Interface Design Techniques for Single-Chip Systems

Robert H. Bell, Jr. Lizy Kurian John
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-0240
{belljr, ljohn}@ece.utexas.edu

Abstract

This paper quantifies the performance of typical functional unit interface designs in single-chip systems. We introduce a specific equation to guide the design of optimal module interfaces. We show how the equation and interface considerations lead to more efficient queue structures for request buffering. For a specific single-chip design, we use simulation to show that: 1) For low request rates, queue structure is relatively unimportant to either system request bandwidth or service latency; 2) For a narrow range of request rates, queue structure has a significant impact on system latency but not bandwidth; 3) For high request rates, queue structure impacts bandwidth significantly; 4) As request service latencies increase relative to the queue size, the impact of the queue structure decreases; 5) Given a particular range of request rates, the complexity of particular queue structures can be traded off with the desired system bandwidth and latency performance. For a particular single-chip system, a maximum 29% bandwidth improvement and 60% latency improvement are achieved when using the more efficient queue structures.

1. Introduction

Given the hundreds of millions of transistors available on a single die in today's technologies, multiple processing elements and resources can be integrated onto a single piece of silicon to reduce communication costs between functional units and to reduce overall system power [1]. The processing elements may be cores in a custom or semi-custom general purpose multiprocessor [2, 3], application-specific embedded devices for telecommunications, multimedia, or consumer electronics applications [1, 4]. The resources may consist of on-chip memory or other functional devices. As an example, in the Power4 multiprocessor chip [2], 170 million transistors are used to implement two processor cores and three L2 cache slices. The cache is partitioned to reduce latency to individual cache address locations and to keep physical placement and wiring flexible. The cores and cache slices are fully interconnected.

In a single-chip system, the interconnect between processing elements and resources usually involves significant wiring delay across the chip. To make matters worse, it is predicted that wire impedance will not scale with shrinks of device feature sizes in future technologies [6], which suggests that communication latencies will increase faster than increases in processor performance and memory capacity [1, 5, 6]. The relatively higher communication latencies will necessitate that request control at the processing elements and request handling at the resources be separated by significant numbers of on-chip cycles. To maintain high-performance system bandwidth in the face of increasing delays, the processor must be able to send requests without complete knowledge of whether the resource has consumed prior requests. To do this, the processor assumes a certain amount of queueing near or in the resource. Unless the interface is allowed to drop requests, the available queueing limits the number of requests that a processor can send until a signal indicates that the request has been consumed. Efficient interface designs must optimize this feedback loop.

Multiple processors may be sending requests to a resource simultaneously. In Figure 1, queues associated with each processor arbitrate at a resource for limited load or store ports. Each processor sends requests assuming

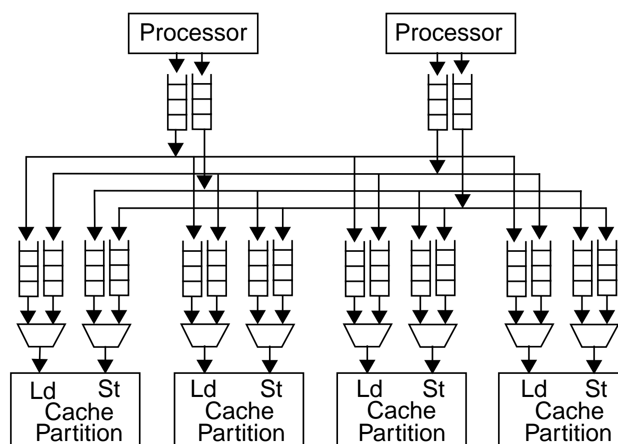

Figure 1: Request Interface in a Single-Chip Design

worst-case traffic from the other processors. Ideally, each processor would have global knowledge of other requests that would interfere with its own traffic. However, supplying global request information to all processors may not be practical due to wiring or placement constraints. Other techniques involving the synchronization of processor activity to eliminate contention may involve complex software or hardware locking mechanisms that impact system bandwidth [7].

The simplest queue structures on an interface maintain requests in FIFO order, i.e. the order in which they are sent to the resource. For specific designs, this constraint may be too restrictive. Alternatives exist in which requests bypass prior requests, but that presupposes some mechanism to maintain fair access to a resource for all requestors, essentially reinstating ordering, which complicates the design process. FIFO queue structures are useful in many kinds of single-chip systems, including the store queues of multiprocessors [2, 8], the request and data queues in streaming DSP devices [4], and ASICs designed from off-the-shelf logic blocks, memory and generic interconnect macros for fast design turn-around.

In this paper, specific FIFO queue structures are examined. The practical aspects of their performance can be understood from both *steady-state* and *burst-mode* analyses. Much prior work examines a high-level queuing system interface using qualitative performance models or analytical techniques that assume infinite queues and no multiple outstanding requests from the processor [9, 10]. Study of the problem of interface synthesis has yielded various interface classes that utilize queues [11], but most works assume FIFO queues without considering the significance of the details of the queue structures [12-14]. Likewise, chip processor simulation studies usually describe the necessity of queues for buffering, but few structural details are provided [15]. Interface classes that permit request drops, such as network-on-chip systems or the processor clustering mechanism described in [15], are beyond the scope of this paper.

In the next section, specific system interface and queue structures are examined, and a design equation is given. Section 3 shows how to design queue structures to enhance performance and examines queue complexity tradeoffs. Section 4 describes the performance simulation system used for this study, gives results for specific queue structures, and examines performance with respect to queue complexity. Section 5 presents the conclusions.

2. Single-Chip System Interface Design

Figure 2 shows a detailed view of an interface for a system. Each processor has a request queue of a specific length at the request port of the resource. The arbitration

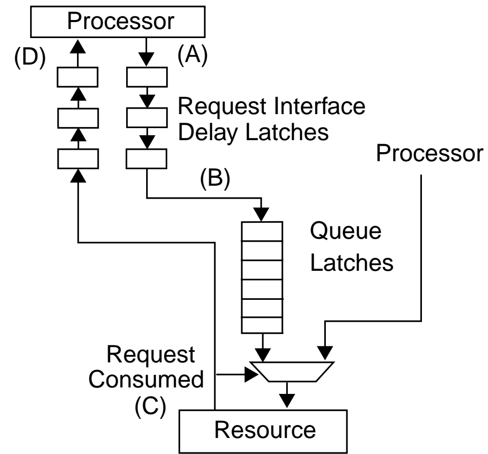

Figure 2: Request Interface Detail

of requests into the resource is determined by the rate at which the resource can consume the requests. If the arbitration is fair and both processors have queued requests ready to be serviced, then each processor will have requests accepted at a maximum rate of one every two cycles, or 1/2. When a request is consumed, the processor is notified over the interface that an additional queue entry is available for another request.

The interface delay in both directions for a specific processor is shown to be three flow-through cycles. Its resource queue is a six-entry simple linear queue, which adds six additional cycles of delay for a particular request to become visible at the resource port. The total number of cycles in the feedback loop from processor request to receiving the notification that the request has been consumed is therefore twelve cycles.

For each processor, the total number of requests outstanding can not be higher than the number of queue entries since it is not guaranteed that the resource will consume any given request. The resource may not accept a request if it is busy servicing the request of the other processors, or if it is waiting to resolve internal resource contention or waiting for data to service previous requests. In the example, the processor can maintain only six requests unanswered since it has only six queue entries.

Assuming a steady-state system in which both processors request at the maximum rate and the resource can service one request every cycle, the consumption rate at the processor interface will approach 1/2, so the request rate out of the processor will approach 1/2. Over the twelve cycle feedback loop, six requests on average can be serviced at a rate of 1/2. Table 1 shows the first six steady-state requests as they arrive at points A, B, C and D in Figure 2. Since confirmation of acceptance of the first request arrives in cycle 12, the processor can send the next request (request 7) in that cycle, and the pattern of a request every other cycle is maintained with the steady-state request rate

matching the consumption rate.

Since this number of serviced requests matches the maximum number of outstanding processor requests, the design has been optimized for maximum bandwidth. Defining N to be the maximum number of outstanding processor requests, M the delay through the queue, G the maximum service rate at the resource, and D the one-way interface delay, then the design is optimized for steady-state performance in the presence of worst-case traffic from other processors when:

$$2D + M = \frac{N}{G} \qquad (1)$$

If the left-hand side of the design equation becomes larger than the right-hand side, cycles will be lost while the processor waits for confirmation of request acceptance from the resource. This implies that the interface delay can be reduced without affecting steady-state bandwidth. If the left-hand side becomes smaller than the right-hand side, then it is possible that the queue has been designed with too many entries and can be simplified. In either case, the interface design is not matched to the worst-case steady-state service rate at the resource.

3. Improving Interface Delays

The interface design equation (1) shows that interface delay can be increased if the maximum request service rate at the resource decreases. Likewise, if the service rate increases at the resource, the interface delay can be decreased. If the interface is designed assuming each processor requests at the maximum rate, too much delay may be built into the interface to provide maximum bandwidth if one processor stops requesting for a while. On the other hand, in a single-chip design spread out over a large area of silicon, there is a minimum delay between modules that must exist. Usually the intrinsic delay between processing elements and resources is not negotiable.

However, if the delay on the interface can be reduced dynamically, then requests can potentially be serviced by the resource more quickly if fewer requests appear at the port from other processing elements. One way to dynamically reduce interface delay is to design queues that incorporate request bypassing to lower level entries. Figure 3 shows two six-entry queue designs with one and two levels of bypass. If a queue entry fed by a bypass mux is open

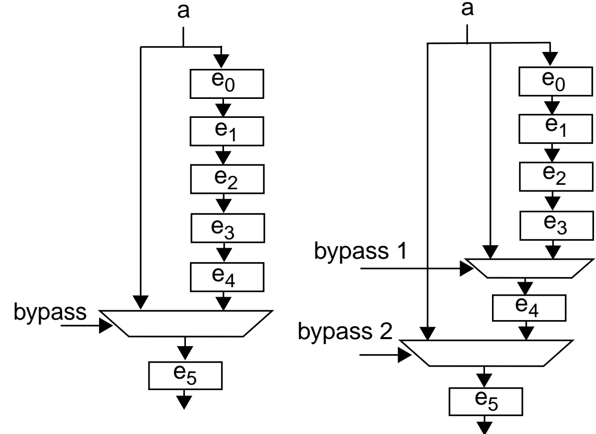

Figure 3: 6-Entry 1- and 2-level Bypass Queues

and no requests reside in the upper entries, a new request can bypass to the lower entry.

Bypassing accomplishes three things. First, even though the processor can not send down more requests than the number of available queue entries, it can group those requests to appear more quickly at the resource port. If the queue is empty, the requests can *burst* down to the lower entries in the queue and be serviced more quickly because the delay through the queue is reduced for the first few requests. This reduces the latency to receive the acceptance indication back at the processor, and more requests can be sent down sooner. In the interface design equation, the M is effectively reduced for the first few requests, even though it will be at its maximum for the majority of the requests and G has not changed. This affinity for bursts of requests separated by quiescent cycles enhances latency performance for some workloads even though overall bandwidth may not increase.

Second, if the resource is in a position to service every processor request that appears, i.e. the other processors are idle and no internal contention or waiting are occurring, then, in the interface design equation, in addition to an increase in G, the M is effectively reduced over a significant amount of time, and overall interface delay is reduced. Interface matching occurs dynamically. In this way, a system can be designed to maximize performance using steady-state delay matching under worst-case traffic assumptions and yet still accommodate increased performance in those cases when request traffic is limited to one or a few processing elements. The design shows an

Table 1: Steady-State Requests (and Cycle of Arrival at a Location in the System of Figure 2)

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}
A	1		2		3		4		5		6										
B				1		2		3		4		5		6							
C										1		2		3		4		5		6	
D													1		2		3		4		5

increased robustness in the face of changing workload characteristics, and system bandwidth is improved.

Third, a single-chip system designed with bypass queues may show improved system scalability as process technologies change. While interface delays will increase because wire resistance and capacitance do not scale, the bypass queues implemented at the resources may not need to change if request traffic is often bursty and resource service rates remain efficient. With this enhanced performance scalability in the face of technology changes, the time needed to reimplement a design to a new technology decreases.

Note that the design equation implies that more than NG bypasses will not improve queue performance under worst case traffic assumptions. Since only NG requests can be serviced in N cycles, any request in the top queue entry will have time to propagate to the lowest entry and be serviced at the maximum rate.

3.1. Choosing the Number of Bypasses

The optimal number of bypasses in a design is a function of the system performance under the set of workloads targeted to run on the design and the increase in complexity as the number of bypasses increases. If there is a large variety of workloads, more bypasses may be indicated in order to deal effectively with the corner-case characteristics of some applications. If the design under consideration is an application-specific design, a specific number may be optimal.

As the number of bypasses increases, the control logic necessary to choose where to place requests in the queue increases. Muxes are needed to steer requests from the input to an entry or to move requests down in the queue. Since requests usually have addresses and control information associated with them, and in some cases data, additional muxes and control lines need to be implemented for each bit being moved. These increases lead to increased silicon usage and power consumption.

The changes needed in the control equations to implement bypassing give an indication of the increased logic complexity. For an N-entry linear queue without bypasses, a request shifts from the previous entry when the queue unloads to the resource (G is set) or when this entry or one of the entries below in the queue does not contain a request:

$$e_i shf = G + \bigcup_{j=i}^{N-1} \bar{e}_j$$

Looking at Figure 3, for a queue with B bypasses, if the queue unloads to the resource, the top entry is shifted if one or more of the entries down to the first bypass has a request, and the other entries shift. If the queue does not unload, the top entry shifts if one of the entries down to the first bypass is empty and the bypass entries are full or one of the non-bypasses has a request. The other entries shift if there is an entry without a request:

$$e_0 shf = G \bigcup_{j=0}^{N-B-1} e_j + \left(\bigcup_{j=0}^{N-B-1} \bar{e}_j \right) \left(\bigcap_{j=N-B}^{N-1} e_j + \bigcup_{j=0}^{N-B-1} e_j \right)$$

$$e_{i \neq 0} shf = G + \bigcup_{j=i}^{N-1} \bar{e}_j$$

If a entry bypasses it controls the mux and overrides the shift into that entry. An entry bypasses if there are no requests in the entries above the entry and it has a request and either the queue unloads or one of the bypasses below it is empty, or the queue does not unload and the bypasses below it have requests:

$$e_{i=\{N-B,\ldots,N-1\}} byp = \left(\bigcap_{j=0}^{i-1} e_j \right) \left(e_i \left(G + \bigcup_{j=i+1}^{N-1} \bar{e}_j \right) + \bigcap_{j=i+1}^{N-1} e_j \right)$$

Not only are the shift and bypass equations more complicated over the linear queue case, each additional bypass adds an extra AND or OR signal into each term of the bypass equation. The levels of logic can increase significantly. The cost is either delay or silicon area to replicate terms. In addition, more complex queue structures may complicate the simulation and verification efforts.

To quantify the impact of bypassing, the queue bypass configurations for the six-entry queue were coded in VHDL and synthesized with IBM's Booledozer tool to find the relative control logic area increases with respect to a linear queue. Synthesis used a typical family of scannable latches and logic gates with no more than four inputs or five outputs allowed per gate. The synthesis was executed so as to keep timing delay the same for all configurations at the cost of silicon area. Modelling a one nanosecond cycle time with intrinsic gate delays of around 50 picoseconds, static timing analysis showed no more than 17 picoseconds of slack difference between any two configurations. Table 2 shows the resulting area increases. Slightly larger increases are seen moving from one to two bypasses, and from four to five bypasses. Enhanced per-

Table 2: Area Increases for 6-Entry Queue with Bypasses

Number of Bypasses	Area Increase vs. No Bypass
1	2.5%
2	8.2%
3	10.9%
4	12.6%
5	18.1%

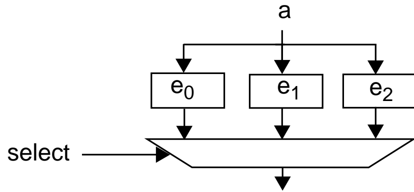

Figure 4: A 3-Entry Generalized Queue Structure

formance and scalability for a particular system and application may justify a particular area increase.

3.2. Alternative Queue Designs

A more generalized queue structure is sometimes used, as shown in Figure 4. However, the queue itself no longer enforces the ordering of the requests. Additional bits of information must be maintained to indicate which request is the earliest. That information must be factored into the mux select and the shift and hold control equations for the queue entries. The generalized queue offers additional algorithmic flexibility at the cost of significantly more complicated control logic.

4. Performance Analysis

To verify the queue performance described in the previous section, the performance model shown in Figure 5 was developed. It consists of two processors, one on-chip L2 cache partition, and an infinite off-chip L3 cache. The first processor feeds requests through four flow-through latches to a 6-entry queue feeding the L2 cache partition. The 6-entry queue can be configured with from zero to five bypasses. The second processor is modelled as a continuous requestor to simulate worst-case mux traffic, but no real requests are generated.

The first processor generates requests statistically with a configurable request rate when not held off by a lack of L2 acceptance confirmations, i.e. the queue appears full. The cache partition models realistic state machines to buffer requests, arbitrate for the L2 directory and cache, cast out data to and read data from the L3, and wait for data operations in the cache. Buffered requests wait for prior addresses in the same congruence class to complete operations. L2 hit rates, castout and address collision rates are statistical and configurable. The best-case latency from request acceptance to completion is 25 cycles. The latency to castout data to the L3 is 40 cycles, and to read data from the L3 is 35 cycles. No data reloads to the processor are modelled.

Requests are consumed as the L2 sends them to the cache. Requests can also *gather* into an eight-request block at a rate of 75%. Gathering implies that specific requests and their data can be treated together as a single blocked request to the cache. This effectively increases the

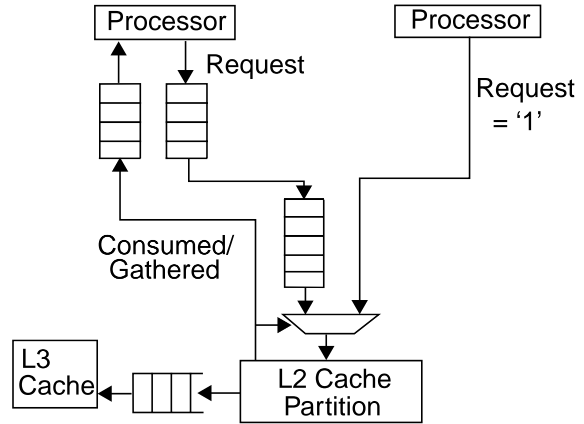

Figure 5: Performance Verification Model

L2 service rate for the purposes of these experiments. It takes four cycles for the processor to receive confirmation that the request has been consumed or gathered.

In the first experiment, the L2 cache is assumed to be infinite, i.e. all request addresses hit in the cache, and there are no castouts or address collisions. Figure 6 shows the number of requests that are processed in 100k cycles as request rates are increased with the queue configured for zero to five bypasses. For request rates below 35% of cycles, there are not enough requests to distinguish the queue structures, and requests increase linearly.When the

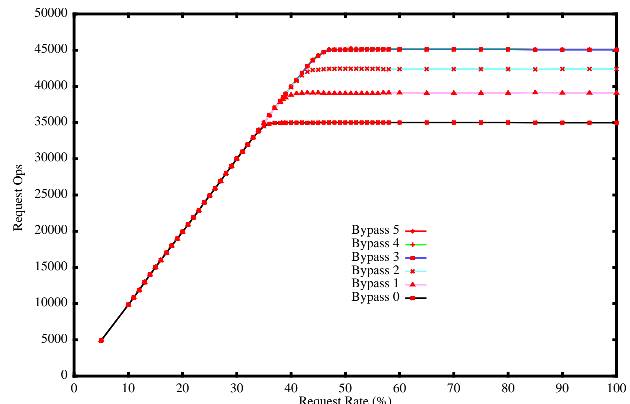

Figure 6: Requests vs. Request Rate, Infinite L2

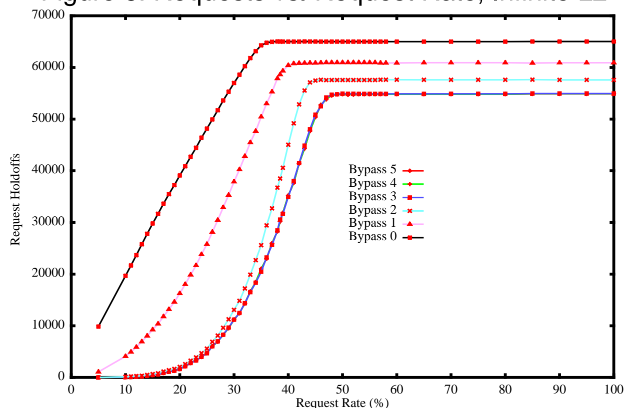

Figure 7: Holdoffs vs. Request Rate, Infinite L2

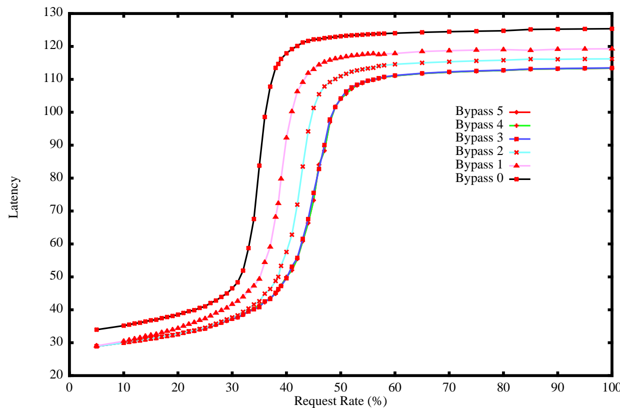

Figure 8: Latency vs. Request Rate, Infinite L2

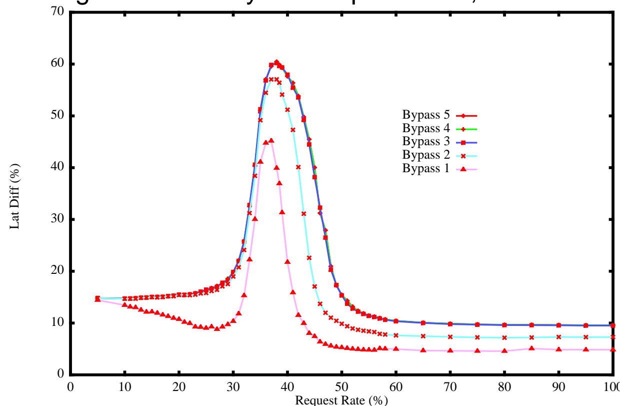

Figure 9: Latency Differences (vs. 0 Byp.), Inf. l2

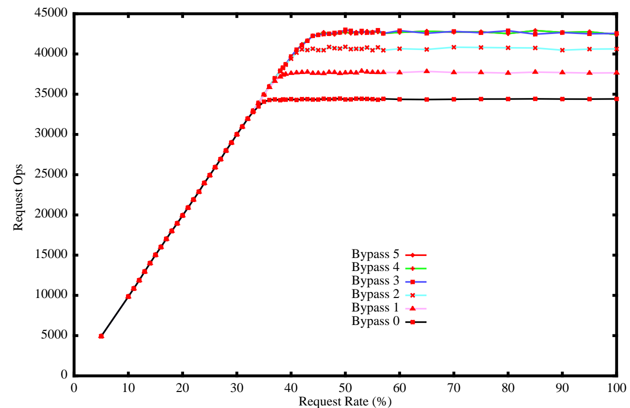

Figure 10: Requests vs. Request Rate, Infinite L3

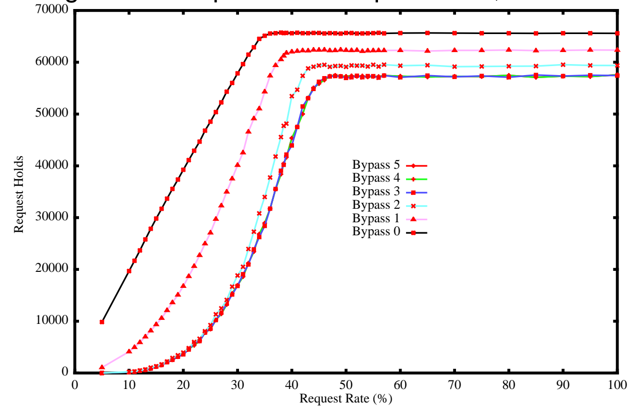

Figure 11: Holdoffs vs. Request Rate, Infinite L3

request rate matches the inherent service rate limit of the L2, steady-state is reached. This state is reached later as the queue is configured with more bypasses. As predicted, using more than three bypasses does not improve queue performance. A 29% bandwidth improvement over the no-bypass case is achieved.

Figure 7 shows the number of cycles that requests are held off at the processor because the queue appears filled, and Figure 8 shows the latency increases that result. Over a relatively narrow band of request-rate increases the latency explodes. This is consistent with a *linearized* queue in combination with an inherent maximum service rate at the resource. A linearized queue is a queue in which requests are in the top entries most of the time so that bypassing is ineffective.

In Figure 9, the latency differences with respect to a linear queue are shown. For low request rates, there are not enough requests to make good use of the bypass capability. For moderate request rates and three or more bypasses, a 60% latency improvement over the linear queue is shown. Note that this improvement occurs in the presence of worst-case mux usage by the other processor. For high request rates, the queue is essentially linearized and little improvement is seen.

Figures 10 through 13 show the same curves for a more realistic L2 and infinite L3. To represent a warmed up cache, the hit rate in the L2 is set to 80% of requests, and the castout and address collision rates to 30% and 1% of requests, respectively. The curves resemble those for the infinite L2, but the maximum bandwidth improvement has dropped to 24% and the latency improvement has dropped to 53%. Holdoffs to the processor increase faster and are higher for three or more bypasses. The curves have lost some smoothness due to the more realistic L2 behavior and L3 interactions. As the L2 hit rate decreases, because of the L3 latency effects, queue effects represent a lower percentage of the overall latency, and bypass effectiveness decreases. Experiments for low L2 cache hit rates have shown that, for moderate and high request rates, bypassing actually decreases performance relative to the linear queue because requests are less evenly paced to the resources, which increases contention for L2 state machines and caches.

Looking at the area increases in Table 2, and the narrow region of bypass effectiveness shown in Figure 9, and considering the reduction in effectiveness when the system has a more realistic memory hierarchy as in Figure 13, a cost-effective queue design might include no more than two or three bypasses. It should be noted that many applications feature request patterns that are bursty in nature. Bursts of operations separated by many quiescent cycles may achieve much higher performance using bypass

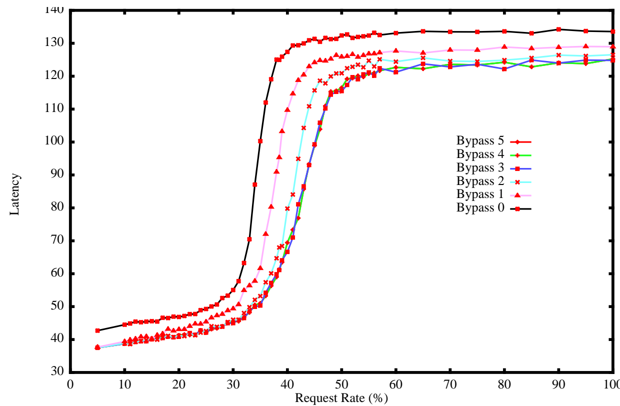

Figure 12: Latency vs. Request Rate, Infinite L3

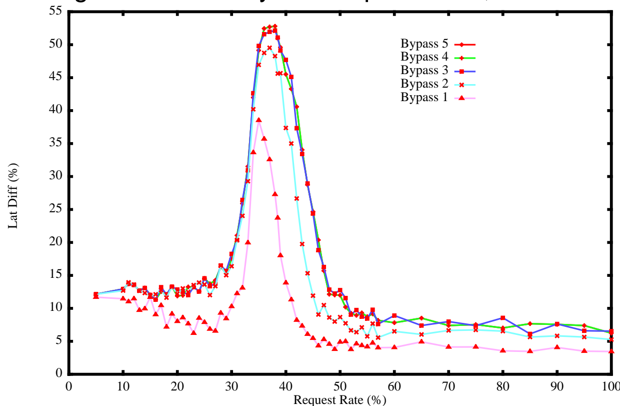

Figure 13: Latency Differences (vs. 0 Byp.), Inf. L3

queues than the performance demonstrated here for requests issued with a mean rate. Future work will explore the performance of bypass queues for burst request patterns in single-chip systems.

5. Conclusions

This paper quantifies the performance of a typical module interface design in a single-chip system. We introduced a specific interface design equation to guide the design of optimal module interfaces for steady-state performance. We also show how the equation and interface considerations lead to a more efficient queue structure design to increase system performance in burst request situations. For a specific design using system simulation, it was shown that system bandwidth and latency effects vary as request rates change. For sparse request rates, queue structure is relatively unimportant to either system bandwidth or latency. For a narrow range of request rates, queue structure has a significant impact on system latency but not bandwidth. For high request rates, queue structure impacts bandwidth significantly. As request service latencies increase relative to the queue size, the impact of the queue structure decreases. Given a particular range of request rates, the complexity of particular queue structures can be traded off with the desired system bandwidth and latency performance.

6. References

[1] L. Benini and G. De Micheli, “Networks on Chips: A new SoC paradigm,” IEEE Computer, January 2002, pp. 70-78.

[2] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le and B. Sinharoy, “POWER4 System Microarchitecture,” IBM Journal of Research and Development, January 2002, pp. 5-25.

[3] K. Diefendorff, “Power4 Focuses on Memory Bandwidth,” Microprocessor Report, October 6, 1999.

[4] J. Friedman and Z. Greenfield, “The TigerSHARC DSP Architecture,” IEEE Micro, January 2000, pp.66-76.

[5] J. Huk, S. W. Keckler and D. Burger, “Exploring the Design Space of Future CMPs,” IEEE Conference on Parallel Architectures and Compilation Techniques, Oct. 2001, pp. 199-210.

[6] V. Agarwal, M. S. Hrishikesh, S.W. Keckler and D. Burger, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures,” IEEE Symposium on Computer Architecture, 2000, pp. 248-259.

[7] J. Hennessey and D. Patterson, Computer Architecture: A Quantitative Approach, Second Edition, Morgan-Kaufman Publishing Co., 1996.

[8] M. Johnson, Superscalar Microprocessor Design, PTR Prentice-Hall, 1990.

[9] M. C. Chiang and G. S. Sohi, “Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment,” IEEE Transactions on Computers, Vol. 41, 1992, pp. 297-317.

[10] K. S. Trivedi, Probability and Statistics, Englewood Cliffs, Prentice-Hall, 1982.

[11] A. Rajawat, M. Balakrishnan, A. Kumar, “Interface Synthesis: Issues and Approaches,” IEEE Conference on VLSI Design, 2000, pp. 92-97.

[12] S. Narayan and D. D. Gajski, “Synthesis of System-Level Bus Interfaces,” IEEE European Conference on Design Automation, 1994, pp. 395-399.

[13] B.I. Park, I.C. Park and C. M. Kyung, “Interface Synthesis for IP-Based Design,” IEEE Asia Pacific Conference on ASICs, 2000, pp.227-230.

[14] A. Baganne, J. L. Phillippe and E. Martin, “A Formal Technique for Hardware Interface Design,” IEEE Symposium on Circuits and Systems, June 1997, pp. 1592-1595.

[15] J.M. Parcerisa, J. Sahuquillo, A. Gonzalez and J. Duato, “Efficient Interconnects for Clustered Microarchitectures,” IEEE Conference on Parallel Architectures and Compilation Techniques, September 23, 2002.