# Verifying the Performance of the PCI Local Bus using Symbolic Techniques

S. Campos E. Clarke W. Marrero M. Minea

June 18, 1996
CMU-CS-96-147

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Revised version of the paper appearing in the proceedings of *IEEE International Conference in Computer Design*, Austin, TX, Oct. 1995

**Abstract**

Symbolic model checking is a successful technique for checking properties of large finite-state systems. This method has been used to verify a number of real-world hardware designs. This methodology, however, is not able to determine timing or performance properties directly. Since these properties are extremely important in the design of high-performance systems and in time-critical applications, we have extended model checking techniques to produce timing information. These results allow a more detailed analysis of a model than is possible with tools that simply determine whether a property is satisfied of not. We present algorithms that determine the exact bounds on the delay between two specified events and the number of occurrences of another event in all such intervals. To demonstrate how our method works, we have modelled the PCI local bus and analyzed its temporal behavior. These results show the usefulness of this technique in analyzing complex modern designs.

This research was sponsored in part by the National Science Foundation under grant no. CCR-9217549, by the Semiconductor Research Corporation under contract 96-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, SRC, or the U.S. government.

**Keywords:** real-time systems, formal verification, symbolic model checking, quantitative timing analysis, PCI Local Bus

# 1 Introduction

Model checking is a technique for verifying finite-state hardware systems [4, 5] that can handle extremely large state spaces efficiently. It determines automatically if a system satisfies its specifications. Models with up to $10^{30}$ states can often be verified in minutes by using symbolic techniques [2, 12]. The method has been used successfully to verify a number of real-world applications. For example, it has been used to find errors in the Futurebus+ cache coherence protocol, adopted as a standard by both IEEE and the U.S.Navy [6].

Although very successful in finding errors, initial model checking algorithms can not be used to verify or compute timing or performance information. Such information is extremely important when designing high-performance hardware systems, or when trying to improve or maximize resource utilization. In addition, guarantees on hardware performance are often necessary when the hardware is to be used in a real-time application. In this case, it is imperative that the performance claims be substantiated with a formal analysis that covers all possible executions.

Several methods have been recently proposed [7, 8, 9] to verify real-time systems. These systems assume that timing constraints are given explicitly in some notation like temporal logic and the verifier determines if the system satisfies the timing constraint. In [3] we have described how to compute timing properties of a real-time software system using symbolic model checking techniques that explicitly compute the timing information as opposed to simply checking a formula. In this paper we show how to apply these techniques to analyze the performance of complex hardware systems.

As in more traditional model checking approaches, a system description given in some hardware description language is compiled into a state-transition graph and represented symbolically using binary decision diagrams [1]. This graph is then traversed using algorithms based on symbolic model checking techniques [2]. As opposed to other approaches, our verification method computes quantitative timing information about a model rather than just determining whether it satisfies a given specification or not. Our algorithms can be used not only to verify correctness but also to evaluate and analyze performance which can lead to a better hardware design.

We present algorithms to determine the minimum and the maximum length of a path between two given sets of states representing specific events in the system. For example, we can use these algorithms to bound the time between asserting a bus request and the corresponding bus grant. In addition, we may need to compute the number of times a third event occurs within such an interval. In the scenario above we may be interested, for example, in the number of times other transactions are issued between the bus request and the corresponding grant. To determine this information, we describe algorithms that calculate the minimum and the maximum number of times a specified condition can hold on a path from a set of starting states to a set of final states. These algorithms compute information that allows for a detailed analysis of hardware performance.

We use these techniques to analyze the performance of the PCI Local Bus. PCI is a high performance bus architecture designed to become an industry standard for current and future high-performance systems. It is used primarily in the Intel Pentium based systems, as well as in the DEC Alpha processor systems. We model the PCI bus, concentrating on its temporal characteristics, and analyze its performance. We compute transaction response time in various configurations of the system. We are able to bound the response time of a PCI transaction as well as to produce detailed information about each phase of the communications protocol. In addition, we compute the overhead imposed by arbitration, by bus acquisition, and by other phases of the protocol. This type of information allows the designers to understand the behavior of the system more accurately than the information generated by traditional verification methods. Our results also uncovered subtleties in the behavior of the system that could have been difficult to find otherwise. We believe that this example demonstrates how our method can be used to assist in the verification and validation of complex hardware designs.

The remainder of the paper is organized as follows. Section 2 presents the underlying results for symbolic model-checking with binary decision diagrams. In Section 3 the symbolic algorithms for computing the minimum and maximum length of the paths between two state sets are presented. Symbolic algorithms for counting the number of states that satisfy a given condition along a path between two sets of states are described in section 4. Section 5 discusses the modeling of the PCI bus and section 6 shows how it can be analyzed using our techniques. Section 7 concludes the paper with directions for future work.

## 2 Symbolic Model Checking

The hardware system being verified is represented as a state-transition graph. A state $\overline{v}$ in this model is represented by a vector assigning values to the state variables $v_1, v_2, \ldots, v_n$. The transition relation $N(\overline{v}, \overline{v}')$ is a formula that evaluates to true when there is a transition in the model from the state $\overline{v}$ to the state $\overline{v}'$, where $\overline{v} = \langle v_1, \ldots, v_n \rangle$ and $\overline{v}' = \langle v_1', \ldots, v_n' \rangle$. A *path* in the transition graph is defined as a sequence of states $\overline{v_0}, \overline{v_1}, \overline{v_2}, \ldots$ such that $N(\overline{v_i}, \overline{v_{i+1}})$ is true for every $i \geq 0$.

Boolean formulas can be constructed from the state variables of the model. A formula is said to be satisfied in a state if and only if the assignment of variable values in the state to the corresponding variables in the formula makes it true. In general, a formula can be satisfied in many states, and we identify a formula with the set of states that satisfy it. Boolean formulas can be represented canonically by binary decision diagrams (BDDs) [1]. Efficient algorithms exist for computing all logical operations on BDDs, as well as for computing existential quantification. Symbolic model checking exploits this efficiency by operating on sets of states represented internally by BDDs [2]. For example, the BDD representing $T(S) = \{s' \mid N(s, s') \text{ holds for some } s \in S\}$, the set of all successors of states in a state set $S$, can be easily constructed from the BDD for $S$ and the BDD for the transition relation in one step, regardless of the number of states in $S$ and $T(S)$.

The properties to be verified by the model checker are expressed in *computation tree logic*, CTL. Computation trees are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state. Paths in this tree represent all possible computations of the program being modelled. Formulas in CTL refer to the computation tree derived from the model. CTL is classified as a *branching time* logic, because it has operators that describe the branching structure of this tree.

Formulas in CTL are built from atomic propositions (in our method, each proposition corresponds to a state variable in the model), boolean connectives $\neg$ and $\wedge$, and *temporal operators*. Each operator consists of two parts: a path quantifier followed by a temporal operator. Path quantifiers indicate that the property should be true of *all* paths from a given state (**A**), or *some* path from a given state (**E**). The temporal operators describe how events are ordered with respect to time for a path specified by the path quantifier. They have the following informal meanings:

- **F** $\varphi$ ($\varphi$ holds sometime in the future) is true of a path if there exists a state in the path that satisfies $\varphi$.
- **G** $\varphi$ ($\varphi$ holds globally) is true for a path if $\varphi$ is satisfied by all states in the path.
- **X** $\varphi$ ($\varphi$ holds in the next state) means that $\varphi$ is true in the next state of the path.
- $\varphi$ **U** $\psi$ ($\varphi$ holds until $\psi$ holds) is satisfied by a path is $\psi$ is true in some state in the path, and in all preceding states, $\varphi$ holds.

Bounded versions of the temporal operators exist [7]. They allow the expression of time-bounded properties, which can be used to verify the real-time behavior of systems.

Some examples of CTL formulas are given below to illustrate the expressiveness of the logic.

- $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$: It is always the case that if the signal $req$ is high, then eventually $ack$ will also be high.
- $\mathbf{AG}(req \rightarrow \mathbf{AF}_{\leq 5}\ ack)$: A request is always followed by an acknowledge within less than 5 steps.
- $\mathbf{EF}(started \wedge \neg ready)$: It is possible to get to a state where $started$ holds but $ready$ does not hold.
- $\mathbf{AG}\ \mathbf{EF}\ restart$: From any state it is possible to get to the $restart$ state.
- $\mathbf{AG}(send \rightarrow \mathbf{A}[send\ \mathbf{U}\ recv])$: It is always the case that if $send$ occurs, then eventually $recv$ is true, and until that time, $send$ must remain true.

## 3 Minimum and Maximum Delay Algorithms

This section presents algorithms for computing minimum and maximum time delays between specified events. All computations are performed on states reachable from a predefined set of initial states. We also assume that the transition relation is total. We consider the minimum delay algorithm first (figure 1). The algorithm takes two sets of states as input, *start* and *final*. It returns the length of (i.e. number of edges in) a shortest path from a state in *start* to a state in *final*. If no such path exists, the algorithm returns infinity. Recall that the function $T(S)$ gives the set of states that are successors of some state in $S$. The function $T$, the state sets $R$ and $R'$, and the operations of intersection and union can all be easily implemented using BDDs.

**proc** *minimum* (*start, final*)
$i = 0$;
$R =$*start*;
$R' = T(R) \cup R$;
**while** $(R' \neq R \wedge R \cap final = \emptyset)$ **do**
    $i = i + 1$;
    $R = R'$;
    $R' = T(R') \cup R'$;
**if** $(R \cap final \neq \emptyset)$
    **then return** $i$;
    **else return** $\infty$;

**proc** *maximum* (*start, final*)
$i = 0$;
$R =$*TRUE*;
$R' =$*not_final*;
**while** $(R' \neq R \wedge R' \cap start \neq \emptyset)$ **do**
    $i = i + 1$;
    $R = R'$;
    $R' = T^{-1}(R') \cap not\_final$;
**if** $(R = R')$
    **then return** $\infty$;
    **else return** $i$;

Figure 1: Minimum and Maximum Delay Algorithms

The first algorithm is relatively straightforward. Intuitively, the loop in the algorithm computes the set of states that are reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach that state.

Next, we consider the maximum delay algorithm. This algorithm also takes *start* and *final* as input. It returns the length of a longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, the algorithm returns infinity. The function $T^{-1}(S')$ gives the set of states that are predecessors of some state in $S'$ (i.e. $T^{-1}(S') = \{s \mid N(s, s')$ holds for some $s' \in S'\}$). We also denote by *not_final* the set of all states that are not in *final*. As before, the algorithm is implemented using BDDs, however, a backward search is required in this case.

## 4 Condition Counting Algorithms

In many situations we are interested not only in the length of a path from a set of starting states to a set of final states, but also in measures that depend on the number of states on the path that satisfy a given condition. For example, we may wish to determine the minimum (maximum) number of times a given condition holds on any path from starting to final states.

Both algorithms in this section take as input three sets of states: *start*, *cond* and *final*. The algorithms compute the minimum and the maximum number of states that belong to *cond*, over all finite paths that begin with a state in *start* and terminate upon reaching *final*.

To guarantee that the minimum (maximum) is well-defined, we assume that any path beginning in *start* must reach a state in *final* in a finite number of steps. This can be checked using the maximum delay algorithm described in the previous section. Finally, we ensure that all computations involve only reachable states, by intersecting *start* with the set of reachable states computed *a priori*.

To keep track at each step of the number of states in *cond* that have been traversed, we define a new state-transition system, in which the states are pairs consisting of a state in the original system and a positive integer. Thus, if the original state-transition graph has state set $S$, then the augmented state set will be $S_a = S \times \mathbb{N}$.

If $N \subseteq S \times S$ is the transition relation for the original state-transition graph, we define the augmented transition relation $N_a \subseteq S_a \times S_a$ as

$$N_a(\langle s, k\rangle, \langle s', k'\rangle) = N(s, s') \wedge (s' \in cond \wedge k' = k + 1 \vee s' \notin cond \wedge k' = k)$$

In other words, there will be a transition from $\langle s, k\rangle$ to $\langle s', k'\rangle$ in the augmented transition relation $N_a$ iff there is a transition from $s$ to $s'$ in the original transition relation $N$ and either $s' \in cond$ and $k' = k + 1$ or $s' \notin cond$ and $k' = k$. We also define $T$ to be the function that for a given set $U \subseteq S_a$ returns the set of successors of all states in $U$. More formally, $T(U) = \{u' \mid N_a(u, u')$ holds for some $u \in U\}$. In the actual BDD-based implementation, an initial bound $k_{max}$ can be selected to achieve a finite representation for $k$, and new BDD variables can be added dynamically if this bound is exceeded. The system is still finite-state because all paths we consider are finite and $k$ is bounded by their maximum length.

**proc** *mincount* (*start, cond, final*)
*current_min* = $\infty$;
$R = \{\langle s, 1\rangle \mid s \in start \cap cond\} \cup \{\langle s, 0\rangle \mid s \in start \cap \overline{cond}\}$;
**loop**
    $Reached\_final = R \cap Final$;
    **if** $Reached\_final \neq \emptyset$ **then**
        $m = \min\{k \mid \langle s, k\rangle \in Reached\_final\}$;
        **if** $m < current\_min$ **then** $current\_min = m$;
    $R' = R \cap Not\_final$;
    **if** $R' = \emptyset$ **then return** *current_min*;
    $R = T(R')$;
**endloop**;

Figure 2: Minimum Condition Count Algorithm

The algorithm for computing the minimum count is given in figure 2. In the algorithm text, *Final* and *Not_final* denote the sets of states in *final* and $S - final$, paired with all possible values of $k$. More formally:

$$Final = \{\langle s, k\rangle \mid s \in final, k \in \mathbb{N}\} \quad \text{and} \quad Not\_final = \{\langle s, k\rangle \mid s \notin final, k \in \mathbb{N}\}$$

The algorithm uses $R$ to represent the state set in $S_a$ reached at the current iteration, while *Reached_final* and $R'$ are its intersections with *Final* and *Not_final* respectively. Variable *current_min* denotes the minimum count for all previous iterations. The computation of the minimum value of $k$ in a set of pairs $\langle s, k \rangle$ can be done by existentially quantifying the state variables (computing $K = \{k \mid \exists \langle s, k \rangle \in S\}$) and following the leftmost nonzero branch in the resulting BDD, provided an appropriate variable ordering is used.

At iteration $i$, the algorithm considers the endpoints of paths with $i$ states. The reached states that belong to *final* are terminal states on paths that we need to consider. The minimum count for these paths is computed, using the counter component of the path endpoints, and the current value of the minimum is updated if necessary. For the reached states that do not belong to *final*, we continue the loop after computing their successors. If all reached states are in *final*, there are no further paths to consider and the algorithm returns the computed minimum.

Finally, we note that the algorithm for the maximum count has the same structure and can be obtained by replacing *min* with *max* and reversing the inequalities. Variants of both algorithms can be used to compute other measures that are a function of the number of states on a path that satisfy a given condition. For example, we can determine the minimum and the maximum number of states belonging to a given set $cond$ over all paths of a certain length $l$ in the state space.

## 5 The PCI Local Bus

The PCI Local Bus [10, 11] is a high performance bus architecture that can have a data width of 32 or 64 bits. It has been designed by Intel to be used in its latest family of processors. Intel's goal is to offer a fast bus design at low cost that will accommodate current as well as future systems. PCI buses can be found in systems based on Alpha or Pentium processors. The majority of Pentium based systems manufactured today employ the PCI bus.

A typical PCI system can be seen in figure 3. The most important subsystems connected to the bus are the processor, a video controller, a SCSI controller, and an ISA bridge controller, which connects the PCI bus to a slower ISA bus. Modems, floppy disk controllers and other low speed components are connected to the ISA bus. Main memory and the secondary cache are connected directly to the processor using a PCI-memory-processor bridge. Other components can be added to the system. Usually expansion slots are provided for this purpose.

Each of the subsystems shown above is allowed to request access to the bus and issue transactions. Slave subsystems are also supported; such subsystems respond to transactions, but do not issue them. A simplified PCI transaction can be seen in figure 4. The request for a transaction starts when a subsystem asserts its request line `REQ`. It then waits until being granted the bus by the arbitration subsystem, which is indicated by the assertion of the `GNT` line. This phase is known as the *arbitration phase*. The next phase is the *bus acquisition phase*. The bus might not be idle when the new master is determined because the previous transaction may still be transferring data. Another transaction cannot be issued before all data has been transferred. The bus is idle whenever both

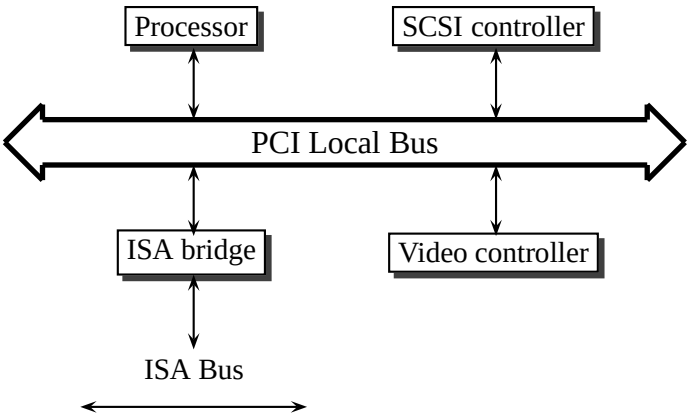


Figure 3: The PCI Local Bus

signals FRAME and IRDY are deasserted in the same cycle, giving access of the bus to the new master. At this point the master asserts the FRAME signal, indicating the end of the bus acquisition phase and the beginning of a transaction. It also has to assert the signal IRDY, meaning that it is ready to send (or receive) data. The bus master has to wait for the target subsystem to respond by asserting its TRDY signal. This indicates that the target is ready to supply (or receive) data. The time interval between the start of a transaction and the assertion of the TRDY signal is called the *target response phase*. Data transfer starts when both IRDY and TRDY are asserted. One clock cycle before the end of the data transfer phase the FRAME signal is deasserted. At the next cycle both IRDY and TRDY are deasserted, and the bus becomes idle. In addition, transactions can be cancelled in various situations. This feature of the protocol is discussed in more detail later.

Arbitration in the PCI bus is implemented by a *two phase arbiter* as seen in figure 5. Each arbiter bank chooses among its incoming requests, and sends its decision to the following bank. The output of Bank2 will be the new bus master. The decision is based on the policy signal, which can be set to *fixed priority* or *round-robin*. If all policies are set to the same value, the global arbitration policy will be either fixed priority or round-robin. However, mixed arbitration policies are possible by combining different policies in the banks.

Our model for the PCI bus follows the description above. Arbitration policies can be set to any possible combination, allowing mixed arbitration policies. However, in our model we must make some restrictions to the protocol described. For example, we must restrict the amount of data being transferred in one transaction. If this restriction is not implemented, no bounds on response time can be determined. In our model a single transaction can transfer between 1 and 16 cache lines of data. Our analysis will show how the information generated by this model can be used to determine the response time for models without this restriction. A similar approach

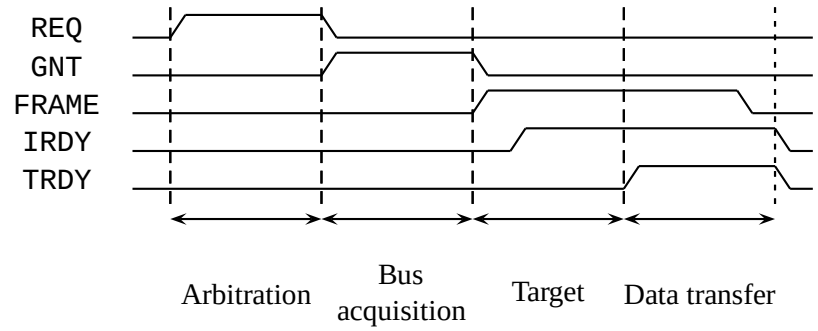


Figure 4: A transaction in the PCI Bus

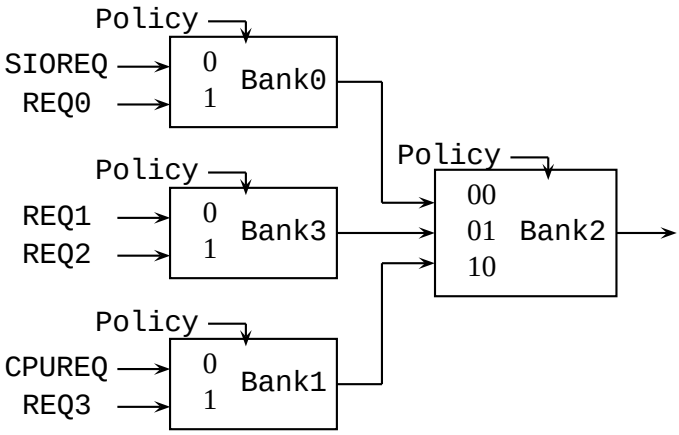


Figure 5: The PCI arbiter

has to be taken with the possibility of cancelling an ongoing transaction. Again, in order to prevent starvation, we must bound the number of times a transaction may be cancelled. Our final model for the PCI bus has $10^7$ reachable states out of a state space of $10^{18}$ states. The transition relation uses less than 10,000 BDD nodes, and the verification was completed in minutes.

## 6 Verification and Performance Analysis of the PCI Bus

Our analysis concentrates on the verification of issues such as transaction termination and arbitration fairness as well as on transaction performance. Being able to estimate the response time of a transaction is extremely important in any bus design, especially in one which has high performance a primary goal. The bus data transfer rate and the overhead imposed by arbitration and communication protocols are examples of parameters involved in such an analysis. If those parameters cannot be determined, it will not be possible to design an optimized system that fully utilizes the available resources.

Moreover, the PCI bus is a good alternative for critical applications in which a bounded response time is vital. However, if the worst case response time of a transaction in the PCI bus hasn't been specified, such applications will most likely be implemented using other bus architectures. By bounding the worst time response of a transaction we hope to help application designers to evaluate the use of the PCI bus more accurately.

The correctness of the PCI bus protocol can be verified using the CTL model checker. For example, absence of starvation for bus access and transaction termination can be verified by the following formulas:

$$\mathbf{AG}(\texttt{REQ} \rightarrow \mathbf{AF}\ \texttt{GNT})$$

$$\mathbf{AG}(\texttt{start\_transaction} \rightarrow \mathbf{AF}\ \texttt{end\_transaction})$$

The properties above show that the response time of PCI transactions is bounded, but they give no indication of their performance. We will use the algorithms described in sections 3 and 4 to determine the response time for transactions. The results of our quantitative analysis also determine the correctness of the algorithm, for example, a transaction always finishes if its maximum response time is less than infinity.

In our performance analysis we will follow the structure of the protocol by computing the response time for each phase of the transaction separately. In this way we can have a better understanding of the behavior of the protocol. By computing the latency of each phase we are able to assert the efficiency of each step in the protocol and obtain the global behavior by adding individual figures. Results will be grouped into two categories, *total bus acquisition latency* and *total transaction latency*. The first category corresponds to the total time between a request being made on the bus and the subsystem actually being able to use the bus. The second category represents the total usage of the bus, that is, the time between asserting the FRAME signal until the end of data transfer. Table 6 shows the response times when the arbitration policy is set to round-robin in all banks and

| Bus Master | Arbitration | | Bus acquisition | | Total bus acquisition | | Target | | Total transaction | |
|---|---|---|---|---|---|---|---|---|---|---|
| | min | max | min | max | min | max | min | max | min | max |
| ISA bridge | 1 | 95 | 1 | 18 | 2 | 113 | 1 | 2 | 2 | 18 |
| SCSI | 1 | 95 | 1 | 18 | 2 | 113 | 1 | 2 | 2 | 18 |
| Video | 1 | 38 | 1 | 18 | 2 | 56 | 1 | 2 | 2 | 18 |
| Processor | 1 | 38 | 1 | 18 | 2 | 56 | 1 | 2 | 2 | 18 |

Figure 6: Response times for global round-robin policy

transaction cancelling is not allowed. Notice that in all cases discussed in this paper the latency for the data transfer phase varies between 1 and 16 clock cycles, there is no overhead associated with it. For that reason, this column will not be shown in the tables.

From the table above we can see two interesting properties of the system. The total transaction latency is at most 18 clock cycles, and in this case 16 clock cycles of data are transmitted. This means that once a master is able to use the bus, it can send data very efficiently. Another characteristic of the protocol is reflected on the bus acquisition times. The maximum of 18 cycles corresponds to one transaction. After being granted the bus the new master may have to wait for at most one more transaction to complete. This shows that once the bus is granted to a master, it will not be granted to another before the first one issues its transaction. Therefore no starvation can occur after a master is granted the bus. This property can be verified by the following CTL formula:

$$\mathbf{AG}(\texttt{GNT} \rightarrow \mathbf{A}[\texttt{GNT}\ \mathbf{U}\ \texttt{FRAME}])$$

A more intriguing result can be seen in the arbitration latency results. The first two subsystems can take almost twice as long to access the bus as the others. In a round-robin environment, all subsystems should be granted equal usage of the resource, but this is not true in our example. By analyzing the execution traces produced by our tools we are able to determine the reason for the unfair access to the bus. The problem arises from the connection of the request lines to the arbiter as seen in figure 7. The ISA bridge and the SCSI controller are connected together to bank 0, while the video and the processor subsystems are alone in their banks. If bus traffic is high, the ISA bridge and the SCSI subsystems may have to wait for the one another before their request reaches bank 2. Subsequently they may have to wait for subsystems connected to the other banks to execute before being granted the bus. In other words, they compete in both levels of arbitration, while the other subsystems only compete in the last level. This causes the worst time latency to be approximately twice as long for these subsystems. We can conclude from these results that two level arbitration *may* have a different behavior than an equivalent one level arbiter. In this case the problem is caused by an asymmetric connection of request lines.

We can also use these results to analyze the overhead imposed by the communication protocol on the transaction

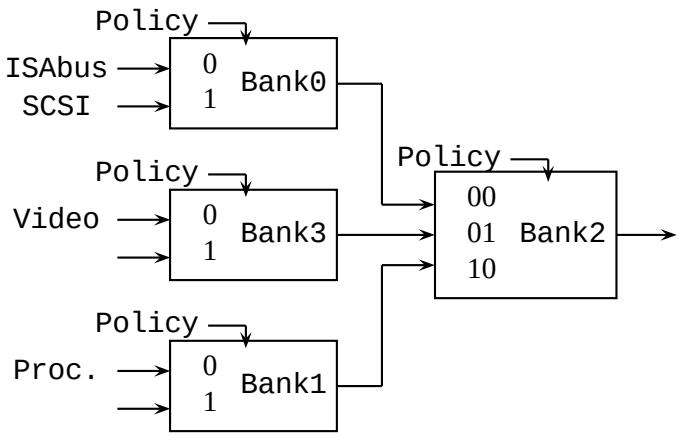


Figure 7: Connections of request lines to the arbiter

time. We have already seen that after asserting the FRAME signal there is an overhead of 2 clock cycles. This overhead is independent of the transfer size. If a transaction is allowed to transfer more than 16 cache lines of data at once, the total utilization of the bus will increase. The designers of the bus can use this information to determine which is the best transfer size for a given system.

The following two formulas have been used to verify the above statements:

$$\mathbf{AG}(\texttt{FRAME} \rightarrow \mathbf{AF}_{\leq 2}(\texttt{state} = \texttt{DATA\_TRANSFER}))$$

$$\mathbf{AG}((\texttt{state} = \texttt{DATA\_TRANSFER}) \rightarrow \mathbf{A}[\texttt{state} = \texttt{DATA\_TRANSFER}\ \mathbf{U}\ \texttt{end\_transaction}])$$

The first formula states that at most two cycles after the transaction starts, it will enter the data transfer phase. The second formula states that once a transaction is in the data transfer phase, it will continue in this phase until its end.

The overhead associated with arbitration can be computed in a similar way. It is more complex, however, because the arbitration latency depends not only on the transaction time, but also on the number of active request lines. We use the condition counting algorithms to uncover more details about this problem. We compute the number of transactions issued on the bus between the time a master requests access and the time it is granted the bus. Up to 5 transactions can be issued during this period for the ISA bridge and the SCSI subsystems, and up to 2 transactions can be issued for the video and processor subsystems. Total transaction time for each of these intermediate transactions is 18 clock cycles. By comparing the total effective data transfer time with the maximum arbitration time, we can see that each intermediate transaction has an arbitration time of one clock cycle. These results are also valid for the video and processor subsystems. We can conclude that the arbitration

| Bus Master | Arbitration | | Bus acquisition | | Total bus acquisition | | Target | | Total transaction | |
|---|---|---|---|---|---|---|---|---|---|---|
| | min | max | min | max | min | max | min | max | min | max |
| ISA bridge | 1 | 19 | 1 | 18 | 2 | 113 | 1 | 2 | 2 | 18 |
| SCSI | 1 | $\infty$ | 1 | 18 | 2 | $\infty$ | 1 | 2 | 2 | 18 |
| Video | 1 | $\infty$ | 1 | 18 | 2 | $\infty$ | 1 | 2 | 2 | 18 |
| Processor | 1 | $\infty$ | 1 | 18 | 2 | $\infty$ | 1 | 2 | 2 | 18 |

Figure 8: Response times for global fixed priority policy

latency can be computed by the formula: Arbitration_Latency = $n$ * (Transaction_Latency + 1), where $n$ is the maximum number of intermediate transactions that can be issued between a request and the corresponding grant (computed with the condition counting algorithms). This formula does not depend on maximum data transfer size.

The above results assume a global round-robin policy. The behavior of the system under a fixed priority arbitration policy has also been studied and the results can be seen in table 8. The ISA bridge is the highest priority subsystem on the bus. Its response time is much lower in the fixed priority configuration than in the round-robin one. However, all other subsystems may starve, since the ISA bridge can continuously issue transactions. Notice that only the arbitration time, but not the transaction time, is affected by the arbitration policy. These response times can be used by the designer to check if the performance of the PCI bus is adequate for a critical application. Other combinations of arbitration policies are possible, but are not presented here for the sake of brevity.

The model described above allows a detailed analysis of the behavior of the PCI bus protocol. Some features of the actual bus, such as parity or data width, have been abstracted from our model, since they do not affect the timing of transactions. However, there are other features that do affect timing such as the possibility of a transaction being cancelled. Errors on the bus may occur, the target may be slow, or unable to produce the data. For example, a transaction requesting data from the ISA bus will most likely experience a long delay, simply because of the relative speeds of the ISA and PCI buses. In the model described above this feature has been abstracted out by the assumption that the target of a transaction responds immediately. A more realistic model that allows transactions to be cancelled has also been implemented.

In order to account for long delay responses and aborted transactions we introduce the concept of transaction cancellation in our model. Transactions may be cancelled any time they are in progress. Transaction cancellations model the fact that in the actual PCI bus whenever a target is unable to answer for a long time, it aborts the transaction, which is reissued later. We model this situation by cancelling the transaction and restarting it immediately by issuing another request. However, reissuing the transaction immediately would not correctly

| Bus Master | Arbitration | | Bus acquisition | | Total bus acquisition | | Target | | Total transaction | |
|---|---|---|---|---|---|---|---|---|---|---|
| | min | max | min | max | min | max | min | max | min | max |
| ISA bridge | 1 | 95 | 1 | 18 | 2 | 113 | 1 | 6 | 2 | 132 |
| SCSI | 1 | 95 | 1 | 18 | 2 | 113 | 1 | 6 | 2 | 132 |
| Video | 1 | 38 | 1 | 18 | 2 | 56 | 1 | 6 | 2 | 75 |
| Processor | 1 | 38 | 1 | 18 | 2 | 56 | 1 | 6 | 2 | 75 |

Figure 9: Response times for global round-robin policy, maximum one cancel

model the response time of a very slow target. To accommodate this situation, in our model a cancelled transaction is restarted as many times as necessary to accommodate the target response time. Using the algorithms described we compute the overhead caused by cancelling and restarting a transaction, and use this result to determine the number of retries for the response delay of a given target.

Moreover, unlimited cancellations may cause starvation. Therefore, in order to compute the worst time response, we must limit the number of cancellations allowed. A cancellation brings the bus to the idle state, as can be verified by the following CTL formula:

$$\mathbf{AG}(\texttt{ABORT} \rightarrow \mathbf{AX}\ \texttt{BUS\_IDLE})$$

As a consequence, consecutive cancellations have the same behavior, because a cancellation brings the system into the same state as before the transaction. Therefore, the total overhead caused by $n$ cancellations is $n$ times the overhead of a single cancellation. Therefore, it suffices to consider the situation in which at most one cancellation occurs. The results for a global round-robin arbitration policy in the presence of at most one transaction cancellation are presented in table 9.

In this table we can see that arbitration latency is not affected by transaction cancellations. The reason is that whenever a transaction is cancelled the current bus master releases the bus and becomes last in the round-robin queue. On the other hand, total transaction latency increases significantly. The execution trace of the transaction with the worst latency shows the following sequence of events (for the ISA bridge subsystem):

1. A transaction starts but is cancelled just before completion, after 17 clock cycles.
2. Another request is made to complete it in the next cycle (one extra clock cycle).
3. An arbitration sequence of 79 cycles follows.
4. A bus acquisition phase starts and takes 17 clock cycles.
5. The transaction starts again, completing after 18 cycles.

The arbitration sequence appearing in item 3 is the same as in the worst case, except that the request is made when *the bus is already idle* because of the cancellation. The difference of 16 clock cycles corresponds to one maximum data transfer phase done by another bus master, as shown by the counterexample for the worst case arbitration latency (not presented for brevity). The total delay caused by the first three items is the equivalent of a worst case arbitration latency plus two clock cycles, caused by the cancellation. A bus acquisition phase and a transaction latency phase, in which no cancellation occurs, account for the last 35 cycles. We can see then that the overhead imposed by a transaction cancellation consists of a worst case arbitration latency, a maximum bus acquisition phase, a maximum transaction latency (without cancellations) and one extra clock cycle. Again, this formula applies for the video and processor subsystems. These results may be used to estimate the performance of an implementation of the PCI in the presence of transaction aborts. The formula derived gives the overhead for one transaction cancellation, and can be extended to many cancellations as well. In this manner, the worst response time in various configurations of the system can be computed.

To summarize the results of our analysis, we have been able to:

- Model the PCI Local bus protocol and verify its correctness. In the round-robin case no starvation of subsystems occur, and transactions always finish, even in the presence of limited cancellations.
- Determine the minimum and maximum latencies for each phase of the protocol, and show which phases are affected by changes in the parameters (such as arbitration policy and presence of cancellations).
- Compute response times independent of specific values for the data transfer phase.
- Determine response time in the presence of limited transaction aborts using the condition counting algorithms described.

These results allow the designers of the protocol to understand its actual behavior and how this behavior changes when parameters of the system are modified. We believe that this is valuable information when verifying and optimizing a new hardware system. This example shows that our method can be used to analyze the performance of modern hardware designs that have very complex behavior. It can help improve the reliability of new products and increase the efficiency of the design process.

# 7 Conclusion

Model checking is a well established technology for hardware verification. However, simply checking that a circuit behaves correctly may not be enough. As time-critical applications become more common, and as the necessity for faster and more efficient circuits increases, some guarantees about system performance may be required. This paper presents algorithms to compute minimum and maximum path lengths as well as the minimum

and maximum number of times an event occurs on all paths from a set of start states to a set of final states. The analysis of the PCI Local bus demonstrates the power of this technique. The PCI is a high-performance bus design used in most Pentium processor based systems. By analyzing its performance we have shown that our techniques can be used in complex industrial designs.

This is the first practical paper, to our knowledge, that combines formal verification and performance analysis. The measurements produced by these algorithms can be used to analyze design decisions before the system is actually implemented. In the PCI bus example, the description of the hardware can easily be modified to model different arbitration policies and different data transfer sizes. This flexibility allows designers to fine-tune system parameters in order to maximize efficiency. We hope that our method can help to increase the reliability of time-critical applications and the efficiency of their design process.

## References


[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Symposium on Logic in Computer Science*, 1990.

[3] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[4] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*. Springer-Verlag, 1981. *Lecture Notes in Computer Science*, volume 131.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[6] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993.

[7] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Lecture Notes in Computer Science*. Springer–Verlag, 1990.

[8] A. N. Fredette and R. Cleaveland. RTSL: a language for real-time schedulability analysis. In *IEEE Real-Time Systems Symposium*, 1993.

[9] R. Gerber and I. Lee. A proof system for communicating shared resources. In *IEEE Real-Time Systems Symposium*, 1990.

[10] Intel Corporation. *82378 System I/O (SIO) - PCI Local Bus*, 1993.

[11] Intel Corporation. *PCI Local Bus Specification*, 1993.

[12] K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.