Report Number: WUCS-00-24

2000-01-01

Design Tradeoffs for Embedded Network Processors

Tilman Wolf, Mark Franklin, and Edward W. Spitznagel

Demands for flexible processing has moved general-purpose processing into the data path of networks. With the development of System-On-a-Chip technology, it is possible to put several processors with memory and I/O components on a single ASIC. We present a model of such a system with a simple performance metric and show how the number of processors and cache sizes can be optimized for a given workload. Based on a telecommunications benchmark we show the results of such an optimization and discuss how specialied hardware and appropriate scheduling can further improve system performance.

Design Tradeoffs for Embedded Network Processors

Tilman Wolf, Mark Franklin and
Edward W. Spitznagel

WUCS-00-24

July 2000

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Design Tradeoffs for Embedded Network Processors

Tilman Wolf, Mark A. Franklin and Ed Spitznagel
{wolf, jbf, aqualung}@ccrc.wustl.edu
Departments of Computer Science & Electrical Engineering
Washington University, St. Louis, Missouri

Abstract

Demands for flexible processing has moved general-purpose processing into the data path of networks. With the development of System-On-a-Chip technology, it is possible to put several processors with memory and I/O components on a single ASIC. We present a model of such a system with a simple performance metric and show how the number of processors and cache sizes can be optimized for a given workload. Based on a telecommunications benchmark we show the results of such an optimization and discuss how specialized hardware and appropriate scheduling can further improve system performance.

1 Introduction

Over the past decade there has been rapid growth in the needs for reliable, robust, and high performance communications networks. This has been driven in large part by the demands of the internet and general data communications. To adapt to new protocols, services, standards, and network applications, many modern routers are equipped with general purpose processing capabilities to handle (e.g., route and process) data traffic in software rather than dedicated hardware. Design of the network processors associated with such routers is a current and competitive area of computer architecture [2], [6], [7], [8], [9], [12]. This paper is aimed at examining certain tradeoffs associated with the design of these embedded network processors.

In the current router environment, single processor systems generally cannot meet network processing demands. This is due to the growing gap between link bandwidth and processor speed. Broadly speaking, with the advent of optical WDM links, packets are arriving faster than single processors (even with their continuing impressive gains in performance) can deal with them. However, since packet streams only have dependencies among packets of the same flow but none across different flows, processing can easily be distributed over several processors. That is, there is an inherent parallelism associated with the processing of separate independent packet flows. Thus, the problems of complex synchronization and inter-processor communications that are typically encountered when dealing with parallelization of many traditional (e.g., scientific) computer applications are not present. From a functional and performance standpoint it is therefore reasonable to consider developing network processors as parallel machines. Indeed, current commercial network processors contain from two [9] to sixteen processors [2].

Advances in VLSI technology have also pushed integration to the point where it is now possible to design and implement multiple CPU network processors, with cache and DRAM, on a single silicon chip. As a result, the memory access latency is reduced and clock rates can be increased. This technology, though, is limited by the maximum chip area that can be manufactured at a reasonable cost. One important design decision for such multiprocessor chips is how many processors and how much associated cache should be placed on a single chip. This is important since, for a given chip size, more processors imply smaller caches and smaller caches lead to higher fault rates. High fault rates,

in turn, impact performance and also the required off-chip memory bandwidth. This bandwidth requirement is yet another important design constraint and is one which, in certain cases, has led to the adoption of Rambus [10] and other techniques. This paper is concerned with determining the optimal configuration of processors and caches for a given network oriented workload and chip size. The following design parameters are considered:

- Number of processors.
- Size of on-chip caches.
- Number of I/O channels.
- Processing workload.

A performance model whose metric is the total processing power of a network processor in terms of instructions per second is developed. The model is used to explore the design space (e.g., set of parameters above) associated with developing a single chip network processor which has multiple processing units on the chip. The evaluation is performed using an analytical model that takes cache configurations, I/O requirements, and workload characteristics into account. It is shown how the overall system can be optimized for maximum processing power for a given workload. As an example, optimization results are presented using statistics gathered from a benchmark of programs (CommBench [13]) which has been designed to reflect activity typical of network processing applications.

Section 2 that follows characterizes the problem in more detail and provides an overall system design. Section 3 covers analysis of the optimization problem. Section 4 introduces a benchmark that is used as a sample workload and Section 5 shows the system optimization results. Section 6 discusses extensions to the system and Section 7 summarizes the work.

2 Design Issues With Multiple Processor Systems-On-a-Chip

As a base model for the analysis, we use the multiprocessor system described in this section.

2.1 Background

There are a host of advantages associated with integrating multiple processing units on a single chip and developing what is referred to as a SOC (System-On-a-Chip) network processor. Chief among them are the ability to achieve higher performance and, by using fewer chips, lower cost. Such implementations are however limited by the size of the chip (i.e., silicon real estate) that is feasible (for cost and technology reasons), the packaging technology that can be utilized (to achieve given pin requirements), and the power which can be dissipated (at a given frequency).

Network processors are used in routers to flexibly process data streams where such data streams can be in the form of either fixed-bandwidth cell-stream connections, or packet datagrams. In either case, there is a real-time bound on the time duration permitted for processing a given packet. On a heavily loaded link, data is sent back-to-back in a virtually continuous manner and if the system is to be responsive and reliable, processing of a packet cannot take more than the average interarrival time.

With gigabit links, for example, packet interarrival time is on the order of hundreds of nanoseconds. SRAM access times on the other hand are about $10ns$. Thus, with a single processor, only a limited amount of processing can be done before the next packet arrives. However, to increase the amount of processing possible between packet arrivals, one can make use of a basic network traffic property. That is, packets from different data streams can be processed independently and thus parallel processors can be used in this situation without the need for complex synchronization and inter-processor communication. Each processor in such a design handles data packets from different flows and, when a processor becomes idle, a system level scheduler routes the appropriate packet

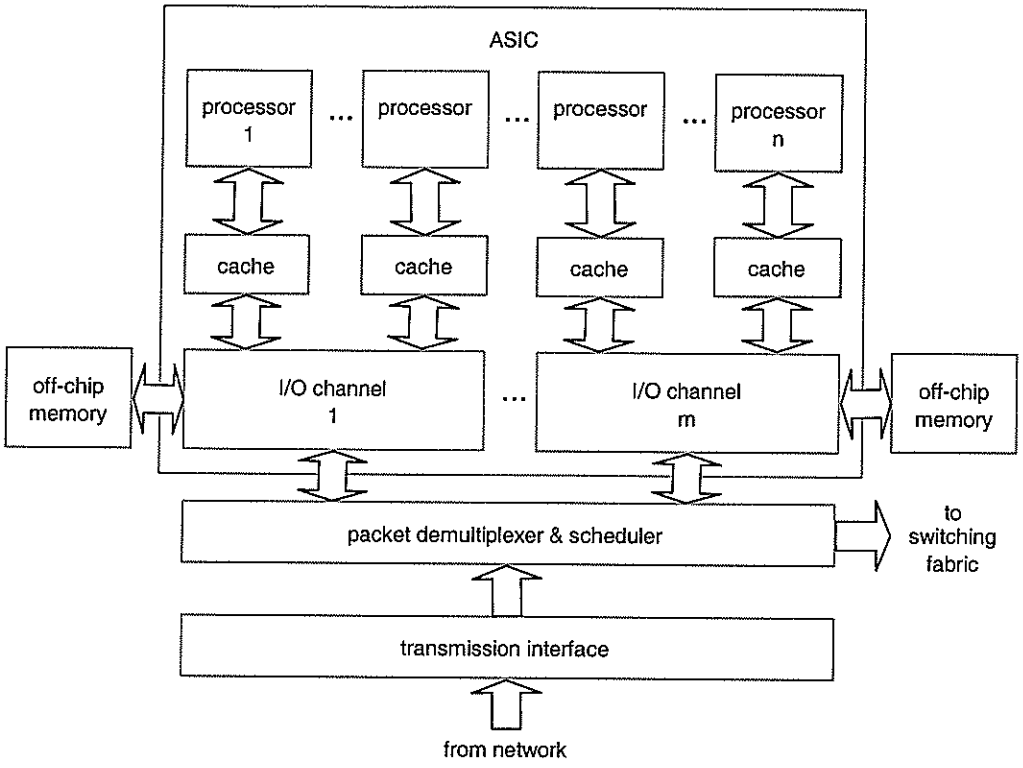


Figure 1: Multiprocessor Router Port Outline.

to a processor as necessary. Thus, with an ideal scheduler and n processors, the amount of time to process a packet is extended to n times the packet interarrival time.

With a dozen processors this time can be pushed into the range of microseconds. Note also that the processors can utilize very simple real-time operating systems since they always process only a single packet and generally will not require multitasking and associated complex context switching capabilities.

In the next section a functional design for a single chip multiprocessor SOC is presented. In use, groups of such chips would be placed on router line cards and appropriately adapted to various link speeds.

2.2 Functional Design

For the remainder of the paper we focus on a single architecture that follows the basic ideas described above. The SOC consists of multiple independent processing engines. The memory hierarchy consists of on-chip, per processor instruction and data cache, and shared off-chip memory. Memory accesses are done through several I/O channels that are shared among sets of processors. The I/O channels are also used by the system controller/scheduler to send packets requiring processing to the individual processors. The overall system is shown in Figure 1.

Typically, a packet is first received and reassembled by the *Transmission Interface* on the input port of the router. The packet then enters a *Packet Demultiplexer* which uses packet header information to determine the flow to which the packet belongs. Based on this flow information the *Packet Demultiplexer* now decides what processing is required for the packet. The packet is then enqueued until a processor becomes available. When a processor becomes available, the packet and the flow information is sent over an I/O channel to one of the processors on the network processor chip. After processing has completed, the packet is returned to the *Packet Demultiplexer* and enqueued before

Component	Symbol	Description
Processors p	n	number of processors on chip
	clk_p	processor clock rate
Caches	c_i	instruction cache size
	c_d	data cache size
	$linesize$	cache line size of instruction and data cache
Program	$compl$	complexity (instructions per byte of packet)
	f_{load}	frequency of load instructions
	f_{store}	frequency of store instructions
	m_{ic}	instruction cache miss rate for cache size c_i
	m_{dc}	data cache miss rate for cache size c_d
	d_{cd}	probability of dirty bit set in write-back data cache of size c_d
I/O Channels io	m	number of I/O channels io
	l_{io}	load on I/O channel
	w_{io}	width of one I/O channel
	clk_{io}	clock rate of I/O channels
Memory	t_{mem}	memory access time for one cache line (see Appendix A)
ASIC	s_x	actual size of component x, with $x \in \{ASIC, p, c_i, c_d, o\}$

Table 1: System Parameters.

being sent through the router switching fabric to its designated output port.

Note, in this design it is assumed that the network processor chip appears at the router's input ports. Alternatively, it can be positioned at the router's output ports. Its position however will have little influence on the design issues considered in this paper. A more detailed functional description of the above design can be found in [14]. Here, we consider the single chip design optimization problem associated with selection of the:

- Number of processors on the chip.
- Cache size per processor cache (and the split between instruction and data cache).
- Number of I/O channels.

We assume that the processors are general purpose RISC processors that can execute one instruction per cycle if no cache misses occur. Thus the access time to on-chip cache is one cycle. The I/O channel is assumed to be in the style of a Rambus interface [10].

3 Analysis

Given that we are interested in the amount of traffic the system can handle, we view the design problem as one of selecting the parameter values which maximize the throughput assuming chip area constraints, reasonable technology parameters, and the operational characteristics of a benchmark of network processing programs.[1]

Throughput in this environment corresponds to the number of packets that can be processed in a given time. This is determined by a combination of the instruction processing requirements of a given application (e.g., number of instructions necessary for routing table lookup, packet encoding,

[1]In this treatment we do not consider latency issues and assume that these requirements are met if the design can keep up with the incoming packet rate.

etc.), and the number of instructions that can be executed per second on the network processor. We assume that all packet processing tasks are performed in software on RISC microprocessors (specialized hardware is considered in Section 6.1). Thus, the throughput is proportional to the number of Instructions Per Second (IPS) that can be executed on the system. Given a typical RISC instruction set, network application benchmark characteristics (e.g., fault rates for different cache designs), and various other parameters (e.g., CPU clock rate, cache miss times, etc.), an optimal system configuration, that maximizes IPS, can be determined. For a given chip size, this configuration falls between the following two extremes:

- Too few processors: Each processor has enough cache to execute programs efficiently (i.e., with very low fault rates) but the total number of processors is too low to achieve high throughput.
- Too many processors: Each processor has very little cache and thus the number of cache misses and off-chip memory accesses is high. The processors spend most of the time stalling and, as a result, have a high Clocks Per Instruction (CPI). Application processing is therefore slow and throughput low.

In the remainder of this section an analytic model reflecting the interactions between these various items is developed and an optimized system determined.

3.1 Configurations

We begin by defining the fundamental chip area limitations that are present. The network processor chip size limits the number of processors, n, the amount of instruction and data cache (assumed to be identical for each processor), and the number of I/O Channels, m, that may be present (parameters and definitions may be found in Table 1). If $s(ASIC)$ is the size of the network processor chip, $s(p_k)$, $s(c_{i_k})$ and $s(c_{d_k})$ are respectively the sizes of a processor k, instruction cache c_{i_k}, and data cache c_{d_k}, and $s(io)$ the size of one I/O channel, then all solutions must satisfy the following inequality:

$$\sum_{k=1}^{n} (s(p_k) + s(c_{i_k}) + s(c_{d_k})) + \sum_{k=1}^{m} s(io) \leq s(ASIC). \tag{1}$$

With identical processors, cache configurations, and I/O channels this becomes:

$$n \cdot (s(p) + s(c_i) + s(c_d)) + m \cdot s(io) \leq s(ASIC). \tag{2}$$

Further, we can assume that the best performance is achieved with a set of design parameters which result in an area as close to $s(ASIC)$ as possible. That is, we need to investigate only configurations that try to "fill" the available chip area.

3.2 Single Processor

We continue with a simple performance model for a single processor and then extend that to multiple processors. Assume a simple RISC design where the ideal CPI is 1. The actual number of instructions that can be executed with a given instruction cache size (in bytes) c_i and data cache size c_d depends on the number of off-chip memory accesses (e.g., instruction cache miss or data cache miss) and the time associated with each of these accesses. The average number of off-chip memory accesses per instruction, M, can be expressed as:

$$M = m_{ic} + (f_{load} + f_{store}) \cdot m_{dc} \cdot (1 + d_{cd}) \tag{3}$$

where m_{ic} and m_{dc} are the instruction and data cache miss rates, f_{load} and f_{store} are the frequencies associated with load and store instructions, and d_{cd} is the probability of a dirty bit set in the write-back data cache (a write-back caching mechanism is assumed).

Given an off-chip memory cache line access and transfer time, t_{mem}, and a processor clock rate clk_p, the average cycles per instruction, CPI, is:

$$CPI = 1 + M \cdot t_{mem} \cdot clk_p. \tag{4}$$

The total number of instructions that can be carried out per second, IPS_1, by a single processor is thus:

$$IPS_1 = \frac{clk_p}{CPI} = \frac{clk_p}{1 + M \cdot t_{mem} \cdot clk_p}. \tag{5}$$

The amount of average off-chip memory bandwidth, BW_{mem_1}, required by a single processor can now be expressed in terms of the number of instructions per second, IPS_1, the number of memory accesses per instruction, M, and the cache *linesize* associated with each off-chip memory. Thus:

$$BW_{mem_1} = IPS_1 \cdot M \cdot linesize = \frac{clk_p \cdot M \cdot linesize}{1 + M \cdot t_{mem} \cdot clk_p}. \tag{6}$$

The I/O channel bandwidth required to transfer packets from the packet demultiplexer to the processors is relatively small compared to the bandwidth generated by memory accesses, and is therefore not considered any further.

3.3 Multiple Processors

The single processor analysis can now be extended to the case where there are multiple processors on the chip. With n processors present, Equation 5 becomes:

$$IPS = \frac{n \cdot clk_p}{CPI} = n \cdot IPS_1. \tag{7}$$

If contention for the I/O system is ignored, and if all the processors are executing the same workload, then the bandwidth to memory can be expressed simply as:

$$BW^*_{mem} = IPS \cdot M \cdot linesize \tag{8}$$

Equations 7 and 8 however assume that the I/O system has sufficient bandwidth so that contention between processors does not increase the memory access time t_{mem} since, if t_{mem} increases, IPS will decrease (see Equation 5).

To account for contention and potential queueing delays it is convenient to first define the parameter l_{io}, $0 \leq l_{io} \leq 1$, as the load on the I/O channel associated with the n processors and their memory requests (i.e., M). A value of l_{io} of 1 would indicate that the entire bandwidth of the I/O system is being used by the processors. In such a situation, contention would be high and t_{mem} would increase.

The approach taken is to select a value of l_{io} so that contention based delays are negligible. Once this is done, then the I/O system bandwidth becomes a design constraint. Enforcing this constraint effectively makes the queueing delay components of t_{mem} negligible and simplifies the design of an optimal system. The proper selection of l_{io} is considered in more detail in Appendix A. The analysis presented there, for example, indicates that for a typical system with a load of $l_{io} = 0.5$ and a DRAM access time of $40ns$, the probability that a memory request is blocked is about 7% and thus, for this analysis, is ignored.

Given a selected load of l_{io} for the processor generated memory requests, the constraint on the bandwidth thus becomes:

$$BW_{mem} = \frac{IPS \cdot M \cdot linesize}{l_{io}} \tag{9}$$

Now a single I/O channel having a clock rate of clk_{io} and a width of w_{io} will have a bandwidth of $clk_{io} \cdot w_{io}$. Given an overall I/O requirement of BW_{mem} for a low contention I/O design, the number of I/O channels m is simply:

$$m = \left\lceil \frac{BW_{mem}}{clk_{io} \cdot w_{io}} \right\rceil \tag{10}$$

This value of m is used below in determining the chip area requirements for the I/O channels.

3.4 Multiple Applications

So far we have considered only a single program to be executed on the processors. A more realistic assumption is that there is a whole set of programs that make up the workload on the processors. The above analysis can easily be extended to accomodate such a workload notion.

Let the network processing workload W consist of l applications $a_1, a_2, ..., a_l$. Each application i is executed a fraction q_i of the total data stream ($\sum q_i = 1$). The actual number of instructions that are executed by an application a_i depends on its ratio q_i as well as on its complexity, $compl_i$. Complexity in this context is a measure of the number of instructions in the application that have to be executed on average for each byte in a packet of data. Let r_i be the fraction of instructions that are executed on average belonging to application a_i.

$$r_i = \frac{q_i \cdot compl_i}{\sum_{k=1}^{l} q_k \cdot compl_k}, \; i = 1, ..., l \tag{11}$$

The fraction r_i determines the contribution of each application to memory accesses and associated processor stalls. Depending on the load and store frequencies $f_{load,i}$ and $f_{store,i}$ of each application a_i, the respective cache miss rates $m_{ic,i}$, $m_{dc,i}$, and the dirty bit set probability $d_{cd,i}$ can be determined. The number of memory accesses per instruction M_W for workload W is:

$$M_W = \sum_{i=1}^{l} r_i \cdot M_i = \sum_{i=1}^{l} r_i \cdot (m_{ic,i} + (f_{load,i} + f_{store,i}) \cdot m_{dc,i} \cdot (1 + d_{cd,i}) \tag{12}$$

M_W can now be substituted for M in the expressions for CPI, IPS and BW_{mem} and thus represent the execution of a selected workload.

3.5 Optimization

To find an optimal design, first assume that instruction and data cache sizes can be selected over a wide range of values. The number of processors that can combined with the given cache sizes will, of course, depend on the chip size as well as the total number of I/O channels needed for contention-free operation.

Substituting the expression for the number of I/O channels necessary, m, from Equation 10 into the area constraint Equation 2, and assuming we attempt to fill the chip area, we obtain:

$$n \cdot (s(p) + s(c_i) + s(c_d)) + \left\lceil \frac{BW_{mem}}{clk_{io} \cdot w_{io}} \right\rceil \cdot s(io) = s(ASIC). \tag{13}$$

This expression reflects both the chip size and I/O channel constraints. The BM_{mem} term can be further expanded using the above equations going back to Equation 3 or 12 for M, the average number of memory accesses per instruction. M in turn is a function of the fault rates for different cache configurations. This function can be obtained from benchmark execution data which is discussed in Section 4. From this data, one can thus obtain fault rates as a function of cache sizes and other characteristics. Thus, given the chip size, $s(ASIC)$, the size of the processor, cache and I/O components, Equation 13 defines the space of parameter choices for the design. If the number of memory sizes, that are considered, is limited (e.g., only sizes that are powers of two), then the optimization can be performed by an exhaustive search procedure. For each choice there will be corresponding fault rates, number of processors and resulting overall IPS. The choice(s) which yields the largest IPS is the design yielding the highest throughput.

i	Application a_i	Description	Type
1	RTR	Radix-tree routing	HPA
2	FRAG	Packet fragmentation and header checksum	HPA
3	DRR	Deficit Round Robin fair scheduling	HPA
4	TCP	TCP traffic monitoring	HPA
5	CAST	CAST data encryption	PPA
6	ZIP	ZIP data compression	PPA
7	REED	Reed-Solomon forward error correction	PPA
8	JPEG	JPEG image transcoding	PPA

Table 2: Description of Benchmark Applications.

4 CommBench and Workload Definition

To properly evaluate and design network processors it is necessary to specify a workload that is typical of that environment. This has been done in the development of the benchmark CommBench [13]. Applications for CommBench were selected to include a balance between header-processing applications (HPA) and payload-processing applications (PPA). HPA processes only packet headers which generally makes them computationally less demanding than PPA that process all of the data in a packet. The applications included in CommBench are shown in Table 2.

4.1 Application Properties

For each application, we need to know the following properties that can be measured experimentally: computational complexity, load and store instruction ratio, instruction cache and data cache miss rate, and dirty bit probability (for all cache sizes over the optimization space).

The complexity of an application can be obtained by measuring the number of instructions that are required to process a packet of a certain length (for header-processing applications, we assumed 64 byte packets):

$$compl = \frac{instructions\ executed}{packet\ size} \tag{14}$$

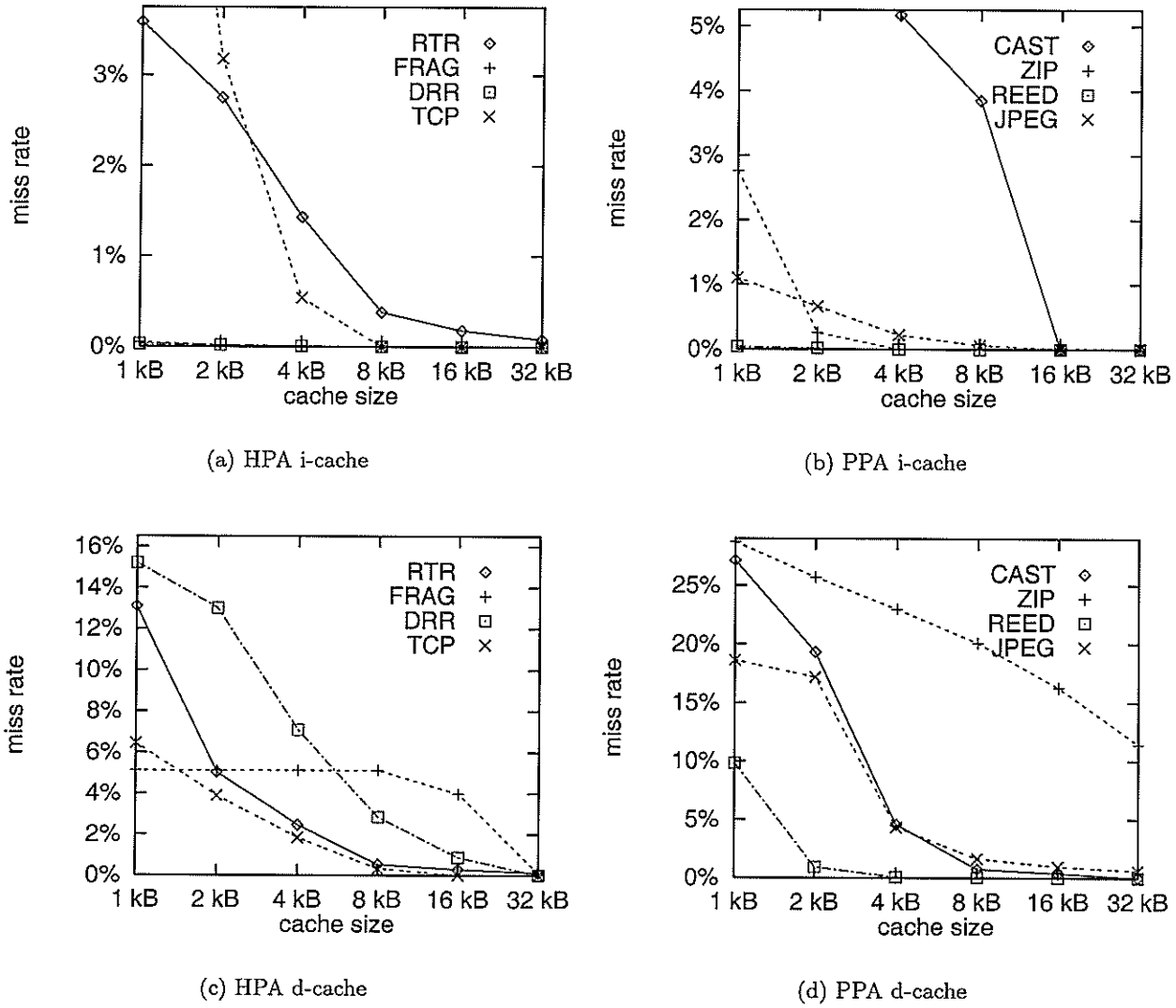
We measured the complexity of the benchmark application with Spixtools [3] on an UltraSparc II. To amortize the system-specific program initialization overhead, the measurements were taken for a large number of packets in a single program run. The complexity numbers, as well as the load and store frequencies, for the different applications are shown in Table 3.

Note that the complexity of payload processing is significantly higher than for header processing. This is due to the fact that payload processing actually touches every byte of the packet payload and executes complex transcoding algorithms. Header processing on the other hand, typically only reads few header fields and does simple lookup and comparison operations.

The cache properties were measured with Shade [4] and Dinero [5]. A 2-way associative write-back cache with a line size of 32 bytes was simulated. The miss rates for the various applications are shown in Figure 2 (for illustration purposes only cache sizes $1kB$ through $32kB$ are shown, but $1kB$ through $1024kB$ were measured). The cache miss rates were obtained such that cold cache misses were amortized over a long program run. Thus, they represent the steady-state miss rates of these applications. As can be seen from the figures, for most applications, miss rates under one percent can be achieved with an instruction cache size of $8kB$ and a data cache size of 16 to $32kB$. Effects of cold caches are considered in Section 6.3.

i	Application a_i	$compl_i$	$f_{load,i}$	$f_{store,i}$
1	RTR	6.85	0.392	0.024
2	FRAG	10.23	0.124	0.067
3	DRR	5.48	0.393	0.083
4	TCP	13.83	0.169	0.077
5	CAST	104	0.203	0.077
6	ZIP	226	0.201	0.059
7	REED	584	0.151	0.051
8	JPEG	81	0.166	0.099

Table 3: Computational Complexity and Load and Store Frequencies of Benchmark Applications.



(a) HPA i-cache

(b) PPA i-cache

(c) HPA d-cache

(d) PPA d-cache

Figure 2: Cache Miss Rates.

component	unit	size
processor	1	$2mm^2$
cache	$1kB$	$.175mm^2$
I/O channel	$1.6GB/s$	$30mm^2$

Table 4: Component Sizes.

The differences between header-processing application and payload applications with respect to computational complexity, cache behavior, and instruction mix (not shown) suggest the use of specialized processors and non-uniform cache configurations for the different application categories. This is considered in Sections 6.1 and 6.2, but for the remaining analysis identical configurations are used for all applications.

4.2 Workload

In this analysis we consider three workloads that are weighted combinations of the applications listed above. The workloads are defined by the vectors $q = (q_1, ..., q_8)$ as follows:

- Workload I: Only header-processing applications, $q_I = (0.25, 0.25, 0.25, 0.25, 0, 0, 0, 0)$.
- Workload II: All applications such that they each execute the same number of instructions, $q_{II} = (0.277, 0.186, 0.347, 0.137, 0.018, 0.008, 0.003, 0.023)$.
- Workload III: Only payload-processing applications, $q_{III} = (0, 0, 0, 0, 0.25, 0.25, 0.25, 0.25)$.

Using individual application complexities (Table 3), the instruction ratios $r = (r_1, ..., r_8)$ for the different workloads can be obtained.

- Workload I: $r_I = (0.188, 0.281, 0.151, 0.380, 0, 0, 0, 0)$.
- Workload II: $r_{II} = (0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125)$.
- Workload III: $r_{III} = (0, 0, 0, 0, 0.105, 0.227, 0.587, 0.081)$.

5 Example System

Given the analysis of Section 3 and the workload and application properties of Section 4, the optimal configurations of a network processor can now be determined.

5.1 Area Constraints

The three main network processor components are the processors, their caches, and the I/O channels. By examining the specifications of several RISC processor cores, estimates on the sizes of these components were obtained and, for $.25\mu m$ technology, are given in Table 4. It is assumed that the sizes scale linearly with the number of each component (e.g., $s(n \cdot c) = n \cdot s(c)$)[2].

Based on these estimates, and using Equation 2, all 'legal' configurations for an ASIC can be enumerated. Table 5, for example, shows the possible configurations for a $100mm^2$ ASIC[3]. The amount of cache is shown for a given number of I/O channels and processors. This value can be split as needed into instruction and data cache. For example, with four processors and two I/O channels,

[2]This is not quite true, however a more complex model can easily be substituted here.

[3]It is assumed that this is the available area after the area needed for routing component connections, power, ground and other miscellaneous logic has been allocated. Also it is assumed that this area does not change significantly for the different component configurations.

I/O channels	processors												
	1	2	3	4	5	6	...	19	20	21	...	34	35
1	68	33	21.33	15.5	12	9.67	...	1.68	1.5	1.33	...	0.06	0
2	38	18	11.33	8	6	4.67	...	0.11	0				
3	8	3	1.33	0.5	0								

Table 5: Configurations for $100mm^2$ ASIC. The Table shows the amount of per-processor cache in kB for a given number of I/O channels and processors.

Work-load	$100mm^2$					$200mm^2$					$400mm^2$				
	c_i	c_d	n	m	IPS	c_i	c_d	n	m	IPS	c_i	c_d	n	m	IPS
I	8	16	13	1	4655	8	16	26	2	9310	8	16	52	3	18621
II	16	16	9	1	2785	16	16	18	2	5570	16	16	37	4	11449
III	16	8	9	2	2392	16	16	17	3	4850	16	16	37	6	9835

Table 6: Optimal SOC Configurations for Various ASIC Sizes (cache sizes in kB, IPS in million instruction per second ($MIPS$)).

total cache size is $8kB$. One possible division for this case would be to have $4kB$ for both instruction and data cache. Note that this table just indicates possible configurations prior to imposing the I/O bandwidth constraint.

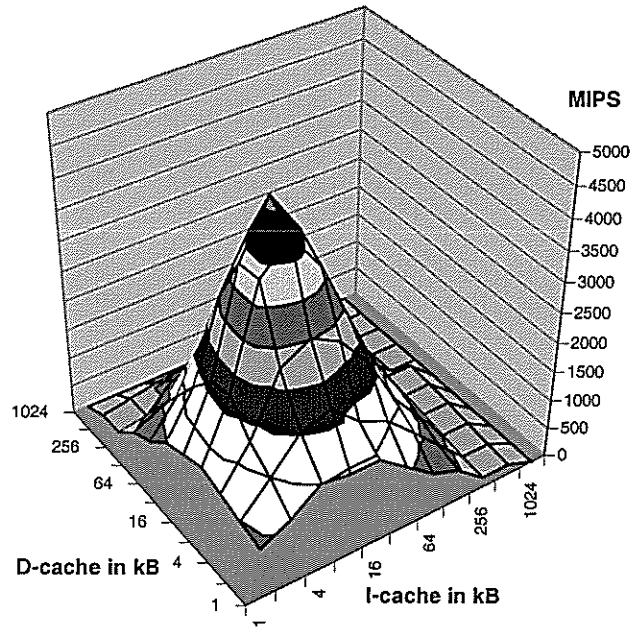
Next, using CommBench data and workload selection, one can determine the resulting cache miss rates and the effective number of instructions that can be executed on the system. This, as explained before, determines the throughput of the system and thus the total performance. The cache miss rates also determine the required I/O bandwidth, which has to be related to the number of I/O channels shown in Table 5.

5.2 Evaluation

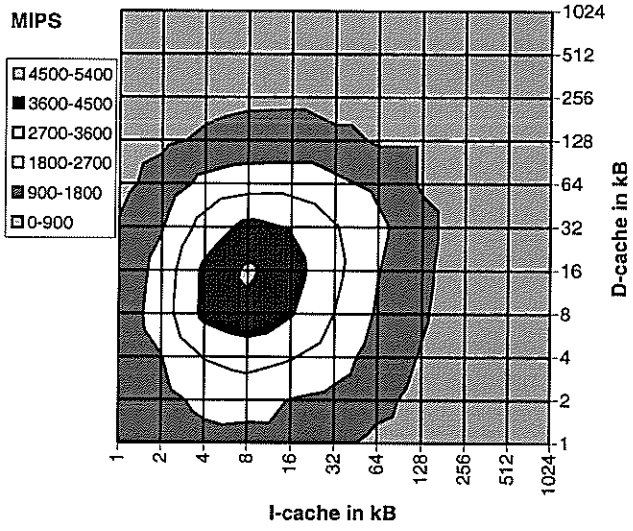
Optimization through exhaustive search can now be done over two independent parameters, the instruction cache size c_i and the data cache size c_d for cache sizes of $1kB, 2kB, 4kB, 8kB, ..., 1024kB$. For each combination of cache sizes, we determine the resulting miss rates and bandwidth requirements. The processors are assumed to be clocked at $clk_p = 400MHz$ and the off-chip memory access time is $t_{mem} = 35$ clocks. Considering a load of $l_{io} = 0.5$ on the I/O channels (see Appendix A), we can determine the number of I/O channels required for a given number of processors. Thus, we search for the maximum number of processors that can fit onto the chip, while still providing the necessary number of I/O channels.

The optimization space is shown in Figure 3. The 3-D figure shows the total MIPS rating of a $100mm^2$ ASIC optimized for Workload I. The other figures show the top-down view of the surface, also for a $100mm^2$ ASIC, optimized for Workloads I through III. It can be seen that there is an optimum in the area around 8 or $16kB$. For smaller or larger cache sizes, the total performance drops significantly. Optimization results for various chip sizes are shown in Table 6. The number of processors scales with the ASIC size, and the optimal cache sizes vary only slightly due to rounding effects.

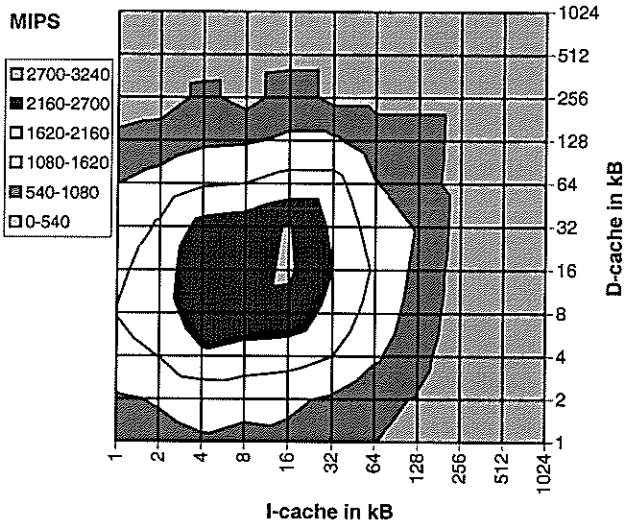
It is notable that the best performance is achieved in only a small configuration space (caches of $8kB$ to $16kB$). This shows that workload-specific optimization is very important. Comparing application properties of CommBench applications to those of workstation applications, as found in the SPEC benchmark [11], one can see that program kernels of networking applications are about one order of magnitude smaller than those of SPEC applications [13]. This means that SPEC programs require much larger instruction caches to achieve the same miss rates.



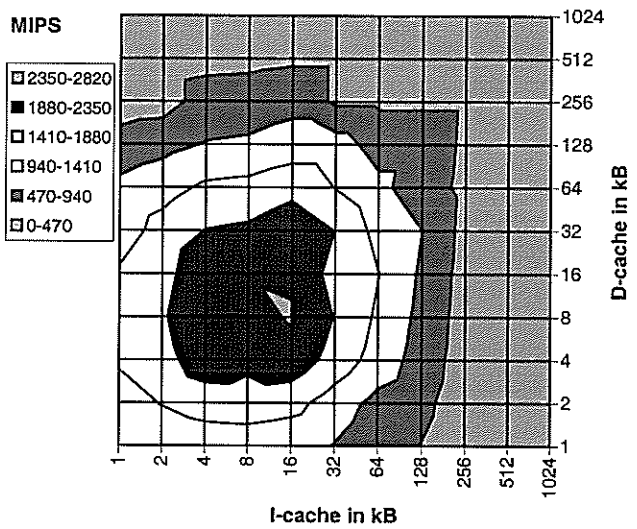
(a) Workload I



(b) Workload I



(c) Workload II



(d) Workload III

Figure 3: Optimization Space for $100mm^2$ ASIC.

In comparison to commercial network processors, the results match roughly with the cache configurations on Intel's IXP1200 [7] with $16kB$ instruction and $8kB$ data cache and Tsqware's TS704 [12] with $16kB$ instruction and $16kB$ data cache. C-Port's C-5 [2] uses smaller caches, where 4 processing engines share $16kB$ of cache. For other commercial products, no information about cache configurations could be obtained.

6 Extensions

For further improvements to the system, we elaborate in this section on three extensions that can be considered: specialized processors, applications-dependent cache configurations, and smart scheduling.

6.1 Processor Pools

One important result from the benchmark measurements is that header processing applications have very different characteristics from payload processing applications. It is a natural extension to use specialized processors for the processing of certain categories of applications.

Header processing applications might be able to make use of sophisticated branch predictors due to their large fraction of load, compare, and branch instructions. Payload processing applications, which are dominated by arithmetic, logic, and shift instructions, could make use of instruction-level parallelism.

Ideally, a system should be configured such that there is the right number of processors for each workload category. This is particularly important if applications can only be executed on processors which are specialized for them because of binary incompatibilities. Thus, the specialization requires knowledge of the expected workload in the system. If the workload is not known in advance, the system can be overengineered to handle 100% header processing traffic as well as 100% payload processing traffic. This will result in a lower overall cost-performance.

6.2 Application-Specific Cache Configurations

Analogous to specializing processors, it is possible to specialize cache configurations for individual applications or groups of applications. Table 6 ($100mm^2$ chip size) shows that the optimal cache configuration for header processing applications (workload I) is $c_i = 8kB$ and $c_d = 16kB$. The optimal configuration for payload processing (workload III), though, is $c_i = 16kB$ and $c_d = 8kB$. As a result of a global optimization, that does not take individual applications into account, workload II, which contains both types of applications, has its optimum at $c_i = 16kB$ and $c_d = 16kB$. This optimum requires $32kB$ of cache per processor because payload processing applications need $c_i = 16kB$ for good performance and header processing applications need $c_i = 16kB$ for good performance. If the optimization can be performed for application specific cache configurations, the optimal configuration is a set of processors with $c_i = 8kB$ and $c_d = 16kB$ for payload processing and a set of processors with $c_i = 8kB$ and $c_d = 16kB$ for header processing. In this case the amount of per processor cache is only $24kB$ versus $32kB$ in the uniform design. The chip area saved by the smaller cache configuration can then be used for additional processors. Naturally, a packet scheduler would need to route packets to the processor with the best cache configuration.

6.3 Scheduling

So far, all packets were distributed randomly over the processors in the system. The packet at the head of the queue in the packet demultiplexer was sent to the first processor that became available. While this approach is easy to implement, it has the disadvantage that it does not take into account what program was executed on that processor prior to the current packet. In most cases these programs will be different, and this will cause an increase in "cold cache."

There are two approaches to improving packet demultiplexer scheduling to reduce cold cache misses:

- send packets of the same type of processing to the same processor
- buffer packets of one type and execute them back-to-back on the same processor

The first approach to using warm caches is to have the scheduler keep track of all programs that are executed on the processors and their expected finishing time (this can be estimated using the computational complexity measure). If the packet at the head of the queue requires processing of type i, the scheduler can check if a processor, that is currently executing program i, is expected to finish within some time Δt. If so, the packet will be queued for this processor rather than sent to another processor that might become available earlier, but has executed a different program. The amount of waiting, Δt, is the time that can be saved by executing the program on a warm cache instead of a cold cache. If the packet has to wait longer than Δt, it is not worth waiting for a processor with warm caches, because it will be done quicker on a processor with cold caches for which it does not have to wait.

Another approach is to buffer packets of the same type and then send them back-to-back to the same processor. This guarantees that except for the first packet the caches are warm. There is a tradeoff between buffering and introducing delay. If too much is buffered before the packets are processed, the first packets can be delayed significantly, but this scheme is much simpler to implement, since it does not require an estimation of finishing times.

Both scheduling approaches require multiple queues for the different processing types. This is also necessary when specialized processors are used, because packets have to be queued until a processor suitable for their specific processing requirements becomes available.

7 Summary and Conclusions

In this paper, we consider a multiprocessor system-on-a-chip that is specialized for the telecommunications environment. Network traffic can be processed by special application software that executes on a set of processors contained on a single chip. The problem analyzed is that of determining the optimal number of processors, associated cache sizes and I/O channels that should be present in such a design given a set of defining parameters and constraints with the principal constraint being the total chip area available.

An analytical model of the system has been presented that reflects the overall computational power based on the computational complexity and cache behaviour of a set of measured programs. The analytic expressions developed are such that the design can be optimized to maximize network processor throughput with the optimal system specified in terms of number of processors, cache sizes, and I/O channels for a given workload characterization. Workload statistics were obtained using CommBench, a telecommunications workload that contains both header and payload processing applications.

The results indicate that, for example, with a $200mm^2$ chip and a workload whose computational load is equally split between header and payload processing, the optimal configuration would have 16 processors, both instruction and data caches of $16kB$, and 2 I/O channels. Using the expressions provided the optimal designs over range of parameter choices is possible.

While this model and the resulting designs are promising, there are some clear extensions to this work which would permit greater performance with the same given area constraints. One example concerns the use of non-uniform cache sizes. In such a situation, scheduler assignment of tasks would be based on their type (e.g., header or payload processing) and resulting cache requirements. Another example would permit non-uniform processors where processor instruction sets are specialized to different applications class requirements and the scheduler routes tasks accordingly.

Further extensions of the current model are also being considered. In particular the effect of cold cache misses is being analyzed and efforts are being undertaken to analyze the effect of having processor scheduler distribute tasks on the basis of minimizing such misses.

References

[1] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains.* John Wiley & Sons, Inc., New York, 1998.

[2] C-Port Corporation. *C-5^{TM} Digital Communications Processor*, 1999. http://www.cport-corp.com/solutions/docs/c5brief.pdf.

[3] R. F. Cmelik. SpixTools instroduction and user's manual. Technical Report TR-93-6, Sun Microsystems Laboratories, Palo Alto, CA, 1993.

[4] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. of ACM SIGMETRICS*, Nashville, TN, May 1994.

[5] J. Edler and M. D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator.* http://www.neci.nj.nec.com/homepages/edler/d4/, 1998.

[6] IBM Corp. *Rainier Network Processor*, 2000. http://www.ibm.com/.

[7] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://developer.intel.com/design/network/ixp1200.htm.

[8] Lucent Technologies Inc. *PayloadPlus^{TM} Fast Pattern Processor*, Apr. 2000. http://www.agere.com/support/non-nda/docs/FPPProductBrief.pdf.

[9] MMC Networks, Inc. *nP3400*, 2000. http://www.mmcnet.com/.

[10] Rambus Inc. *Rambus(R) Technology Overview*, Feb. 1999. http://www.rambus.com/docs/tech-over.pdf.

[11] Standard Performance Evaluation Corporation. *SPEC CPU95 - Version 1.10*, Aug. 1995.

[12] T.sqware Inc. *TS704 Edge Processor Product Brief*, 1999. http://www.tsqware.com/.

[13] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, Apr. 2000.

[14] T. Wolf and J. Turner. Design issues for high performance active routers. In *Proc. of the International Zurich Seminar on Broadband Communications*, pages 199–205, Zurich, Switzerland, Feb. 2000.

Appendix A: Off-Chip Memory Access

This appendix presents a brief analysis of the memory access time, t_{mem}. We assume a split-transaction, pipelined design with separate address and data busses. Requests for memory are signaled over the address bus, then the memory bank retrieves the requested data, and sends it back over the data bus. Before any bus transaction, a component has to gain control over either the address or data bus, which is immediately released after transmission.

Consider the example of an off-chip memory read (the write analysis is similar and is not considered here). The transaction begins when the processor requests access to the bus. This takes a time designated as $t_{control\,access}$ and represents the queueing delays associated with multiple processors accessing the bus. Once the bus has been acquired, the time to send the memory address is designated as $t_{signaling}$. Since this is a split-transaction bus, the bus is now released, however the memory proceeds to access the required information and this takes $t_{mem\,access}$ time. Since several memory requests may be present in the memory itself, there is the potential for a queueing delay within the memory. When memory access is completed, the memory must now reacquire the bus to complete the transaction. It experiences queueing delays again and this time designated as $t_{transfer\,access}$. Once the bus is acquired, the data transfer proceeds and takes $t_{transfer\,time}$. The total latency associated with a memory access is thus:

$$t_{mem} = t_{control\,access} + t_{signaling} + t_{mem\,access} + t_{transfer\,access} + t_{transfer}. \quad (15)$$

To use the I/O channel efficiently, both pipelining and memory interleaving techniques are commonly incorporated in the bus design. For analysis purposes we assume an h way interleaved memory and constrain the design so that only h memory requests are permitted at any given time. Since the processors associated with these h possible requests are all dealing with different non-interacting flows, we assume that different memory banks are assigned to each flow and thus remove any possible queueing delays associated with contention between the h flows within the off-chip memory system. $t_{mem\,access}$ now is a single value obtained from the memory chip manufacturer.

Consider now the time associated with the memory reacquiring the bus after a memory request has been satisfied, $t_{transfer\,access}$. From the perspective of the memory, the bus can be viewed approximately as an M/G/1 system. With such a system and an average utilization or load, l_{io}, the probability of the memory having h requests outstanding for use of the bus is [1]:

$$Prob[\#\,queued\,requests = h] = (1 - l_{io}) \cdot l_{io}^{h}. \quad (16)$$

The probability of having more than h requests outstanding is:

$$Prob[\#\,memory\,request > h] = 1 - (1 - l_{io}) \cdot \sum_{i=0}^{h} l_{io}^{i}. \quad (17)$$

The average time to access the transfer component of the I/O channel, $t_{transfer\,access}$, then depends on the average number of requests that will be served before it can gain access.

$$E[t_{transfer\,access}] = t_{transfer} \cdot (1 - l_{io}) \cdot \sum_{i=1}^{h-1} l_{io}^{i} \cdot i. \quad (18)$$

To simplify the analysis we consider the queueing delays associated with $t_{control\,access}$ to be constrained in the same manner as those associated with $t_{transfer\,access}$. Doing this, Equation 18 above applies to $t_{control\,access}$ with $t_{signalling}$ replacing $t_{transfer}$.

Consider now a system with a 16 bit wide, $800MHz$ bus and a DRAM access time of $40ns$. For $t_{signalling}$, we assume 4 bus clocks or $5ns$. $t_{transfer}$ for a 32 byte memory line is 16 bus clocks or $20ns$. With up to $h = 4$ interleaved requests, we choose a load on the I/O channel of $l_{io} = 0.5$ to get a probability of over 93% that there are 4 or less simultaneous memory requests. The average access times then become $t_{transfer\,access} = 18ns$ and $t_{signalling\,access} = 5ns$. Thus, the total memory access time t_{mem} is

$$t_{mem} = 5ns + 5ns + 40ns + 18ns + 20ns = 88ns.$$

In terms of the processor clock, this corresponds to 35 clock cycles on a $400MHz$ processor.