Performance Analysis of Superscalar Processors Using A Queueing Model

Y. Zhu and W.F. Wong
Department of Information Systems and Computer Science
National University of Singapore
Singapore 119260
E-mail: {zhuyongx, wongwf}@comp.nus.edu.sg

Abstract

Superscalar processors have become the *de facto* standard of RISC processors today. Due to its complexity, estimating the performance of any superscalar processor design is a difficult task. To this end, several analytical models of such processors have been proposed. In this paper, we present a novel *Multiple Class and Multiple Resource Queueing Model* (MCMR) of superscalar processors. In this model, instruction classes and functional units can be modelled and studied. With analysis, numerical calculation and discrete event simulations, we were able to identify a bottleneck in superscalar processor design.

1 Introduction

Current techniques for determining the performance of a new superscalar processor design are usually based on *trace-driven* simulation. A benchmark is run on existing hardware to generate a trace. A simulator configured for the new implementation then "runs" through the trace and generates performance information. In order to obtain accurate performance data, this process must be repeated for many large benchmarks with extremely long traces. This approach has some shortcomings. The traces are long and expensive to store. Simulation of these long traces is also time-consuming.

Another approach is modelling. Previous works on this approach includes those of [Austin & Sohi 1992], [Lam & Wilson, 1992] and [Dubey et al., 1994] which are based on probability theory. These studies are generally experimental studies which attempt to obtain performance by simulation. In [Noonberg & Shen, 1994] the use of probability vectors were proposed in obtaining a more theoretical model of a superscalar processor. However, none of the above use queueing theory in their modelling. Our paper presents a novel queueing model of superscalar processors. In this theoretical model, a benchmark and a trace are analysed only once to compute the instruction distributions. These distributions are essentially an abstraction of the characteristics of the trace. Each processor implementation's specification is analysed to determine the functional unit resource set. Once the instruction distributions of a program are determined, its performance on an arbitrary machine, characterised by the functional unit resource set, can be computed without re-examing the trace.

Section 2 describes our model in detail. Section 3 gives analytical performance results of our model. A discrete simulation setup to verify our model is described in Section 4. The verification of our analytical results by discrete simulation in Section 5. This is followed by a conclusion.

2 MCMR Model Description

A *multiple-class multiple-resource* (MCMR) system is a queueing system where there are several classes of customers, each requiring a particular set of resources to service. There have been a number of studies on the MCMR system. Assuming a simple distribution for the arrival process, [Lazowska et al., 1984] described a simple method to estimate a MCMR system's performance. Some steady state solutions were also reported in works in [Matta & Shankar, 1995] and [Jain, 1991]. For modeling a superscalar processor, we used a MCMR system shown in Fig. 1.

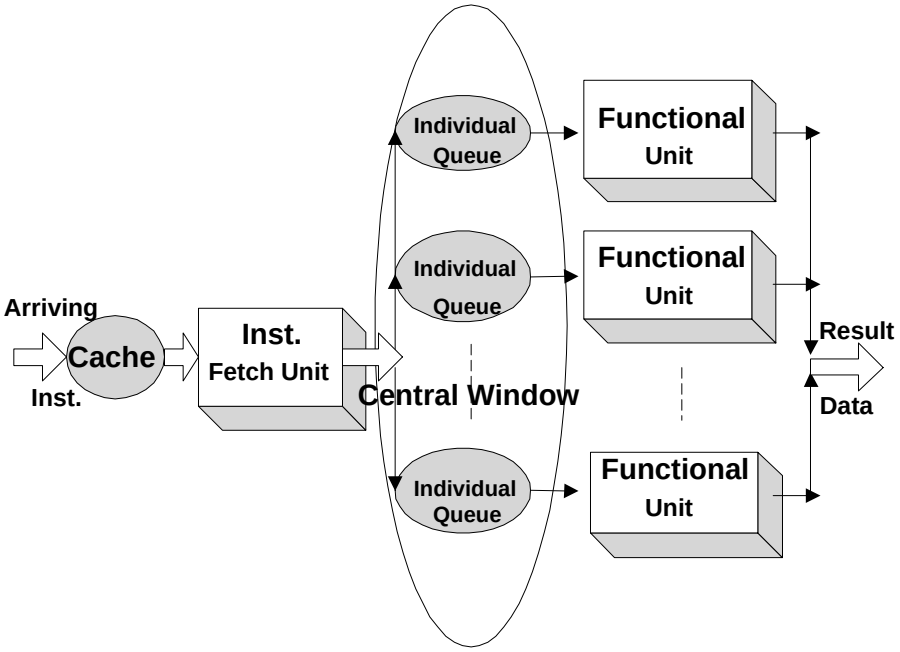

Figure 1: A MCMR Queueing Model on a Superscalar Processor

In our MCMR model of a superscalar processor, 'MC' denotes the multiple instruction classes while 'MR' represents the multiple functional units.

We have a set R of *resources* and a set of C of *customer classes*. Each class in C represents a class of customers that requires a particular set of resource. Specifically, each class-c customer requires some subset R_c of resources $R_c \subseteq R$. Furthermore, the class-c customer requires some number of units, denoted by $c.req$, of each resource $r \in R_c$. In our MCMR model of a superscalar processor, each instruction would come belong to a customer class and before each member of an instruction class can be executed, a number of functional units would need to be acquired.

In front of the functional units is a *central window* which is a shared buffer that holds instructions waiting for execution. If a functional unit is busy, the instructions destined for it will queue in the central window until the functional unit becomes available.

Let $\lambda_c(t)$ denote the *instantaneous arrival rate* of class-c requests, $\mu_c^r(t)$ stand for the service rate of resource r for class-c requests, and $1/\mu_c^r(t)$ denote the instantaneous service time of a class-c request at r. Transient conditions arise when the statistics of the customer arrival processes or the service rates of the resources vary with time, due to externally time-varying factors or dynamic control decisions based on current or delayed system state information.

An arriving class-c customer is blocked at a resource $r \in R_c$ if and only if $c.req$ exceeds the amount of the resource that is currently available. An arriving class-c customer is blocked if and only if it is blocked at any $r \in R_c$. A blocked customer is either lost or retried later. Among the

main performance measures of interest are the instantaneous blocking probabilities of the different classes, instantaneous average number of customers at resources, etc.

To estimate the steady-state performances of such systems, we apply a queueing analysis to the superscalar processor model. Validations against discrete event simulations show the accuracy and advantages of the MCMR superscalar processor model.

3 Model Analysis

Let $|C|$ be the member number of set C in the model. Each class c is an open class with arrival rate λ_c. We denote the vector of arrival rates by $\vec{\lambda} \equiv (\lambda_1, \lambda_2, ..., \lambda_{|C|})$. In the case of superscalar processors, the arrival rates equal the output rate of the instruction fetching unit. We define the instruction class occurrence with $\vec{O} \equiv (O_1, O_2, ..., O_{|C|})$ and let Sp_{fetch} denote the (maximum) number of instructions fetched per machine cycle. With these definitions, the arriving rates can be expressed:

$$\lambda_{c, c \in C^r} = \frac{1}{O_{c, c \in C^r}} Sp_{fetch} \tag{1}$$

Because the throughputs of the classes in open models are part of the input specification, the solution technique for these models is quite simple. We now describe the computation of the various performance measures which can be obtained from the model.

3.1 Processing Capacity

A system is said to have sufficient *capacity* to process a given offered load $\vec{\lambda}$ if it is capable of doing so when subjected to the workload over a long period of time. For multiple class models, sufficient capacity exists if the following inequality is satisfied:

$$\max_r \{ \sum_{c \in C^r} \lambda_c \} < 1 \tag{2}$$

This simply ensures that no functional unit is saturated as a result of the combined loads of all the classes. In the derivations that follow, we will assume that this inequality is satisfied.

3.2 Throughput

By the Forced Flow Law the *throughput* of class c at resource r as a function of $\vec{\lambda}$ is:

$$x_{c, c \in C^r}(\vec{\lambda}) = \lambda_{c, c \in C^r} \tag{3}$$

Every instruction class $c, c \in C^r$ contributes $x_{c, c \in C^r}$ to the sum throughput of the resource r, or a functional unit of a superscalar processor. The throughput of the superscalar processor is the sum of all the throughputs of the individual functional units.

3.3 Utilisation

¿From the Utilisation Law, the *utilisation* is expressed in terms of arriving rate vector $\lambda^r_{c, c \in C^r}$ and the service time demanded by the customer $d^r_{c, c \in C^r}$

$$u_{c, c \in C^r}(\vec{\lambda}) = x_{c, c \in C^r}(\vec{\lambda}) \mu_{c, c \in C^r} = \lambda_{c, c \in C^r} d_{c, c \in C^r} \tag{4}$$

The utilisation of a resource is the product of the throughput of that resource and the average service requirement at that resource. This measures how busy a resource r is serving instructions

from class c. In other words, we can tell from the measurement the proportion of time a functional unit spends servicing instructions from class c.

3.4 Residence Time

As with single class models, the *residence time* is given by

$$Res_c^r(\vec{\lambda}) = d_c^r[1 + N_c^r(\vec{\lambda})] \tag{5}$$

where $N_c^r(\vec{\lambda})$ is the average number of customers seen at server k by an arriving class c customer. The intuition behind this formula is similar to that for single class models. The explanation depends on the scheduling discipline used. For first come first service (FCFS) servers, the residence time is simply the sum of an arriving job's own service time, $v_c\mu_c^r$, where the v_c denotes the average number of visits that a system-level request makes to that resource, and the service times of the jobs already present at the arrival instant, $v_c[N_c^r(\vec{\lambda})\mu_c^r]$, since at FCFS servers all classes must have the same service time at each visit. For processor shared (PS) servers, the residence time is the basic service requirement, $v_c\mu_c^r$, "inflated" by a factor representing the service rate degradation due to other jobs competing in the same queue, $1 + N_c^r(\vec{\lambda})$.

An implication of the assumptions made in constructing separable networks is that the queue length seen on average by an arriving customer must be equal to the time-averaged queue length. Thus, for queueing servers under the FCFS rule:

$$Res_c^r(\vec{\lambda}) = d_c^r[1 + q_k(\vec{\lambda})] \tag{6}$$

where $q_k(\vec{\lambda})$ is the time averaged queue length at server k (the sum over all classes). Applying Little's law:

$$Res_c^r(\vec{\lambda}) = d_c^r[1 + \sum_{j=1}^{|C|} \lambda_j r_{j,k}(\vec{\lambda})] \tag{7}$$

Notice now that the right hand side of the above equation depends on the particular class c only for the basic service demand d_c^r. Thus, $\frac{Res_c^r(\vec{\lambda})}{Res_j^r(\vec{\lambda})}$ must equal $\frac{d_c^r}{d_j^r}$, giving $Res_j^r(\vec{\lambda}) = \frac{d_j^r}{d_c^r} Res_c^r(\vec{\lambda})$. Substituting into the equation above and rewriting, we have:

$$Res_c^r(\vec{\lambda}) = \frac{d_c^r}{1 - \sum_{j=1}^{|C|} u_{j,k}(\vec{\lambda})} \qquad \text{(queueing servers under the FCFS rule)} \tag{8}$$

In our superscalar processor model, the residence time is the time which class c instructions have to spend before they can be served by a functional unit r. The residence time is the basis for computing the system response time, as will be later discussed.

3.5 Queue Length

Assuming a FCFS rule and applying Little's law to the residence time equation above, the queue length of class c at server k, $q_c^r(\vec{\lambda})$, is given by

$$q_c^r(\vec{\lambda}) = \lambda_c Res_c^r(\vec{\lambda}) = \frac{u_{c,k}(\vec{\lambda})}{1 - \sum_{j=1}^{|C|} u_{j,k}(\vec{\lambda})} \tag{9}$$

After the individual queue lengths q_c^r are obtained, the mean queue length of all instruction classes at server k, resource r, can be computed as the sum of the individual mean queue lengths q_c^r of class c at a functional unit.

3.6 System Response Time

The *response time* for a class c customer, $r_c(\vec{\lambda})$, is the sum of its residence times at all sequential devices. More specifically, the instruction fetching unit is in a place sequentially before the other functional units. Consequently, the response time for the class c is:

$$r_c(\vec{\lambda}) = Res_c^r(\vec{\lambda}) + \frac{1}{Sp_{fetch}} \tag{10}$$

The response time for the whole system is given by $\max_{c, c \in C} r_c$. This is also the time it takes each instruction to travel through the entire processor, i.e. the latency.

3.7 Average Number in System

The average number of class c customers in the system can be calculated using Little's law, or by summing the class c queue lengths at all servers:

$$q_c(\vec{\lambda}) = \lambda_c r_c(\vec{\lambda}) = \sum_{r, r \in R^c} q_c^r(\vec{\lambda}) \tag{11}$$

In our superscalar processor model, the average number in the system includes instructions both in central window and the instruction fetching unit buffer. The central window is assumed to account for the most part of the instructions in the system. Using this measure, we can decide if the central window size fits the the usage requests of the application.

3.8 Blocking Probability

The class c customers require R_c resources which are R_c servers of the same service rate μ with buffer size B. Therefore, the birth-death process has the following arrival and service rate:

$$\lambda_n = \sum_{c \in C^r} \lambda_c \tag{12}$$

$$\mu_n = \begin{cases} n\mu & n = 1, 2, \ldots, m-1 \text{ where } C^r \text{ has members less than } m \\ m\mu & n = m, m+1, \ldots, m-1 \end{cases} \tag{13}$$

The following expressions are for the probability of n jobs in the system:

$$p_n = \begin{cases} \frac{\lambda^n}{n! \mu^n} p_0, & n = 1, 2, \ldots, m-1 \\ \frac{\lambda^n}{m! m^{n-m} \mu^n} p_0, & n = m, m+1, \ldots, B \end{cases} \tag{14}$$

In terms of the traffic intensity $\rho = \lambda / m\mu$, we have

$$p_n = \begin{cases} \frac{(m\rho)^n}{n!} p_0, & n = 1, 2, \ldots, m-1 \\ \frac{\rho^n m^m}{m!} p_0, & n = m, m+1, \ldots, B \end{cases} \tag{15}$$

The probability of zero jobs in the system is computed by the relationship

$$\sum_{n=0}^{B} p_n = 1 \tag{16}$$

It gives

$$p_0 = [1 + \frac{(1 - \rho^{B-m+1})(m\rho)^m}{m!(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!}]^{-1} \tag{17}$$

The *blocking probability* is the probability when all the R_c servers and their buffers are used up. We denote it as $p_B^{R_c}$.

$$p_B^{R_c} = \frac{\rho^B m^m}{m!} p_0 \tag{18}$$

The blocking probability for the all the full system is:

$$p_B = \max_{c, c \in C} p_B^{R_c} \tag{19}$$

The blocking probability is determined by the characteristics of the instructions being processed and the processor's configuration. It indicates the likelihood of an arriving instruction being blocked from being served and therefore is an important indicator of congestion and performance.

4 Discrete Simulation

We verified our analytical results by comparing it against that obtained from discrete simulation an equivalent superscalar processor. This was done using a discrete simulation language called QMSM (queueing model simulation) which is similar to discrete simulation languages introduced in [MacDougall, 1987] and [Niels, 1990].

In QMSM view of system, a system comprises a collection of inter-connected *facilities*. A facility may represent some resource of the system. The inter-connection of facilities is determined by the guests' travel routes rather than some explicit statements. Guests represent the active objects of the system. The system behaviour is modelled as the guest movements. A guest can reserve facilities, and schedule future activities. When the resource in request is not available, a guest is queued. A state change of any system object is an event. In a discrete event simulation language like QMSM, an event is collection of the activities happening at the same time instant.

We defined the processor's resources and behaviour using QMSM in our simulation. The functional units of the processor are defined as facilities. The guests of different classes represent the classified instructions. The instructions are classified according to their requests for functional units which execute the instructions. Therefore, we have as many instruction classes as there are functional units. The processor behaviour including instruction fetching, instruction issuing and instruction execution is modeled as different events. To model pipelining, facilities are given buffers the size of the pipeline stages.

To test the model, we chose two benchmarks in the popular SPEC 92 suit: `022.li` from SPECint92 and `030.matrix30` from SPECfp92. Both were executed on a Sun SPARC workstation. Traces for the two SPEC 92 benchmarks were collected by a modified version of the QPT [Ball & Larus, 1991] profiling and tracing software.

¿From analysing the trace, we obtain the occurrences of f different instruction classes. This information is used to generate pseudo-instructions representing the instruction classes in the simulator. These pseudo-instructions are regarded as guests by our simulator. These guests will visit facilities which represents the functional units. The guests undergo the events which represent the benchmark trace behaviour as mentioned before. After all the guests had passed through the simulator, the performance results are calculated.

5 Results

The characteristics of the two benchmarks used, with respect to the functional units, are summarised in Fig. 2

The instructions in the trace obtained on Sun SPARC workstations are divided according to the functional units, namely the integer execution unit (IEU), floating point unit (FPU), load and

Benchmark	IEU Inst.	FPU Inst.	LSU Inst.	PDU Inst.(100%)	Nop	Branch
022.li	38.42%	0	33.23%	4.661320853e+9	1.38%	23.78%
030.matrix300	32.53%	25.51%	38.54%	1.693589255e+9	0.00037%	3.42%

Figure 2: Characteristics of the Benchmarks in Our Scope

store unit (LSU), and prefetch and dispatch unit (PDU). The memory management unit, external cache unit and memory I/O interface unit do not execute instruction directly. Therefore, these do not have their own instruction classes.

Some other SUN SPARC architecture features such as the instruction fetching capacity, integer execution unit pipeline stages, floating point unit pipeline stages, load & store unit pipeline depth and central window size are considered in our simulation.

For the benchmarks characterised above, the analytical results from our MCMR model are given in Fig. 3

Benchmark	Throu.	Ave. Que.	Fetch Unit Que.	Num. in Sys.	Sys. Resp. T.
022.li	1.43300	2.26643	1.50000	6.79930	4.81779
030.matrix300	1.93170	1.94289	1.50000	7.77155	4.86300

Figure 3: Analytical Performance Results

Fig. 4 shows the results of the QMSM discrete simulator.

Benchmark	Throu.	Ave Que.	Fetch Unit Que.	Num. in Sys.	Sys. Resp. T.
022.li	1.30700	1.65367	1.50100	4.96100	3.84225
030.matrix300	2.00000	1.62500	1.50002	6.50000	5.45540

Figure 4: Simulation Performance Results

The analytical and simulation results generally match well. We note that the analytical results show the performance slightly better than the simulation ones do. We attribute the differences to the randomness in the instruction generation process in the simulation. Nevertheless, we can still argue that validations against discrete event simulations show the accuracy and advantages of the MCMR superscalar processor model.

Moreover, according to the results above performance is mostly limited by the instruction fetching capacity. A central window of size exceeding the number of instructions in the system, i.e. `Number in Sys.`, does not significantly improve the performance.

6 Conclusion

In this paper, we proposed a novel queueing model of superscalar processors. To use this model, an instruction trace is analysed once to obtain a few key parameters which characterise the trace. These are then fed into the model to compute the performance of any number of superscalar processor configuration. Our initial results show that, even without considering instruction dependencies, the performance of a superscalar processor is limited by instruction bandwidth. The overall performance bottleneck lies in the instruction fetching unit. The unit limits the growth of instruction bandwidth no matter how fast the rest of processor works. Even very large instruction window sizes do not improve the situation significantly. Encouraged by this, we intend to refine the model to pursue further studies in superscalar processor design.

7 Acknowledgement

We are indebted to M.K. Quek and T.C. Cheng for their assistance in collecting the traces.

References

Austin, Todd M. and Sohi, Gurindar S. (1992) Dynamic Dependency Analysis of Ordinary Programs, *Proc. 19th Int'l Symp. on Comp. Arch.*, 1992, pp. 342-351.

Ball, Thomas J. and Larus, James R. (1991) Optimal Profiling and Tracing Programs, *CS-TR-91-1031*, Computer Science Dept., Univ. of Wisconsin-Madison, USA Jul 1991.

Dubey, Pradeep K. et al. (1994) Instruction Window Size Trade-Offs and Characterization of Program Parallelism, *IEEE Trans. on Comp.*, 43(4), Apr 1994, pp. 431-442.

Jain, Raj (1991) *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc. New York USA 1991, pp. 535-536.

Lam, Monica S. and Wilson, Robert P. (1992) Limits of Control Flow on Parallelism, *Proc. of 19th Int'l Symp. on Comp. Arch*, 1992, pp. 46-57.

Lazowska, Edward D. et al. (1984) *Quantitative System Performance*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA 1984, pp. 135-136.

MacDougall, M.H. (1987) *Simulating Computer Systems Techniques and Tools*, MIT Press, USA 1987.

Matta, Ibrahim and Shankar, A. Udaya (1995) Z-Iteration: A Simple Method for Throughput Estimation in Time-Dependent Multi-Class Systems, *Proc. of SIGMETRICS '95*, Ottowa, Canada 1995, pp. 126-135.

Niels, Houbak (1990) *SIL–A Simulation Language: User' Guide*, Springer-Verlag, Berlin, Germany 1990.

Noonburg, Derek B. and Shen, John P. (1994) Theoretical Modeling of Superscalar Processor Performance, *Proc. of MICRO 27*, San Jose, USA 1994, pp. 53-62.