Performance Analysis of a Reconfigurable Shared Memory Multiprocessor System for Embedded Applications

Darcy Cook[1] & Ken Ferens[2]

[1]JCA Electronics, 118 King Edward St. E., Winnipeg, MB. R3H0N8 Canada
[2]Electrical and Computer Engineering, Room E2-390 Engineering Information and
Technology Complex, University of Manitoba, Winnipeg, MB R3T5V6 Canada
Email: Ken.Ferens@ad.umanitoba.ca

Abstract. This paper presents a method to predict performance of multiple processor cores in a reconfigurable system for embedded applications. A multiprocessor framework is developed with the capability of reconfigurable processors in a shared memory system optimized for stream-oriented data and signal processing applications. The framework features a discrete time Markov based stochastic tool, which is used to analyze memory contention in the shared memory architecture, and to predict the performance increase (speed of execution) when the number of processors is varied. Performance predictions for variations of other system parameters, such as different task allocations and the number of pipeline stages are possible as well. The results of the prediction tool were verified by experimental results of a green screen application developed and run on a Xilinx Virtex-II Pro FPGA with MicroBlaze soft processors.

1   Introduction

A custom multiprocessor development environment would provide added benefit to a developer by providing optional selectable features, frameworks, and tools, such as: architectural frameworks for specific classes of applications; predefined memory configurations; selection of the number of processors; specification of task allocation; and performance prediction tools given a set of selected features. For instance, a framework for a particular class of applications could be invoked from a menu item and automatically built with developer-specified parameters. However, the choice of the parameters may not be a simple matter, because of the design trade-offs to consider. Choosing a set of architectural parameters with consideration of economic constraints may not yield the expected performance increase. Lacking predictive tools, a developer would need to build, run and test the system to determine if it meets perfor-mance requirements. Indeed, this could be a long, iterative, and tedious process. Such a development environment would benefit well by having design time

performance prediction tools, which would complement the selectable features and framework building tools.

An important parameter to consider in designing custom multiprocessor systems is the number of concurrent processors. For industrial applications, one must weigh the cost of adding processors to a design against the expected improvement of execution times. One may conjecture that increasing the number of concurrent processors will result in faster application execution times. However, increasing the number of processors in a multiprocessor system with global memory will result in an increased probability of memory contention. More memory conflicts will result in longer processor waiting times, more processor idle time, and longer task execution time. Indeed, adding additional processors to a system may result in more tasks executed in parallel, but each of the tasks may take longer to execute. Consequently, it is not always clear whether adding more processors will result in enough improved performance to justify the cost of the additional processors. A prediction tool is required. This paper proposes such a performance prediction tool.

The remainder of this paper is organized as follows. Section 2 discusses motivation and previous work that has been done in this area. Section 3 details the proposed solution to the problem. Section 4 compares the predictions of the proposed analysis method with experimental measurements obtained from a multiprocessor system implemented within an FPGA, and with the waiting time prediction of a simple analysis method. Also, simulations of larger numbers of processors were performed and compared. The proposed analysis method agrees with the experimental results. Finally, the conclusions are given in Section 5.

2   Motivation and Related Work

Historically, architectural frameworks for multiprocessor systems have been developed to fit as many applications as possible by accommodating widely differing requirements. This is because the effort of developing application specific architectures was prohibitive and not cost effective. Current developments in technologies, such as soft processors within FPGA, have decreased the effort and cost of developing custom multiprocessor systems [1], [2]-[4]. However, while the cost and effort have decreased, application specific solutions are still not economically viable. A compromise is a multiprocessor system that is tailored to a particular class of problems, which is defined by a common computing model. This can result in a solution that is more computationally efficient than a general solution, and more flexible than an application specific solution. Narrowing the analysis of a multiprocessing system to consider only problems well-suited to a particular computing model can allow for better analysis results, since the computing model provides

additional information compared to general parallel processing problems. This paper develops a multiprocessor system framework for a stream-oriented class of embedded applications, with analytical performance prediction tools.

There have been several studies in the past that have analyzed memory interference in shared memory multiprocessor systems. However, these studies did not use a computing model to constrain the design. Consequently, the analysis methods were applied to a wider range of problems, but at the cost of a less accurate analysis. Since the model of computing was not considered in these studies, assumptions could not be made about the contents of global shared memory, and therefore it had to be assumed that the global shared memory must contain both data and instructions. In the case where global memory contains data and instructions, a single bus global memory multiprocessor system is inefficient because there is a very high probability of memory interference, which counteracts the benefits of executing tasks in parallel. Therefore, in these studies more complex architectures were leveraged to reduce memory interference, such as cache systems and multiprocessor systems with multiple memories, either connected in a crossbar network ([5]-[9]), or with multiple buses ([2],[6],[10]).

Previous studies assumed that all processors requested access to shared memory with the same probability throughout all execution. Therefore, memory requests were often modeled as a Bernoulli process with a fixed probability for discrete time analysis ([5]-[7],[9],[11]), or as a Poisson process with a fixed probability for continuous time analysis [8]. This paper takes a different approach, because, in the stream-oriented data and signal processing computing model, the shared memory contains only data and not instructions. This means that the probability of requesting access to shared memory depends on the specific task being executed, since the function of a task will determine the frequency and size at which data in shared memory needs to be accessed. In this paper each task has its own probability of accessing global memory. As a result, a novel feature of the multiprocessor framework presented in this paper is that the probability of a memory request is allowed to change for a particular processor as the processor switches from executing one task to another.

This paper also differs from previous work in how memory service time is modeled. Most of the previous studies assumed that the memory service time is constant ([5],[7],[9],[11],[12]]). Furthermore, some researchers assumed a Bernoulli process [6] or Poisson process [8] for memory service. In these studies, the type of memory used in the system was abstracted and encapsulated within the hardware architecture. This paper recognizes that the memory service time is dependent on the specific memory type used, and not on the hardware architecture itself. Therefore, the memory service time is represented by a phase

type distribution that can be adjusted to fit the characteristics of the specific memory used in an application.

In summary, the main contributions of this paper are:

1. A statistical measuring tool to measure the expected performance increase with an increase in the number of processors in a multiprocessor system.
2. A method of determining the expected additional execution time for each task due to memory waiting time for analyzing the benefits of adding more processors to a multiprocessor system.
3. By limiting the analysis to consider only a computing model that is specific to stream-oriented data and signal processing, the statistical analysis method produces more accurate results than more general analysis methods.
4. A task dependent, memory-request probability model is developed.
5. A phase type distribution that can be adjusted to specifically fit the characteristics of the memory technology is developed.

3   Solution Strategy

The goal of the analysis is to determine the expected amount of time a processor must wait to access data memory, when another processor has access to the memory. From this information the execution time of the task can be extended to represent the average execution time when considering the time waiting for memory access. This allows for the tools to compare different implementations of the system, such as different numbers of processors.

3.1   Partitioning the Processing Period

One of the difficulties encountered in the analysis was that each processor could execute several different tasks in a processing period, and each task generally could have different and independent memory request probabilities (α_i). The difficulty was in dealing with a variable number of memory request probabilities in a processing period. Further compounding the problem was that the time at which a processor switches from one task to another task is independent from that of other processors in the system. To work around this difficulty, the processing period was partitioned into windows, where each window was chosen so that, within a window, the set of tasks being executed did not change for all processors. The analysis was performed on each window, and after the analysis of a window, the resulting memory access waiting times for the tasks in that window were used to adjust the processing period. This was continued for all windows in the processing period.

For example, a processing period with a full pipeline is shown in Figure 1. The first window analyzed is the largest window where a processor does not execute

more than one task; this is the overlapping section of tasks *A, C,* and *D,* as shown in Figure 1(a). The system is then analyzed to determine how much time those sections of *A, C,* and *D* are lengthened when the time waiting for memory access is considered.
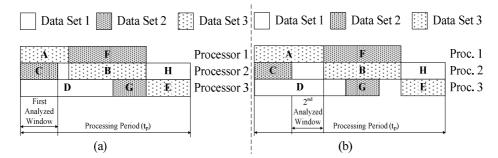

Figure 1 First and second windows analyzed in the processing period.

Suppose that after analyzing the first window, the time for task *A* was extended (due to memory access time). Consequently, the processing period was now lengthened to compensate for Task *A*'s extension, as shown in Figure 1(b).It can be seen that the overall processing period was lengthened, because task A was extended; *B* must come after *A* is finished; and *E* must come after *B* is finished. This change would also create a gap between task *G* and *E* on Processor 3, which is now an additional window that must be analyzed, that was not a window before the first iteration. The second window that is to be analyzed is shown in Figure 1(b). This window consists of task *A*on processor 1 and task *D* on processor 3. Processor 2 does not execute during this window, so the system is analyzed as if there are only two processors for this window. After the second window is processed the processing period may change again. This process is repeated until the entire processing period is analyzed, at which point the entire processing period will be adjusted to take into consideration the memory access waiting times.

3.2   **Analyzing a Partition**

In order to analyze any particular partition of the processing period to determine the amount of time that each processor waits for memory access while another processor is being served, the memory requests by each processor were modeled by a discrete time Markov chain. A state diagram and a transition matrix were used to represent the discrete time Markov chain. From the point of view of each processor in the system, there are four states that the memory controller can be in at any given time. The first state occurs when the memory controller is polling the current processor to see if it has a memory request. The second state occurs when the memory controller is in the midst of servicing a memory

request from the current processor. The third state occurs when the memory controller is servicing other processors and the current processor does not have a pending memory request. The fourth state occurs when the memory controller is servicing other processors and the current processor has a pending memory request. When the memory controller is servicing other processors, it is said to be on vacation with respect to the current processor. The third and fourth states represent the cases where the memory controller is on vacation, with respect to the current processor. The additional task-execution time due to a processor waiting for memory access in a multiprocessor system is determined by the amount of time spent in state four. Accordingly, the goal of the proposed analysis method is to determine the amount of time spent in state four, from which the effect of memory interference on the task execution time can be determined. Figure 2 shows the state transition diagram for processor i, with the probabilities of changing states shown on the transition edges. The transitions from state to state are defined by the probability transition matrix in Eq. (6).
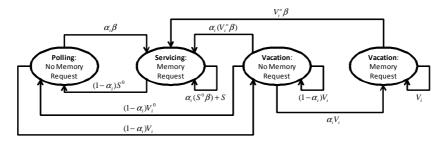

Figure 2 State transition diagram for processor i.

Each memory request queue was individually modeled as a Geo/PH/1 system with PH vacations. This means that the arrival of the memory access requests is a Bernoulli process with arrival probability α_i (for processor i); service is a negative binomial phase type with representation $(\beta,S),k$; and there are vacations with representation $(v_i, V_i),m$. The v_i is a vector describing the probability of starting in each phase of V_i; V_i represents the phases of the vacation; V_i^0 is the vector representing the end of the vacation; and m is the order of the square matrix V_i. The system is considered to be on vacation when the memory controller is serving other processors.

The probability transition matrix for a processor i is represented by Eq. (1). The notation $0_{(ij)}$ represents an i by j matrix full of zeros. Each cell in the matrix ($Cell_{ij}$, where $i,j = 1..4$) represents a probability of transition from one state to another.

$$P_i = \begin{bmatrix} 0 & \alpha_i \beta & (1-\alpha_i)V_i & 0_{(m1)} \\ (1-\alpha_i)S^0 & \alpha_i(S^0\beta)+S & 0_{(mk)} & 0_{(mk)} \\ (1-\alpha_i)V_i^0 & \alpha_i(V_i^0\beta) & (1-\alpha_i)V_i & \alpha_i V_i \\ 0_{(1m)} & V_i^0\beta & 0_{(mm)} & V_i \end{bmatrix} \tag{1}$$

The cells in the first row of the above matrix represent the transition probabilities of the memory controller transitioning from State 1 to States 1, 2, 3, and 4, respectively. The first cell ($Cell_{11}$) represents the transition probability of the memory controller going back into State 1, when it is currently in State 1. In other words, this is the probability the memory controller will check again to see if there is a memory requests from processor i after it has determined there were no memory requests by processor i. The probability of remaining in this state for the next time quanta is 0, because, (i) if the controller determined that the processor i had no memory request, then the controller will transition to the state "Vacation: No Memory Request" with probability $(1- \alpha_i)Vi$; and (ii) if the controller determined that the processor i had a memory request, then the controller would transition to the state "Servicing: Memory Request" with probability $\alpha_i\beta$. Similarly, $Cell_{14} = 0_{(m1)}$ represents the probability that the controller will transition to State 4.

The second row in the above matrix represents the transition probabilities of the controller transitioning from State 2 "currently servicing the memory request of processor i" to States 1, 2, 3, and 4. The system will remain in this state in the next time quanta if memory service does not finish, or if service finishes but another memory request is made immediately. If service finishes and another memory request *is* not made immediately, then the system will return to the first state where the memory controller will wait one time quanta for another memory request. The second state consists of a number of sub-states, where the number of sub-states is k, which is the order of the memory service matrix S.

The third row of the above matrix represents the case where the controller is on vacation (i.e., the memory controller is serving another processor's memory request) and there are currently no memory requests for processor i. The system can go to any other state from this state. If vacation ends (i.e. the memory controller has finished serving other processors) and no memory request arrives, then the system will go to the first state to wait one time quanta for a memory request to arrive. If vacation ends and a memory request does arrive, then the system will go to the second state where the memory request will be served. If vacation does not end and no memory request arrives, then the system will remain in the third state. If vacation does not end and a memory request arrives, then the system will go to the fourth state. The third state consists of a number

of sub-states, where the number of sub-states is m, which is the order of the vacation matrix V_i.

The fourth row of the above matrix represents the case where the system is on vacation and there is currently a pending memory request. In this case no more memory requests can arrive so the system can either remain in the current state if vacation does not end, or it will go to the second state if vacation does end, where the pending memory request will be served. The fourth state consists of a number of sub-states, where the number of sub-states is m, which is the order of the vacation matrix V_i.

3.2.1   Vacation Model

For a given problem the arrival probability, α_i, for each processor in the partition that is being analyzed is known, and the service of the memory, $(\beta, S), k$, is also known; therefore, $(vi, Vi), m$, is the only remaining unknown process.

The vacation probability transition matrix for processor i is given by:

$$V_i = \begin{bmatrix} A_{(i \bmod N)+1} & B_{(i \bmod N)+1} & & & & \\ & A_{((i+1) \bmod N)+1} & B_{((i+1) \bmod N)+1} & & & \\ & & A_{((i+2) \bmod N)+1} & \cdots & & \\ & & & \cdots & \cdots & \\ & & & & A_{((i+N-2) \bmod N)+1} \end{bmatrix} \quad (2)$$

Each submatrix A_j represents transitions between states of processor j while it is being served by the memory controller. Each matrix B_j represents the transitions from the states where the memory controller was serving processor j, to the states where the memory controller is serving processor $j+1$ (or processor 1, if $j = N$, where N is the number of processors in the system). The subscripts of the A_j matrix and the B_j matrix consist of the "mod" operator. This operator is necessary in the notation to account for the wrapping around of the subscript. That is, when processor j is done being served, the memory controller moves to processor $j+1$, but if $j+1$ is greater than the number of processors in the system (N), then it wraps around to processor 1. For example, if there were 4 processors in the system ($N=4$), then the matrix used to represent the vacation of processor 3 ($i=3$) would be of the form:

$$V_3 = \begin{bmatrix} A_4 & B_4 & \\ & A_1 & B_1 \\ & & A_2 \end{bmatrix} \quad (3)$$

The form of the matrix A_j representing transitions between states when the memory controller is serving processor j is shown below.

$$A_j = \begin{bmatrix} 0 & \alpha_j \beta \\ (1-\alpha_j)S^0 & \alpha_j(S^0\beta)+S \end{bmatrix}$$

(4)

The first row in the above matrix represents the state when there are 0 memory requests. In this case the processor will only remain in service if a memory request arrives, in which case the memory request will begin being served. If no memory request arrives then the memory controller will go on to serve the next processor (this transition is represented in matrix B_j).

The second row represents the state when there is 1 memory request that is currently being served. If service finishes and no new memory requests arrive, then the state represented by the first row (0 memory requests) is entered. The state remains the same if either service does not finish, or if it finishes but a new memory request arrives to start a new memory request service.

The form of the matrix B_j representing transitions between states when the memory controller is moving from serving processor j to serving the next processor (which is $(j+1)\ mod\ N$) is shown below.

$$B_j = \begin{bmatrix} (1-\varphi_{(j+1)\bmod N})(1-\alpha_j) & \varphi_{(j+1)\bmod N}(1-\alpha_j)\beta \\ 0_{(1k)} & 0_{(kk)} \end{bmatrix}$$

(5)

The 1st row of this matrix represents the state when there are 0 memory requests for processor j. The 2nd row represents the state when there is 1 memory request for processor j. Since the processor will never start a vacation when there is a pending memory request that can be serviced, the probability of starting to serve the next processor when there is one memory request is 0, which is why the second row consists of zeros. When there are 0 memory requests the memory controller will start to serve the next processor, but it could transition to the state where there is no pending memory request for the next processor, or it could transition to the state where there is a pending memory request for the next processor, depending on whether a memory access request has arrived for the next processor since it was last served. The parameter φ_j represents the probability that a memory request is made by processor j from the time its vacation starts to the time that its vacation ends. This means that $1-\varphi_j$ represents the probability that there are no memory requests made by processor j in the time that its vacation starts to the time its vacation ends. The first entry in the first row of matrix B_j represents the transition from serving processor j to serving the next processor (processor $(j+1)\ mod\ N$) when there are no memory requests pending for processor $(j+1)\ mod\ N$. The second entry in the first row of

matrix B_j represents the transition from serving processor j to serving the next processor (processor $(j+1)$ mod N) when there is one memory request pending for processor $(j+1)$ mod N. The vector v_i that represents the start of vacation for processor i can be represented as follows:

$$v_i = \left[\left(1 - \varphi_{(i+1)\bmod N}\right) \quad \varphi_{(i+1)\bmod N}\beta \quad 0 \quad ... \quad 0\right]$$

(6)

The first entry in the vector represents the transition to serving the next processor (processor $(i+1)$ mod N) when there is no pending memory request for the next processor. The second entry in the vector represents the transition to serving the next processor when there is one pending memory request. The rest of the vector is filled with zeros. The vector that represents the transitions when the vacation of processor i ends is given by V_i^0, and can be represented as follows:

$$V_i^0 = \begin{bmatrix} 0 \\ ... \\ 0 \\ 1 - \alpha_{((i+N-2)\bmod N)+1} \\ 0_{(1k)} \end{bmatrix}$$

(7)

This shows that the vacation for processor i finishes after processor $((i+N-2)$ mod N)+1 (which is the previous processor to i in the cycle) was being serviced, but now has 0 memory requests, and no new memory request arrived.

The parameter φ_i is defined as the probability that a memory request occurs for processor i while processor i is on vacation. To determine this value, first the amount of time spent in the vacation process needs to be known. The probability of the vacation process ending in a particular number of time quanta needs to be determined for all time quanta amounts where the probability is significant. The first step in calculating these probabilities is to create a new Markov chain with a probability transition matrix V_i' by combining v_i, V_i, and V_i^0, as shown below:

$$V_i' = \begin{bmatrix} 0 & v_i \\ V_i^0 & V_i \end{bmatrix}$$

(8)

Starting in state 1 of V_i', the system will transition to the sub-matrix V_i by the probabilities in the starting vector v_i, it will then sojourn within V_i until it returns to state 1 by the probabilities defined in V_i^0. Since the vacation process $(v_i, V_i), m$ starts through v_i and ends through V_i^0 and sojourns within V_i during vacation, the vacation time is the same as the time that it takes to return to state

1 of V_i' for the first time when starting from state 1. The parameter $f_{x,y}^{(n)}$ is defined as the probability of first visiting state y from state x in a Markov chain at the nth time quanta. The following result has been shown to be valid by [13].

$$f_{x,y}^{(n+1)} = \sum_{z=1}^{m} v_{x,z} f_{z,y}^{(n)} - f_{y,y}^{(n)} v_{x,y} \qquad for\ n \geq 1 \tag{9}$$

The m is the number of states in the Markov chain, and $v_{x,y}$ is the probability of transitioning to state y from state x. This also means that $f_{x,y}^{(1)} = v_{x,y}$ for any x and y. The probability of finishing the vacation process in n time quantum can then be calculated by using (14) to calculate $f_{1,1}^{(n)}$ in the Markov chain represented by V_i', which is the probability of first returning to state 1 starting from state 1 in n time quanta. The probability of processor i requesting access to memory in one time quanta was previously given as α_i. This can be used to calculate the probability of processor i requesting access to memory within n time quantum, defined as $\sigma_i^{(n)}$, with the following equation.

$$\sigma_i^{(n)} = \alpha_i \sum_{h=1}^{n} (1 - \alpha_i)^{h-1} \ for\ n \geq 1 \tag{10}$$

Eqs. (9) and (10) can now be used to calculate the probability that the vacation for processor i will end in n time quantum and that there will be a memory request made by processor i during that vacation. This probability is defined as $\varphi^{(n)}$ and is calculated by:

$$\varphi_i^{(n)} = \sigma_i^{(n)} f_{1,1}^{(n)} for\ n \geq 1 \tag{11}$$

Therefore the probability that a memory request occurs for processor i while processor i is on vacation is given by:

$$\varphi_i = \sum_{n=1}^{\infty} \varphi_i^{(n)} \tag{12}$$

It is not practical to use (12) to determine φ_i since this equation involves an infinite sum. It can be shown that if the probability of requesting access to memory for each of the processors is less than 1 (i.e., $\alpha_i < 1$ for all i), then the probability of eventually finishing a vacation is 1. This means that state 1 of V_i' is a recurrent state for which the following equation holds true [11]:

$$\sum_{n=1}^{\infty} f_{1,1}^{(n)} = 1 \tag{13}$$

This fact can be used to determine a practical limit to the sum in (12) by choosing some acceptable error limit ε_r, where:

$$\sum_{n=1}^{r} f_{1,1}^{(n)} = 1 - \varepsilon_r \qquad (14)$$

Eq. (14) gives an upper limit, r, to the sum in (13), where the probability of the vacation ending in more than r time quantum is considered insignificant. The smaller the error, ε_r, the larger the value of r, which means that more accuracy in the calculation of φ_i will come at the cost of increased computational overhead. Once the value of r is calculated, φ_i can be approximated by:

$$\varphi_i = \sum_{n=1}^{r} \varphi_i^{(n)} \qquad (15)$$

The calculation of φ_i is shown in the form of a pseudo code function below.

```
Function φ_i = calc_phi (V_i', α_i)
   -- order_V_i' is the order of the square matrix V_i'
   order_V_i' ← get_order(V_i')
   n ← 1
   F ← V_i'  -- F is the matrix holding the probability of first
        --moving from each state to each other state of V_i'for
        -- the current value of n, where f(1,1) is the
        -- probability of finishing the vacation in n time
        -- quantum
   sum ← F(1,1)
   -- σ_i is the probability of a memory request occurring in n
   -- time quantum for the current value of n (Equation15)
   σ_i ← α_i
   φ_i ← σ_i * F(1,1) -- Equation 16

   -- check to see if the error limit has been reached
   while (sum < 1- ε_r)
      --this next for loop calculates the next value of F as
      -- shown by Equation 14
      for x=1 to order_V_i'
         for y=1 to order_V_i'
            total ← 0
            for z=1 to order_V_i'
               total ← total + V_i'(x,z)*F(z,y)
            end (for)
            F_new(x,y) ← total - F(y,y)* V_i'(x,y)
         end (for)
      end (for)
      n ← 2
      F ← F_new
      sum ← sum + F(1,1)
      σ_i ← σ_i+ (1- α_i)^{n-1}-- Equation 15
      φ_i ← φ_i + σ_i * F(1,1) -- Equation 16 and 17
   end (while)
end (function)
```

An iterative algorithm to calculate φi that depends on the probabilities in the probability transition matrix Vi' is outlined below.

```
Fori=1 to N
   φᵢ ← αᵢ-- initialize all φᵢ parameters
   εᵢ ← 1          -- initialize all εᵢ
end (for)
while (|εᵢ| > 10-12 for any i)  -- check for convergence
   fori=1 to N
       φᵢ_old ← φᵢ  -- save the last φᵢ parameter, because a
                        -- new one will be calculated
       -- create the Vᵢ' matrix the based on the latest φᵢ
        Vᵢ' ← build_vacation_process(φᵢ,i)
       -- calculate the new value of φᵢ
        φᵢ = calc_phi (Vᵢ', αᵢ)
    εᵢ ←φᵢ_old - φᵢ   -- calculate the error between the
                        -- current φ and the last one
   end (for)
 end (while)
```

The above algorithm first assigns an arbitrary value to φ_i for all i, $1 \leq i \leq N$. In this case the value for φ_i is assigned α_i. While any value between 0 and 1 can be assigned and the algorithm will still work, using a value that is closer to the actual final value will result in faster convergence. Since φ_i is the probability that there will be a memory request while processor i is on vacation and α_i is the arrival probability, in general, the larger α_i is, the larger φ_i will be, which is why α_i is used as a starting guess.

An error value ε_i is kept for each processor. This is the difference between the latest value of φ_i and the value of φ_i that was calculated previously. When all of the error values are less than 10^{-12}, then this means that the algorithm has converged to a final value of φ_i for all i. The error values are initialized to 1 at the beginning to ensure that the while loop is entered the first time. Then the vacation matrices for each processor i (V_i, V_i^0, v_i) are built using the current value of φ_i for all i. Then a new value of φ_i for all i is calculated using the function *calc_phi*. The difference between the new values of φ_i and the previous values of φ_i are calculated. This difference is checked to see if it is less than 10^{-12} for all i. If the error is smaller than the limit, then the values of φ_i have converged to the final values, otherwise the process needs to be repeated. Once the final values of φ_i are determined, there are no longer any unknowns, so the vacation process is fully defined. This algorithm depends on convergence of the φ_i values. If the values of φ_i do not converge then the algorithm would continue indefinitely and, therefore, it would not be stable. Explicit proof of convergence for this algorithm is not offered in this paper; however, an argument for proof of convergence could be made that is similar to the proof of the stability of token passing rings made by Georgiadis and Szpankowski in [14].

3.2.2   Determining the Memory Access Waiting Probability

Now that the vacation process for each processor is defined, the probability distribution of the discrete time Markov chain that models the memory accesses of each processor can be determined. From the probability distribution the amount of time that is spent waiting for access to the memory when the memory controller is currently serving another processor can be determined.

First the vacation process, $(v_i, V_i), m$, for each processor i is built using the determined values of φ_j for all $1 \leq j \leq N$, as shown with (3), (4), (5), (6), and (7). Then the vacation process is used to build the probability transition matrix for the Markov chain representing the memory accesses of processor i, as shown with (1). Then the steady state probability vector π_i can be calculated for the Markov chain. The steady state probability vector for P_i can be calculated by solving for $\pi_i = \pi_i P_i$. The steady state probability vector represents the probability of being in any given state of P_i in a steady state condition.

The fourth block row of P_i shown in (1) represents all of the states where processor i has a pending memory request, but the memory controller is currently serving other processors. Therefore, the sum of the probabilities in the steady state vector that represent the states in the fourth block row of the P_i shown in (1) is the probability that the processor has a pending memory request, but the memory controller is serving another processor. Each of the rows in the matrix shown in (1) is made up of several sub-rows of which the number depends on the number of states in the memory service process, which is k. The first row shown in (1) is actually only one row. The second row represents k actual rows. The third and fourth rows are each made up of m sub-rows, where m is the order of the vacation process. The order of the vacation process is also dependent on the order of the memory service, and can be determined by (16):

$$m = (k+1)(N-1) \tag{16}$$

There is one block row in the vacation for each processor, except the processor who the vacation process is defined for, so there is *N-1* block rows in the vacation. Each block row in the vacation is made up of *k+1* rows, one row for the case where the currently serviced processor has 0 memory requests, and k rows for the case where the currently serviced processor has 1 memory request that is being serviced. This means that the sum of the last m items in the steady state vector π_i will be the probability that processor i will have to wait for memory access when it has a pending request because the memory controller is currently serving another processor. This is represented mathematically as:

$$\zeta_i = \sum_{h=2+k+m}^{1+k+2m} \pi_i[h] \tag{17}$$

Where ζ_i is defined as the probability that processor i will have to wait for memory access when it has a pending request because the memory controller is serving other processors, and the notation $\pi_i[h]$ means the item h in vector π_i.

3.2.3 Adjusting the Partition to Account for Waiting Time

Now that ζ_i for each processor can be calculated, these values can be used to adjust the length of the partition of the processing period that is being analyzed so it can be adjusted to account for the time that each processor spends waiting for memory access. The partition ends when the first task that is executing in the partition ends. In order to determine which task ends first, the end time of each of the tasks is calculated taking into consideration the memory access waiting time. The end time of each task can be calculated with (18):

$$t_{remaining\_new\_i} = \frac{t_{remaining\_old\_i}}{1 - \zeta_i} \qquad (18)$$

The $t_{remaining\_old\_i}$ is the time remaining in the task execution without considering the memory access wait times, from the beginning of the current partition being analyzed. The task that has the smallest $t_{remaining\_new\_i}$ is the task that will end first, and therefore this is the new partition time, defined as $t_{partition\_new}$.

The other calculation that needs to be done in order to adjust the processing period to be able to analyze the next partition is to determine how much of the task processed on each of the processors is done within the analyzed partition. The ratio of the task executed in the partition time to the total execution time of the task that is executed on processor i is defined as θ_i. The value of θ_i can be calculated by (19):

$$\theta_i = \frac{t_{partition\_new}}{t_{total\_task\_i}}\left(1 - \zeta_i\right) \qquad (19)$$

The $t_{total\_task\_i}$ is the total execution time of the task executed by processor when the memory access waiting time is not considered. The remaining time that needs to be analyzed for each task can then be calculated by (20):

$$t_{task\_remaining\_i} = t_{total\_task\_i} - t_{total\_task\_i}\left(\theta_i + \theta_{prev\_i}\right) \qquad (20)$$

The θ_{prev_i} is the ratio of the amount of the task that was analyzed in previously analyzed partitions. After the amount of each task that is remaining is calculated the entire processing period can be updated, and the next partition can be analyzed.

4   Experimental Results

Validation of the statistical analysis method was accomplished by comparing it with the measurements taken from a real multiprocessing system. We used a Xilinx Virtex-II Pro FPGA [15] on the Xilinx XUPV2P development platform [16]. MicroBlaze [17] soft processors were used, where each processor was clocked at 100 MHz. This system was limited to four MicroBlaze processors for a given application. The minimum time quantum was 10 ns. A polling based memory controller was implemented within the FPGA. The global memory was implemented with RAM with access time less than processor clock.

A green screen video system was used to test the analysis method. This is a common technique used for television weather forecasts to show the weather map behind the meteorologist. The example application takes two images that are in YUV colour format, converts them both to RGB colour format, then replaces all of the green pixels in the primary image with the corresponding pixel from the secondary image. Then the combined image is converted back to the YUV colour format. The example application was divided up into 16 tasks.

The first step in implementing the system was to run the example application on a system with a single processor, to determine the serial execution time of each of task (Table 1). The tasks were then allocated to each of the N processors based on the single processor task parameters using a greedy scheduling algorithm from [18]. In order to analyze this system to predict the effect of the memory access waiting times, a memory access model is needed. In this case the global memory service was modeled by a negative binomial process, with $k=18$ and $p_e=0.95$. This resulted in a probability distribution that closely matched the experimental data.

Table 1   Single processor task parameters.

Task	Initial Task Time (ns x 10)	Access Request Probability (α_i)	Task	Initial Task Time (ns x 10)	Access Request Probability (α_i)
0	1484	0.054	8	968	0.525
1	1484	0.054	9	968	0.525
2	1484	0.054	10	1268	0.549
3	1484	0.054	11	968	0.525
4	1484	0.054	12	2630	0.681
5	1484	0.054	13	1268	0.549
6	968	0.525	14	1368	0.459
7	1268	0.549	15	1368	0.459

An example application where the global memory bandwidth is saturated after only adding a few processors to the system was chosen specifically to show the benefit of this analysis method over a more simplistic approach. The analysis of

the processing period was performed for a system with 2, 3, and 4 processors and compared to the experimentally measured processing period times. Also, a simple analysis method was applied that calculates the processing period by assuming that the maximum bandwidth of the global memory had been reached. Therefore, the processing period time was calculated by subtracting the portion of each task that was due to memory accesses, then calculating the processing period of the parallel tasks, then adding the sum of the time for each memory access (which was previously subtracted). This method essentially assumes that every time a memory access was requested, the global memory was already being serviced by another processor, so that the processing time can be parallelized but all of the memory accesses were executed serially. In addition, a comparison was made with an *oblivious* processing period, which ignores memory interference.
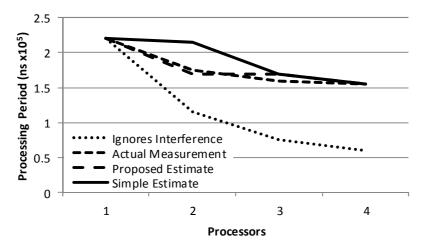

Figure 3 Processing period results (saturated bandwidth).

Figure 3 shows that the proposed analysis predictions are quite close to the actual measurement of the task periods. The simple analysis method is not accurate in the case of 2 processors. The simple analysis method gives an upper limit to the expected processing period time, which is close to the actual measurement only when the global memory bandwidth is saturated (3 and 4 processors), but it is not a good estimate when the bandwidth is not saturated (2 processors). The proposed analysis method in this paper is superior to the simple analysis method because it does not depend on memory bandwidth saturation, and it gives a good estimate for all numbers of processors tested. Furthermore, the results show the significance of including the effect of memory interference; the predictions made by the oblivious method, which ignores memory interference, are significantly overvalued, while the proposed estimate agrees with the actual hardware.

Additional simulations were performed to predict and compare the performance of larger numbers of processors. An application was chosen such that it could be divided into different numbers of tasks and allocated to different numbers of processors, as follows: {2,8}, {3,12}, {4,16}, {5,20}, {6,24}, {7,28}, {8,32}, {9,26}, {10,40}, {11,44}, {12,48}, {13,52}, {14,56}, {15,60}, and {16,64}; where in {p,t}, *p* is the number of processors and *t* is the number of tasks. The application required 10000 ns of single CPU time. Each task was assigned a random task time, with the constraint that the sum of task times of all tasks was kept constant at 10000 ns. In addition, each task was assigned a random probability of memory access, with the constraint that the sum of probabilities of all tasks remained constant at 1.0. This ensured the application had a 100% chance of accessing global memory. Finally, each task was randomly allocated to one of four slots of a processor.

Figure 4 shows that the processing period of the proposed method decreases with increasing number of processors, as expected. The rate of decrease is increasingly dampened with increasing number of processors because of the increased probability of memory contention. Adding additional processors to the system results in more tasks executed in parallel, but each of the tasks take longer to execute because of memory contention. The proposed prediction tool clearly shows the expected improvement of performance, and it provides a means to judge and justify the cost of adding processors to the system. In particular, the results show that the system reaches global-memory interference-saturation at about 6 processors, and, accordingly, there is decreasing benefit in adding more processors than 6 to this system.
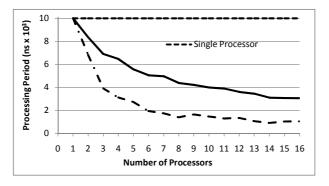

Figure 4 Prediction and comparison of multiple processor performance.

5   Discussion

Figure 3 and Figure 4 demonstrate the importance of considering the effect of memory interference in a shared memory multiprocessor system. On the one hand, the oblivious prediction, which ignores the effect of memory interference, overshoots the processing period by an average of 45%, and in some cases as much as 80%. On the other hand, the proposed prediction method agrees with the experimental results, and provides a very good estimate of the actual processing period. Furthermore, the results demonstrate that the proposed prediction method leads to a faster and more economical system. For instance, Figure 3 shows that, on the one hand, the simple analysis method predicted a very minimal improvement for increasing the number of processors from one to two, and this may have incorrectly lead a developer to choose a one-processor system. On the other hand, the proposed prediction method would have suggested a three-processor system, which runs 30% faster than a single-processor system, at the cost of an additional two processors. Finally, the simulations suggest the proposed prediction method remains superior to the oblivious method. The simulations show that while the oblivious method predicts the same global memory saturation point, the proposed prediction method provides a better economic cost and benefit analysis.

6   Conclusion

This paper has provided a method for determining the expected amount of time that each processor in a stream-oriented shared memory multiprocessor system will wait for memory access because another processor is being served. Since the memory controller operated by polling each processor for memory requests, each processor's memory requests was individually modeled as a discrete time Markov chain Geo/PH/1 system with PH vacations. Then by using an iterative algorithm to determine the vacation process for each processor, the system was analyzed to determine the amount of time that each processor spent waiting for memory access when another processor was accessing the memory. The proposed analysis allows the processing period of a pipelined execution of a data flow graph to be adjusted to account for the memory access wait time. The proposed analysis tool is useful for evaluating different task allocations, number of pipeline stages, and number of processors in a system to see how changes in these parameters could change the performance of a system.

Acknowledgement

References

[1] Cong, J., Han, G. & Jiang, W., *Synthesis of an Application-Specific Soft Multiprocessor System*, Proc. 2007 ACM/SIGDA 15th International Symposium of FPGA, pp.99-107, 2007.

[2] John, L.K. & Liu, Y., *Performance Model for a Prioritized Multiple-Bus Multiprocessor System*, IEEE Transactions on Computers, **45**(5), pp.580-588, May 1996.

[3] Othman, S.B., Salem, A.K.B., Saoud, S.B., *MPSoC Design of RT Control Applications based on FPGA SoftCore Processors*, 15th IEEE International Conference on Electronics, Circuits, and Systems, pp.404-409, Sept. 2008.

[4] Ravindran, K., Satish, N., Jin, Y. & Keutzer, K., *An FPGA-Based Soft Multiprocessor System for IPV4 Packet Forwarding*, International Conference on Field Programmable Logic and Applications, pp.487-492, Aug. 2005.

[5] Bhandarkar, D.P., *Analysis of Memory Interference in Multiprocessors,* IEEE Trans. Comp, **C-24**(9), Sept 1975.

[6] Das, S.K. & Sen, S.K., *Analysis of Memory Interference in Buffered Multiprocessor Systems in Presence of Hot Spots and Favorite Memories*, Proceedings of the 10th International Parallel Processing Symposium, pp.281-285, Apr 1996.

[7] Mudge, T.N., Al-Sadoun, H.B. & Makrucki, B.A., *Memory-Interference Model for Multiprocessors based on Semi-Markov Processes,* IEEE Proceedings on Computers and Digital Techniques, **134**(4), pp. 203-214, July 1987.

[8] Naderi, M., *Modelling and Performance Evaluation of Multiprocessors, Organizations with Multi-Memory Units*, SIGARCH Comput. Archit. News, **16**(5), pp. 35-51, Dec 1988.

[9] Sethi, A.S. & Deo, N., *Interference in Multiprocessor Systems with Localized Memory Access Probabilities*, IEEE Transactions on Computers, **C-28**(2), pp. 157-163, Feb 1979.

[10] Paul, J.M. & Mickle, M.H., *Multiprocessor Shared Memory Access and Rewards,* Journal of the Franklin Institute, **335**(4), pp. 629-641, May 1998.

[11] Isaacson, D.L. & Madsen, R.W., *Markov Chains Theory and Applications*, New York, United States, Wiley, 1976.

[12] Reijns, G.L. & van Gemund, J.C., *Analysis of a Shared-Memory Multiprocessor via a Novel Queuing Model*, J. Syst. Archit.(Netherlands), **45**(14), pp.1189-1193, July 1999.

[13] Rosenblatt, M., *Markov Processes: Structure and Asymptotic Behavior*, New York, United States: Springer-Verlag, 1971.

[14] Georgiadis, L. & Szpankowski, W., *Stability of Token Passing Rings*, Queueing Syst. Theory Appl., **11**(1-2), pp. 7-33, Jul 1992.

[15] Xilinx Inc., *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, DS083 V4.7, Nov. 2007.

[16] Xilinx Inc., *Xilinx University Program Virtex-II Pro Development System-Hardware Reference Manual*, UG069 V1.1, Apr 2008.

[17] Xilinx Inc., *MicroBlaze Processor Reference Guide*, UG081 V9.0, 2008.

[18] Hu, T.C., *Parallel Sequencing and Assembly Line Problems*, Operations Research, **9**(6), pp.841-848, 1961.