of the node at the destination of the edge at time $t + 1$. The one exception to this is with regard to the edges that specify input or output to the functional node. These edges carry their weight either to or from the specific functional node.

An algorithm for creating a unique program execution graph has now been given. Since it is known that any deviation from this graph will result in an error, it is now necessary to show that all properties of the graph are checked at all times. The invariants checked by Argus each correspond to checking some properties of this graph:

Dataflow - The dataflow invariant discussed earlier explicitly checks two different properties of in the execution graph. First, the dataflow invariant ensures that the shape of the graph is correct with respect to the existence and placement of different edges. This is done by checking that the source of all functional input edges originate at nodes specified by the input n-tuple and that all output edges terminate at a node specified by the output n-tuple. The dataflow checker also verifies that all identity edges are correct as the SHS values associated with values not modified by the function must still be correct.

In addition to checking that the shape of the graph is correct, the dataflow checker also ensures that the data values traveling along edges between any two registers, from a register to functional node, or from a functional node to a register are all correct. This property is ensured by the presence of EDC on all of the edges. Note that this applies to identity edges as well.

Control Flow - The control flow invariant checks that the correct instruction is fetched and performed every cycle. This ensures that the step from $\mathcal{S}_t$ to $\mathcal{S}_{t+1}$ is represented by the correct instruction and that the operation specified by the given instruction is actually the one performed. This corresponds to checking that the function of the functional node is what it is specified to be.

Memory - The memory checker checks that any edges that originate or terminate within the memory have correct shapes and values. The memory checker requires that EDC is on all memory values and that all data passed to or from memory be checked by EDC. This checks that all data flowing along an edge is correct. The memory checker also verifies that the addresses of memory accesses is specified correctly. This then ensures that all edges that contain a register origin (destination) and a memory destination (origin) have the correct source and destination nodes. This then checks the shape of all memory accesses. It should be observed that the shape of identity edges in memory are not explicitly checked as there is no means of directly moving data from one memory location to another.

Computation - This last invariant is simply related to the execution graph by verifying that the result of the operation performed by the function node is indeed the correct result.

The properties of the graph that are verified by the different invariant checkers has now been identified. The last step is to prove that these properties can be used to check all properties of the graph at all times.

Argus Completeness Theorem. *The ideal Argus implementation is capable of detecting all errors at all times in simple cores.*

Proof: This theorem will be proved using induction.

Base Case($t = 0$): Instead of assuming that the initial state is inherently correct, it is instead assumed that the correct initial state of any system can be specified externally. A checksum (i.e. CRC) can then be used to determine whether the current state of the system is equivalent to the previously specified correct initial state.

Induction Case($t \rightarrow t + 1$): This step assumes initially that all the values present in $\mathcal{S}_t$ are correct. It is then necessary to demonstrate that all the values in $\mathcal{S}_{t+1}$ are also correct. It is known that the control flow checker verifies the correctness of the instruction that is obtained to compute the state at $t + 1$. This control flow checker also ensures that the operation that is performed by the functional node is the correct operation to be executed.

With respect to the shape of the graph, the source and destination of each edge is verified by the dataflow checker or the memory checker depending on the source and destination types of the edge as described previously. The value along each edge is also checked by an EDC specified by either the dataflow or the memory checker. This then ensures that the shape and value of each edge in the transition from t to $t+1$ must be correct. Finally, the computation checker verifies that the output of the functional node is the correct result that should be obtained when applying the specified operation to the given inputs.

This then demonstrates that all the properties of the execution graph are specified correctly and that the values that flow from t to $t+1$ are completely checked by the Argus error detection mechanism. Having shown this to be true for any transition from state $\mathcal{S}_t$ to $\mathcal{S}_{t+1}$ it is possible to conclude that an ideal implementation of Argus is capable of detecting all errors at all times in a simple core. $\square$

4.2 Equivalence of Argus-1 to Argus

Having demonstrated the completeness of Argus, it is important to show that a more practical implementation of Argus, in this case Argus-1, is equivalent with some known holes to the implementation of an ideal version of Argus. It has been previously noted that there are several holes such as the finite SHS register size that could result in the Argus-1 implementation permitting silent errors. However, the primary concern here is whether checking dataflow and control flow on a basic block granularity is equivalent to checking one instruction per cycle. Using the framework developed previously it will be shown that Argus-1 and Argus are equivalent with the admission of several known holes in Argus-1.

Argus-1 Equivalence Theorem. *The Argus-1 implementation is equivalent to the ideal Argus implementation with the assumption that all checking mechanisms are error free and there is no aliasing in any checksums or signatures.*

Proof: It has already been proved that the Argus implementation is complete. It is sufficient therefore to show that Argus-1 can be reduced to Argus.

It is assumed that all checking mechanisms are error free and that no aliasing occurs in the signatures or checksums. It is also acknowledged that Argus-1 does not replay stores to ensure that data was actually written correctly into memory. Acknowledging these holes in the system, it remains only to show that the basic block granularity of checking performed by Argus-1 is equivalent to the instruction granularity of checking performed in the ideal version of Argus.

It is necessary for the purposes of this proof to modify the execution graph slightly since checking is now being performed following the execution of several instructions associated with a basic block. The initial state is specified by $\mathcal{S}_t$ and the final state is specified by $\mathcal{S}_{t+n}$ after the n instructions in the basic block are executed. Instead of representing the entire state of the system at intermediate nodes, only nodes that are updated by the output of an instruction are shown in the intermediate states. The algorithm for creating and placing edges is exactly the same as before except the first three steps are repeated for all n instructions before completing the algorithm with the fourth step. This ensures that identity edges map information that goes unchanged during the course of an entire basic block. Having constructed this new framework for basic block granularity execution, it is now possible to consider the properties of this graph that are checked by the Argus-1 implementation.

Dataflow - The dataflow checker implemented in Argus-1 is able to build a signature that directly corresponds to the correct dataflow graph to be executed regardless of the number of instructions that are contained in the dataflow graph. This implies that the signature created by the dataflow checker after the execution of a basic block must be equivalent to the signature of the dataflow graph at compile time. The dataflow checker therefore verifies that dataflow invariant is upheld under basic block granularity checking.

Control Flow - The control flow checker is able to verify that the set of instructions performed during the execution of the basic block is correct by generating a signature corresponding to the instruction stream that was executed. Since this signature must be the same for a basic block regardless of the number of instructions in it, then the control flow checker is able to detect any errors in control flow within a basic block.

Computation - Since the computational checkers are able to verify the result of one function at a time, they are able to verify the computation of all instructions within a basic block since never more than one instruction is executed per cycle.

Memory - The memory checker operates similarly to the control flow checker. A signature is maintained that represents all of the loads and stores performed during the execution of a basic block. The memory checker is therefore able to verify that all memory accesses are performed to the correct locations. The memory checker is unable to verify that stores are actually written, but this is a known hole.

Finally, it should be noted that a watch-dog timer ensures that basic blocks are of finite size and that a check is reached within a finite period of time. Since all the properties of the graph from $\mathcal{S}_t$ to $\mathcal{S}_{t+n}$ have been shown to be verified by Argus-1 or are known to the user as holes, then Argus-1 is equivalent to an ideal implementation of Argus. $\square$

5 Extending Error Detection to Multicore Processors Using Dynamic Verification of Memory Consistency

It has now been shown that errors can be completely detected within simple cores. However, in the presence of chip multiprocessors with many simple cores [6] it is necessary to also verify the correctness of the memory system. Previously Argus was able to verify the correctness of the memory system by assuming that there was only one processor accessing the memory. However, in a chip multiprocessor environment, multiple processors can be trying to access the memory system simultaneously. Ensuring that these accesses occur both coherently and consistently is a very difficult problem.

Dynamic Verification of Memory Consistency (DVMC) is a run time checker developed by Meixner that is able to verify that all memory accesses in a multicore environment are performed correctly with respect to a specified memory consistency model [13]. A proof that this memory checker comprehensively detects all errors with respect to the memory consistency model can be found in [14]. In order to be able to use DVMC in conjunction with Argus to check an entire chip multiprocessor, it is first necessary to understand the invariants checked by DVMC with respect to the memory system. Only the conceptual elements necessary for proving the completeness of Argus with DVMC will be presented here.

Before investigating the different variants upheld by DVMC it is first necessary to make several terms more precise. In a shared memory system a memory access is said to execute when the processor computes the address of the load or store and instruction is issued to the memory system. A memory access is then said to complete when it become visible to the processor that issued the access. Finally, a memory access is said to perform when it becomes visible to all the processors in the system. This terminology will be important in later discussions using DVMC.

5.1 Uniprocessor Ordering

The first invariant that is checked by DVMC is the uniprocessor ordering invariant. This invariant ensures that all reads from a specific address will return the values most recently written to that address by the same processor unless another processor has written a different value in between. This invariant is similar to the memory invariant in Argus with the exception that it does not verify that address being read from or written to is indeed the correct address. Note that the uniprocessor ordering invariant holds with respect to each address separately.

5.2 Cache Coherence

The second invariant checked by DVMC is that of cache coherence. DVMC requires that a specific type of cache coherence known as Single-Writer, Multiple Reader (SWMR) cache coherence be employed. The purpose of this invariant is to ensure that only a single processor is updating the value stored at a specific address at a time. Again note that cache coherence is an invariant that is applied to each address in the memory system separately. In conjunction with the uniprocessor ordering invariant it should be observed that a total ordering of all memory accesses to a single address can be specified.

5.3 Allowable Reordering

The last invariant checked by DVMC is that of allowable reordering. The allowable reordering invariant ensures that all memory accesses obey the memory consistency model. The memory consistency model specifies the possible orderings that can occur between accesses to different memory addresses. Recall that all accesses to a single address can be ordered by cache coherence and uniprocessor ordering. The allowable reordering invariant checks that all accesses to all memory locations obey an ordering that agrees with the memory consistency model specified by the processor. The combination of all three invariants then specify whether a total ordering of all memory accesses performed is correct.

6 Proving the Completeness of Argus with DVMC

The first step in proving the completeness of the Argus with DVMC checking system is to define the system model for the chip multiprocessor. For the purposes of this proof, it is assumed that the system consists of the following:

- N processors each with a Von Neumann architecture.
- A shared memory system composed of one main memory and N caches, with each cache being associated with a single processor. The caches are maintained by a SWMR cache coherence protocol. All addresses specified in the memory system are physical addresses. There are no virtual addresses and therefore no TLB's.
- An interconnect network of some form that connects all N processors.

It is important to realize that this is a very simple model of a chip multiprocessor. No assumptions are made about the nature of the memory consistency model except that it has a SWMR cache coherence protocol. For the purposes of this proof a simple model of hardware is employed to make the proof more intuitive. The simplified model proposed here does not contain the hardware necessary for supporting more relaxed memory consistency models (i.e. write buffers), but this hardware could easily be incorporated into the model.

The idea behind this proof is to recognize that Argus and DVMC have each already been proved to be correct and complete within the context of their own system models. The problem then becomes demonstrating that the boundaries of error detection coverage for both Argus and DVMC meet squarely in the model of the chip multiprocessor defined previously. The key idea is that caches can be used to define the interface between Argus and DVMC.

Before demonstrating that the interface between Argus and DVMC can be defined using caches, it is first necessary to specify some important terminology. To this point in the thesis, the concept of a memory access has not had a precise definition. A memory access is now defined to be a read or write request for an address that is issued explicitly to the processor's cache. The concept of a value being correct in a register or in memory has already been defined. However, it is also necessary to define the correctness of a value a cache. Note that it is not necessary to define that the address of a value in the cache is correct as this will be handled by Argus' memory checker. A value in a cache is said to be correct with respect to a physical address A if the value is equivalent to the latest store performed to the address A. It is important to recognize that this definition of a correct cache value makes no assumption as to the correctness of the value with respect to program execution.

6.1 Three Lemmas

In this section three lemmas are presented that will be used in the proof of completeness of the Argus with DVMC system. These lemmas deal with properties involving the caches and the programs with which they are associated. Lemma 1 shows that correct execution of a program between memory accesses will lead to correct memory accesses being issued to the cache. Lemma 2 shows that SWMR cache coherence ensures that all values in every cache are correct at all times. Finally, Lemma 3 takes the results of Lemmas 1 and 2 and uses them to prove that correct memory accesses will always be performed given that the initial conditions in Lemmas 1 and 2 hold.

Lemma 1. *Correct execution of one program between memory accesses $\rightarrow$ correct memory accesses issued.*

Proof: Assume that the execution up to the n^{th} memory access is correct. Therefore using the dataflow, control flow, and computation invariants of Argus, it is known that the n^{th} memory access will execute correctly (recall the formal definition of executing a memory access). This implies that all of the fields in the memory access instruction are correct and will be sent to the cache correctly. All of the memory accesses issued to the processor's cache will then be correct given that the execution between memory accesses is correct. $\square$

Lemma 2. *SWMR cache coherence $\rightarrow$ all values stored in a cache are correct at all times.*

Proof: The single writer multiple reader cache coherence protocol enforces the policy that only one processor will ever write to an address at a time. Therefore whenever a write occurs it will overwrite all other copies of that value within the system. This then ensures that any new reads occurring after a write will observe the most recently written value into that address. Since all future readers now see the most recently written value to an address then all future readers will see correct values in their caches. It is important to note that once a value is correct within a cache it is protected by EDC and is therefore maintained correctly.

Lemma 3. *Correct memory accesses issued to a cache and correct cache values at all times $\rightarrow$ correct memory accesses performed.*

Proof: It is assumed that all of the memory accesses issued to the cache are correct and that all of the values in the cache are correct at all times. The Argus memory checker will then ensure that any accesses to the cache will access the correct address in the cache and that the value returned will be the correct value stored in the cache and therefore the memory access will have committed correctly. However, it is yet to be seen that the n^{th} memory access will perform correctly with respect to all other memory accesses. It is known that DVMC supports two other invariants: uniprocessor ordering and allowable reordering. The uniprocessor ordering invariant ensures that the n^{th} memory access will perform correctly with respect to all other memory accesses to the same address that were issued from the same processor. It is also known from Lemma 2 that the n^{th} memory access will perform correctly with respect all other memory accesses to the same address that were not issued by the same processor. Finally, the allowable reordering invariant ensures that the n^{th} memory access will perform correctly with respect to all other memory accesses that are not to the same address as the n^{th} memory access. The n^{th} memory access therefore performs correctly with respect to all other memory accesses.

6.2 Argus with DVMC Completeness

Having now developed all of the necessary building blocks, it is now possible to give the proof of completeness.

Argus with DVMC Completeness Theorem. *Argus with DVMC $\rightarrow$ complete error detection coverage in chip multiprocessors with simple cores.*

Proof: This is proved using induction on memory accesses. The first step is to select a program $\mathcal{P}$ at random from the N programs executing on the CMP and use induction on memory accesses issued to the cache.

Base Case - It is assumed that the initial state configuration of the processor and its associated cache is correct. If necessary this can be verified by an external checksum. Using the dataflow, control flow, and computation invariants that are verified by Argus, it is possible to guarantee that execution up to the first memory access is correct.

Induction Case - It is assumed that the execution up to the n^{th} memory access is correct. It is then necessary to demonstrate the n^{th} memory access occurs correctly and that the execution up to the $n+1^{\text{th}}$ memory access occurs correctly. It is already known that the execution up to the n^{th} memory access is correct, therefore it is possible to apply Lemma 1 to see that the n^{th} memory access is issued correctly. DVMC also ensures that SWMR cache coherence is maintained, which by Lemma 2, ensures that the values in the caches are always correct. Combining the results of Lemmas 1 and 2 and applying Lemma 3, it is possible to see that the n^{th} memory access is performed correctly. Using the dataflow, control flow, and computation invariants of Argus it is possible to guarantee that execution from the n^{th} memory access to the $n+1^{\text{th}}$ memory access will also be correct. This then satisfies the inductive hypothesis that the execution up to the $n+1^{\text{th}}$ memory access be correct.

It has been verified that the program $\mathcal{P}$ executes correctly on the CMP. Since the choice of the program $\mathcal{P}$ was arbitrary, then the above logic can be applied to all programs executing on the CMP. It has therefore been shown that Argus with DVMC completely checks all programs executing on a chip multiprocessor with simple cores. $\square$

7 Discussion

The completeness of Argus, the equivalence of Argus-1 to Argus, and the completeness of Argus with DVMC have now all been proved. However, it is still interesting to consider what the actual results of implementing Argus or Argus with DVMC on a simple core or a chip multiprocessor might be. While this is not directly within the scope of this paper, it is work that has been done in conjunction with the above proofs by both Meixner and the author and has practical ramifications for actual implementations of Argus.

7.1 Experimental Verification

In order to determine the actual error coverage of the Argus-1 on an OR 1200 core [10] a gate level representation of the core with Argus was developed by Meixner. This gate level representation was then downloaded onto an FPGA board. Both hard and soft errors were then randomly injected into this core while the core was executing a specialized benchmark designed to stress all components in the core repeatedly over a short period of time. This specialized benchmark was chosen to make sure that the errors could not be masked due to errors occurring in unused or infrequently used components in the processor. 5000 different errors were injected into randomly chosen gates (there were about 40,000) different gates. The error detection percentages can then be seen in Table 1.

Error Type	Unmasked, Undetected	Unmasked, Detected	Masked, Undetected	Masked Detected
Transient	0.76%	37.4%	38.2%	23.7%
Permanent	0.46%	37.6%	38.2%	23.7%

Table 1: Error Detection Coverage of Argus-1

Based on these results it is possible to see that Argus-1 does provide error detection coverage that is extremely close to the ideal error detection coverage of Argus. Indeed, of all of the unmasked errors, Argus-1 was able to detect 98.0% of the transients and 98.8% of the permanent errors. This shows that even having a few holes in an implementation of Argus is not crippling and can still provide extremely high error detection coverage.

7.2 Area Overhead

The area overhead for Argus was computed by the author and Meixner by actually laying out the core and the caches. The core was laid out using CAD tools and the VTVT 250 nm standard cell library [15]. The cache area overhead was computed using Cacti [16]. Initially it was found that OR1200 core unmodified used 6.59mm^2 and the Argus-1 core consumed an additional 16.6% of area. However, when caches are included in this computation then the total area overhead of Argus-1 drops to 10-11% as can be seen in Table 2.

	OR1200	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%

Table 2: Area Overhead of Argus-1 in mm^2

These results show that very little area overhead is incurred in an actual implementation of Argus-1. This is a promising result as it indicates that the potential cost for implementing Argus is small. In addition to this a small area overhead should be indicative of a small power overhead. While this result has not yet been confirmed it is being investigated at this time.

7.3 Future Work

This work has shown that error detection can be completely performed on chip multiprocessors with simple cores. DIVA presents a comprehensive checker of more complex cores that Argus is not mapped onto as easily. Demonstrating that DIVA, Argus, and DVMC could detect all errors in a heterogeneous multicore chip is a subject for future work. In addition to this, checking conversion between virtual and physical address spaces is a subject of future work. This would have to involve having some mechanism for checking the correctness of TLB operation as well. Lastly, checking I/O and interrupts/exceptions is a topic of ongoing work. Being able to prove that errors can be detected in these components of the processor is still an open problem.

8 Conclusion

The proofs of completeness for both Argus and Argus with DVMC have demonstrated the potential of invariant checkers to comprehensively detect all errors in a simple core and within a mulicore respectively with minimal overhead. While these checkers can help to ensure that rising error rates due to shrinking transistors will never result in a correctness problem, the question of the performance impact of rising error rates still has to be addressed. In addition to this, the proofs in this thesis have left several holes open that could prove to be difficult verify. It might ultimately be shown that these holes cannot reliably be checked in any self contained way. Despite these future obstacles, the completeness of Argus and Argus with DVMC are significant results that go a long way to ensuring the correctness of programs in the face of rising error rates.

Acknowledgments

I would like first and foremost to thank my advisor, Daniel Sorin, for all of the guidance that he has given me over the past two years. His mentoring has truly formed me into the researcher and person that I am today. I would also like to thank Albert Meixner and Bogdan Romanescu for their help in all of my endeavours and for allowing me to collaborate with them in their research. A special thank you to all of my office mates for supporting me during the tough times. To Jason Bosko for always sticking with me on my long and winding engineering thought tangents. To Phil Gorman for never failing to be there with a light hearted moment. I would be remiss if I did not take time to thank my family for supporting me in all that I have done. Lastly, thank you to Kamaal Patterson, for showing me that there are greater truths to be found and experiences to enjoy.

References

[1] J.F. Ziegler *et al.*, "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)" *IBM Journal of Research and Development*, vol. 40, Jan. 1996.

[2] J. Srinivasan *et al.*, "The Impact of Technology Scaling on Lifetime Reliability", *International Conference on Dependable Systems and Networsk*, June 2004.

[3] T.M. Austin. "DIVA: A Reliable Submicron Microarchitecture Design", *Proceedings of the 32nd International Symposium on Microarchitecture*, Nov. 1999.

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

[5] E. Rotenberg. "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors" *Proceedings of 29th International Symposium on Fault Tolerant Computing*, June 1999.

[6] P. Kongetira *et al.*, "Niagara: A 32-way Multithreaded SPARC Processor" *IEEE Micro* 25(2), Mar./Apr. 2005.

[7] A. Meixner, M. Bauer, D. Sorin "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores" *Proceedings of the 40th International Symposium on Microarchitecture*, Dec. 2007.

[8] J. Hennessy and D. Patterson *Computer Architecture: A Quantitative Approach* Morgan Kaufmann, 3rd Edition, May 2002.

[9] D. Kuck *et al.*, "Measurements of Parallelism in Ordinary FORTRAN Programs", *Computer* (27), Jan. 1974.

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

[11] A. Meixner and D. Sorin. "Error Detection Using Dynamic Dataflow Verification", *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2007.

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

[13] A. Meixner and D. Sorin. "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures", *International Conference on Dependable Systems and Networks*, June 2006.

[14] A. Meixner and D. Sorin "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures", *Duke University, Dept. of Electrical and Computer Engineering Technical Report #2006-1*, Apr. 2006.

[15] J. 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.

[16] N. Jouppi and S. Wilton. “An EnhancedAcces and Cycle Time Model for On-Chip Caches”, DEC WRL Research Report 93/5, July 1994.