Performance Analysis of the Intel IXP1200 Network Processor

Final Report

http://www.cs.cmu.edu/˜uhengart/15-740

Rajesh Krishna Balan and Urs Hengartner
{rajesh,uhengart}@cs.cmu.edu

Abstract

Programmable network processors such as Intel's IXP1200 promise easy extendibility and high performance. In this work, we present an evaluation of the IXP1200 from a computer architect's point of view. We determine metrics like average number of instructions in flight, IPC, and branch mispredictions. In addition, we investigate memory bus bandwidth and memory access latencies. We show that the IXP1200 is able to achieve high IPC numbers due to its hardware support for multiple threads. This feature also allows it to hide the relatively high latencies for accessing memory.

1 Introduction

Network processing in routers so far has mainly relied on either custom-built network ASICs or on general-purpose processors. The first approach has the benefit of being able to achieve high performance, on the other hand, it is difficult to extend an ASIC to make it support new protocols. The second approach is easily extendable, but it may suffer in terms of performance. Recently, programmable network processors which try to unify the positive aspects of both worlds have started to appear. These processors typically consist of an embedded control processor and several data processing engines. The control processor is responsible for executing the control plane functionality of a router (e.g., maintenance of the routing table), whereas data-plane (per packet) operations (e.g,. packet classification) are performed by the data processing engines. An example of such a network processor is the Intel IXP1200 network processor [2].

The IXP1200 consists of a StrongArm core and six programmable RISC cores (also known as 'microengines'). The core runs the VxWorks operating system and controls the microengine threads. Each microengine can execute up to four threads. Its instruction set is specially designed for packet processing. An instruction is piped through a five-stage pipeline with the execution stage taking one cycle for each instruction. These microengines do not do out-of-order speculation.

In this work, we present an evaluation of the IXP1200 network processor. The evaluation is based on a simulator of the IXP1200. For multiple IP forwarding applications with different kinds of network ports, we determine metrics like average number of instructions in flight, IPC, and branch mispredictions. In addition, we investigate memory bus bandwidth and memory access latencies.

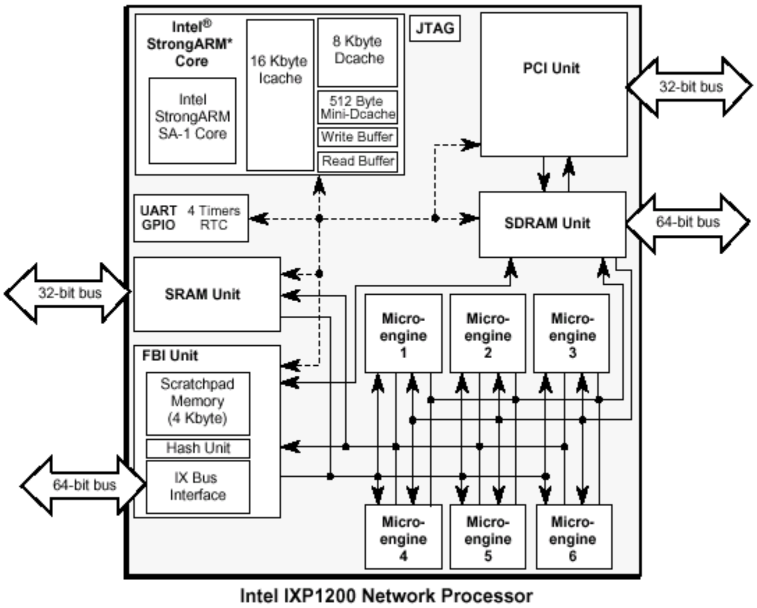


Figure 1: Architecture of the Intel IXP1200 network processor. (*Source:* Intel IXP1200 Network Processor Datasheet, page 11)

In Section 2, we give an overview of the architecture of the IXP1200. Section 3 describes how the IXP1200 forwards packets. Our evaluation of the IXP1200 is the topic of Section 4. Section 6 concludes this paper and discusses future work.

2 Architecture

Figure 1 presents an overview of the architecture of the Intel IXP1200 network processor. The processor consists of an Intel StrongArm Core, six RISC cores ('microengines') with an entirely new instruction set, a vast register file replicated across all the microengines, and shared memory resources in the form of scratchpad RAM located in the FBI unit (the unit responsible for transferring packets to and from MAC-layer devices). The IXP1200 contains SDRAM to store packets entering the device and faster SRAM to store heavily used data structures like the route lookup tables. The SDRAM and SRAM are accessed via separate 64-bit buses. Requests to access the SDRAM or SRAM are queued in the SDRAM and SRAM controllers (known as units) respectively. Each microengine is directly connected to the SRAM controller, SDRAM controller, and the FBI unit. The FBI unit transfers packets to and from MAC-layer devices using either a standard 32-bit PCI bus (accessed via a PCI controller unit) or a special 64-bit IX bus (which can be used to access network cards which are IX bus compatible). In the following sections, we describe some of the components in more detail.

2.1 StrongArm Core Microprocessor

The StrongArm Core is the same 32-bit RISC processor used in the Intel StrongArm SA-1100. We will be not be doing any experiments involving the StrongArm Core for this project.

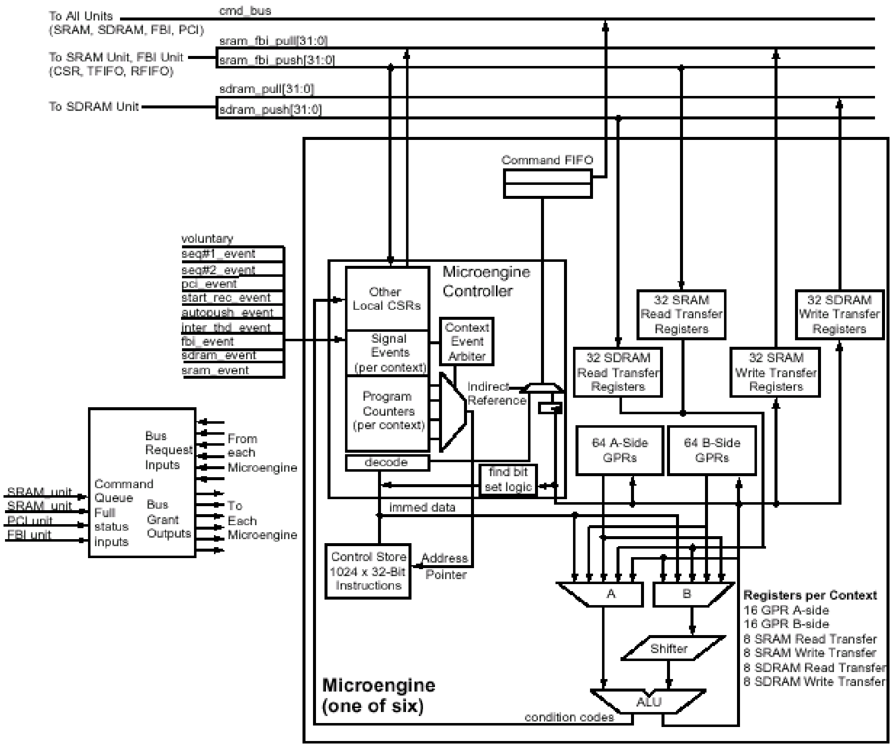


Figure 2: Architecture of a microengine. (*Source:* Intel IXP1200 Network Processor Hardware Reference Manual, page 4-2)

2.2 Microengines

The six 32-bit microengines perform data movement and processing without support from the StrongArm core. Each microengine has four independent program counters, hardware support for very low overhead context switching (the overhead is usually 1 cycle), a programmable 1K instruction memory, 128 32-bit general purpose registers, 128 32-bit transfer registers to service the SRAM and SDRAM units, and an ALU and shifter that is capable of performing an ALU and shift operation in a single cycle. The instruction set was specifically designed for networking applications and and provides instructions for byte manipulation and pattern matching. Figure 2 provides an overview of a microengine.

The 128 general purpose registers can be addressed using relative or absolute addressing. Relative addressing divides the registers among the four threads (32 each), absolute addressing allows a register to be shared. The registers are single-ported only and divided into two banks in order not to impair performance.

The transfer registers are divided into 32 SRAM read, 32 SRAM write, 32 SDRAM read, and 32 SDRAM write registers. The 4 sets of registers are connected to the SRAM or SDRAM (depending on the set) via separate 32-bit data buses. The SRAM transfer registers are also used to move data between the FBI unit and the microengine. A microengine can issue a data transfer between multiple transfer registers and the SRAM or SDRAM with a single command.

The 32-bit ALU and shifter can operate on data supplied by the transfer and general purpose registers and on immediate data. The ALU can perform addition, subtraction, and logical

Stage	Description
P0	Instruction lookup
P1	Instruction decode and formation of source register address
P2	Read of operands from source registers
P3	Perform ALU, shift, or compare operations
P4	Write result to destination register

Table 1: Work per pipeline stage.

operations and generate condition codes based on these operations.

Each microengine contains a set of control and status registers. The StrongArm core uses these registers to program, control, and debug the microengines.

Each microengine instruction is pipelined through a five-stage execution unit. Table 1 lists the work performed in each stage. Data is forwarded from P3 if necessary. Branch decisions are made in the P1, P2, or P3 stage, depending on the type of branch instruction. Three mechanisms are supported to reduce the aborted cycles as a result of a branch decision:

Deferred branches An instruction that follows a branch decision is allowed to execute before the branch takes effect (delay slot optimization required).

Setting condition codes earlier Instructions are rearranged such that the condition codes upon which the branch decision is made are set two or more instructions before the branch.

Branch guessing An optional token in the branch instruction allows instructions to be prefetched from the "branch taken" path before the actual branch decision is made. If the token is omitted, the operation assumes that the branch is not taken.

Hardware multithread support allows four separate threads to share execution time on a microengine. When issuing a memory reference, a thread can put itself to sleep or perform useful work independent of the reference. As such, the microengine can hide long latencies caused by referencing off-chip memory. In addition, an explicit yield instruction allows threads to sleep until a specified event occurs.

2.3 FBI Unit and IX Bus

The FBI unit contains receive and transmit FIFO buffers (RFIFO and TFIFO), 4 Kbyte scratchpad RAM, and a hash unit for generating 48-bit and 64-bit hash keys. The FIFO buffers are used for caching data received from the network devices and for caching data that should be sent out, respectively. The FBI unit also controls the IX bus. The IX bus is similar to the PCI bus but it is simpler and faster. In addition, the FBI unit includes control and status registers (CSR), accessible by both the microengines and the StrongArm core. An overview of the FBI unit is given in Figure 3.

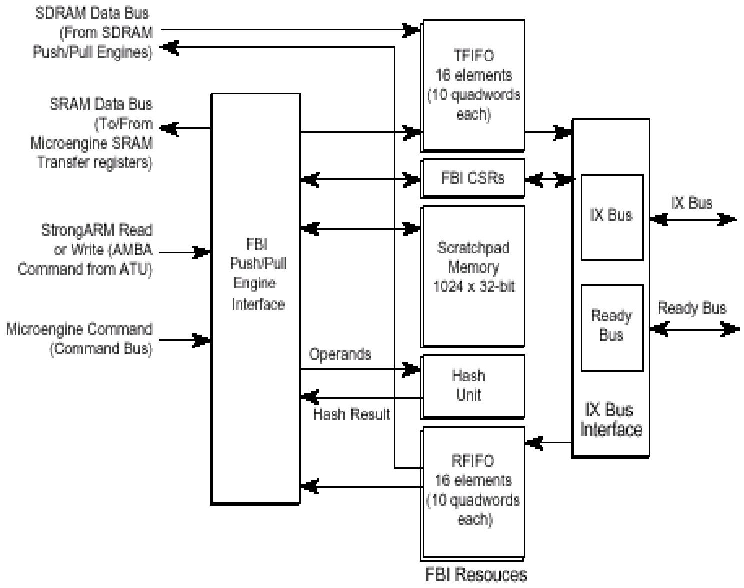


Figure 3: Architecture of the FBI unit. (*Source:* Intel IXP1200 Network Processor Hardware Reference Manual, page 6-2)

3 Packet Forwarding

To become familiar with the IXP1200 and the way its different components interact to forward packets, we now describe one of the example applications shipped with it in detail. This knowledge was gathered by looking at the microcode as documentation for these applications was hard to find.

We chose the 'pfwd12' IP routing application. This application processes incoming packets from 12 100Mbit/s network ports (connected to the IX bus), makes a routing decision based on the IP destination address of the packets and forwards them to one of the outgoing ports. The reason we chose this particular application is that it closely follows the guidelines for receiving/transmitting packets given in the IXP1200 programming manual in that it has separates receive and transmit scheduler threads (look below).

A single thread running on microengine 4 assigns incoming packets to 12 receive threads. Each of these threads is responsible for one of the 100Mbit/s ports. Microengines 0-2 each run four receive threads that process incoming packets, check their IP checksum, select their output port, and queue them in SDRAM. A transmit scheduler running on microengine 5 schedules the transmit threads. Four transmit threads run on microengine 3. These threads dequeue packets and forward them to their output port.

The receive scheduler thread waits until one of the receive threads signals that it is ready to process data (one of the FBI unit's control registers is used for signalling). It then consults one of its SRAM transfer registers to find out which ports have data available. This information is stored in the SRAM transfer register by the FBI unit, which itself is signalled of data availability by the individual network devices using the ready bus parallel to the IX bus. The receive scheduler then tracks each receive thread's processing status and finds out which of them are ready. For each receive thread that is ready, it checks whether its port (receive thread i is assigned to port i) has data available. If so, it issues a receive request to the FBI unit and instructs it to move data into the ith receive FIFO buffer element. As soon as the data is in the buffer, the FBI unit directly notifies the corresponding receive thread.

The receive thread loads the first 16 bytes (containing source and mac layer address, packet length, and network layer protocol) into four of its SRAM transfer registers. In parallel, the address for a packet buffer in SDRAM is retrieved from SRAM (SRAM has a structure which points to the next free location in SDRAM). If the packet is found to be an IP packet, the FBI unit is instructed to move the next 32 bytes directly into SDRAM. In parallel, the first 24 bytes of these 32 bytes (containing IP header) are copied into six SRAM transfer registers. Based on this information, the IP total length, checksum, and time-to-live fields are checked. If the fields are correct, the time-to-live field is decremented by one.

Next, the IP destination is extracted and a route lookup is performed. The data structure used for the lookup is stored in SRAM. The lookup returns an index, which is used to retrieve the MAC destination address of the next hop from a table stored in SDRAM. The MAC address is prefixed to the packet already stored in SDRAM. At the same time, the modified IP header is copied to SDRAM.

Then, the packet is enqueued. For each output port, there is a queue in SRAM. The enqueuing operation adds another entry containing information about the packet's location in SDRAM to the corresponding queue. Also, the rest of the packet is now copied to SDRAM; if necessary, additional receive requests are sent to the FBI unit.

The transmit scheduler first determines which of the ports are ready to transmit data (similar to the way the receive thread checks which ports have data to send). It then picks a free element in the transmit FIFO buffer and generates a task assignment message. This message contains the queue number (which corresponds to the available port) and the available element. Task assignment messages are stored in the scratchpad RAM and are processed by the transmit threads on a FIFO basis.

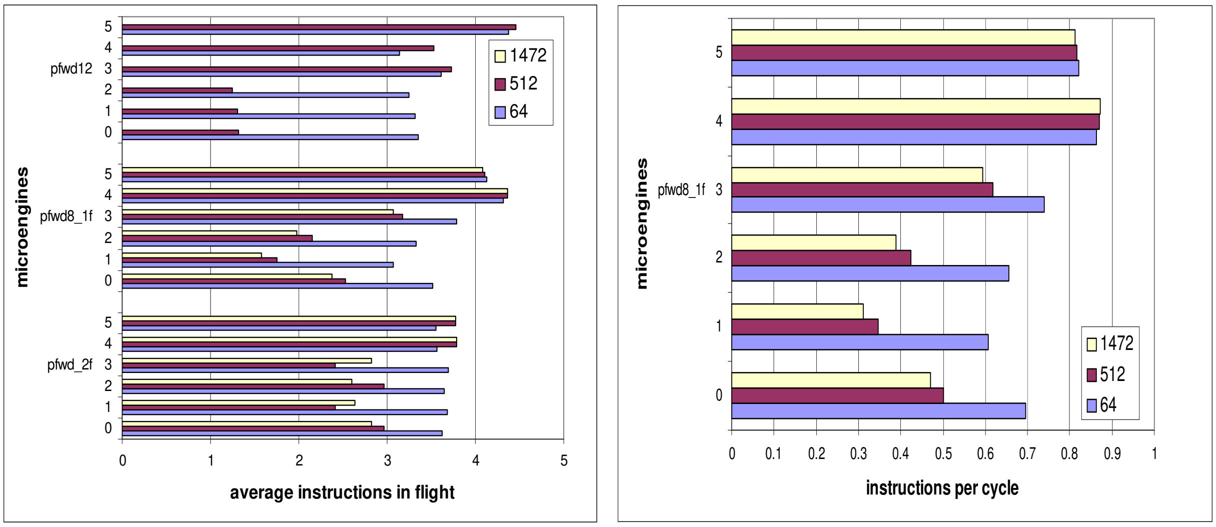
A transmit thread reads the next task assignment message, gets the packet descriptor from SRAM, and has the data transferred from SDRAM to the transmit FIFO buffer. Finally, the FBI unit is instructed to move the packet to the network device.

4 Evaluation

4.1 Methodology

Because these kind of network processors have begun to show up only recently, there is still a lack of commonly accepted benchmarks or detailed evaluations of their performance. As such, we had to resort to using evaluations of conventional processors (such as Cvetanovic and Kessler's performance analysis of the Alpha 21264 [1]) as models from which to derive our methodology to evaluate the performance of the IXP1200. Hence, we decided to look at metrics like average number of instructions in flight, instructions per cycle, branch prediction accuracy, and latencies in accessing memory as measures to gauge the performance of the IXP1200.

We evaluate the performance using the application described in Section 3, pfwd12, which has 12 100Mbit/s ports. In addition, we also used 2 other applications; pfwd8_1f, which has 8 100Mbit/s



(a) Average number of instructions in flight.

(b) Average IPC.

Figure 4: Instructions in flight/IPC.

and 1 1Gbit/s port, and pfwd_2f, which has 2 1Gbit/s ports. pwfd8_1f's microengine threads are structured in a similar way as pfwd12's, whereas pwfd_2f does not have separate receive and transmit schedulers (16 receive threads and 8 transmit threads directly receive and send data).

We chose to run our evaluation on the simulator shipped with the IXP1200 development environment because the actual hardware does not provide any fine-grained performance information. The simulator guarantees that there are always packets available at each input port. We ran experiments with 3 different packet sizes (64, 512, and 1472 bytes respectively). During each experimental run, the simulator had to forward 1000 packets. We chose this rather small number of packets because of the long running time of the simulator (about 1 hour per run when running the simulator in Windows NT (via vmware running in linux) on a Pentium III 500 MHz) and the large size of the output file (about 1 Gbyte per run). However, since the results for 1000 packets are nearly identical to the results for only 100 packets, we are confident that the results for more than 1000 packets would look similar to the results presented here.

The simulator can be run from either within the development environment or as a commandline application. The simulator can be configured to output detailed information about each stage of the execution pipeline for every microengine. Our evaluation results are gathered both from statistics directly provided by the simulator and from scripts written to process the simulator's output.

4.2 Instruction Execution

Figure 4 (a) shows the average number of instructions in flight in each of the six microengines for the three example applications and for three different packet sizes[1]. Since the microengines do

[1]Note that there are no results for the 1472 bytes case of the pfwd12 application, where our evaluation repeatedly failed due to unclear reasons.

not do any sort of out-of-order processing or speculation, the average number of instructions in flight directly leads to the average number of executed instructions per cycle. This observation is confirmed in Figure 4 (b), where we show the average IPC for the pfwd8_1f application. The lack of out-of-order processing and speculation also implies that the average number of instructions in flight is limited by the number of stages in each microengine (i.e., 5) and that the IPC is limited by the number of pipelines in each microengine (i.e., 1).

As can be seen in Figure 4 (a), the usage patterns vary for the three applications. For pfwdf12 and pfwd8_1f, microengines 0-2, which run the receive threads, the pattern looks different than the pattern for microengines 3-5, running the transmit threads, receive thread and transmit threads, respectively. The fact that pfwd_2f is differently structured than the other two applications is clearly visible from its different usage pattern, here, microengines 0-3 run the receive threads and 4-5 the transmit threads

For pwfd12 and pfwd_2f, the receive work is evenly distributed among the microengines 0-3, whereas for pfwd8_1f, there is some variation among the them. This observation may be explained by an uneven distribution of the processing of the packets from the 1 Gbit/s port.

For pfwd12 and pfwd8_1f, it can also be seen that the microengines which do the scheduling tasks (microengines 4 and 5) do the most amount of useful work as their average instructions in flight values are the highest. In addition, the value for the microengine doing the transmit task (microengine 3) is also quite high, thus there is some danger of this microengine becoming the bottleneck.

Finally, another finding is that the amount of useful work done by the microengines decreases as the size of the data packets increases. This is because when the packet size increases, more time needs to be spent transferring the packet to and from the various registers and memory locations in the IXP1200, whereas the overhead for header processing and route lookup stays constant.

4.3 Branch Prediction

As explained in Section 2.2, there is a way for the programmer to specify when he believes that a branch is taken most of the time. In the next evaluation, we look at this kind of branches and examine how often the prediction was correct and the branch was taken. We do not look at branches where the specification was omitted because omission does not necessarily imply that the programmer assumed that the branch would not be taken. It could also mean that the programmer did not know about the fate of the branch.

Figure 5 shows the percentage of correct and wrong predictions for the pfwd8 and the pfwd12 application. The presented results are only for microengine 4 (receive scheduler). The code running on microengines 3 and 5 is missing such branch instructions and the branches in microengines 0-2 were predicted correctly all the time. The latter finding is of little surprise, closer inspection of the code reveals that the branch would fail only for problems like non-correct IP packets or missing route entries. None of these problems can occur in our simulation. As seen in Figure 5, for pfwd12, the percentage of wrong predictions is bigger than the percentage of correct predictions. When looking at the source code, we find that these wrong guesses are caused by wrongly assuming that there is no concurrent request outstanding for the FBI unit. The second application, pfwd8, correctly predicts more than 50% of its branches. pfwd8's receive scheduler is slightly different from pfwd12's and does not include the particular code causing the mispredictions in pfwd12. Therefore, omitting

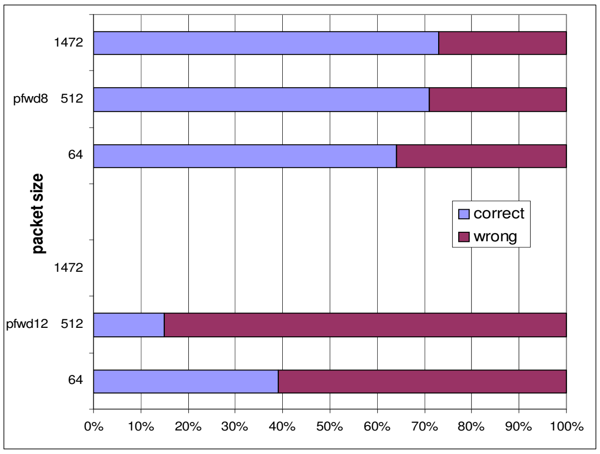


Figure 5: Branch predictions.

the assumption that the branch is taken in pfwd12's source code should help its performance.

4.4 SDRAM and SRAM Bus

In this section, we take a look at the SRAM and SDRAM data bus. The SDRAM bus connects the microengine SDRAM transfer registers to the SDRAM memory. The 64-bit bus is divided into a 32-bit push and a 32-bit pull bus. In all our experiments, both the push and the pull bus were idle most of the time ($\geq$ 68%). In the remaining time, data was transferred from/to the receive/transmit FIFO buffers. Data traffic between SDRAM and the microengines was marginal ($\leq$ 2% for each engine).

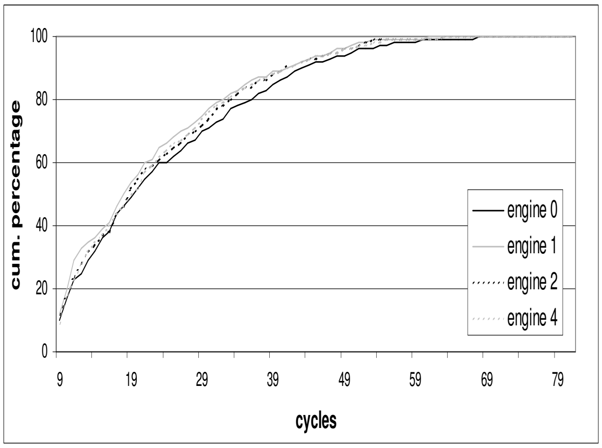
The SRAM bus is shared between the SRAM memory and the FBI unit equally. It connects the SRAM transfer registers to the SRAM memory and to the FBI unit. This 64-bit bus is also divided into a 32-bit push and a 32-bit pull bus. Both buses are idle most of the time ($\geq$ 64%). Transferring data between the FBI unit or the SRAM and the microengines consumed only a small amount of time ($\leq$ 5% per engine).

In summary, neither the SRAM nor the SDRAM bus seem to be a bottleneck for the IXP1200.

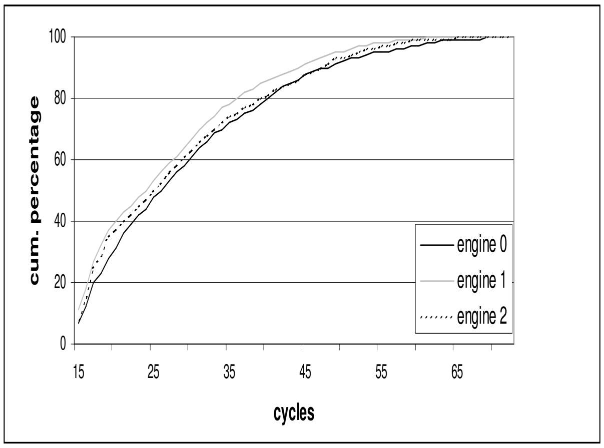
4.5 Latency

Accessing external resources such as SDRAM memory very often results in serious performance penalties for a processor because of the large latency associated with such an access. In this section, we explore the IXP1200's latency for accessing the FBI CSR (control and status registers), the receive FIFO buffer, SRAM and SDRAM memory, and the scratchpad RAM in the FBI unit.

Figure 6 (a) shows the cumulative distribution graph for the latencies in accessing the FBI unit while Figure 6 (b) shows the latencies in accessing the receive FIFO buffer. Note that the presented data covers only read data, since the simulator does not provide data for written data. Since we expect memory latency to be independent of the actual application, the data presented here is only from the pfwd12 application. Also note that all the figures in this chapter have different scalings. For the FBI CSR latency, more than 50% of the accesses take at least 25 cycles to complete, whereas



(a) FBI CSR latency.



(b) Receive FIFO buffer latency.

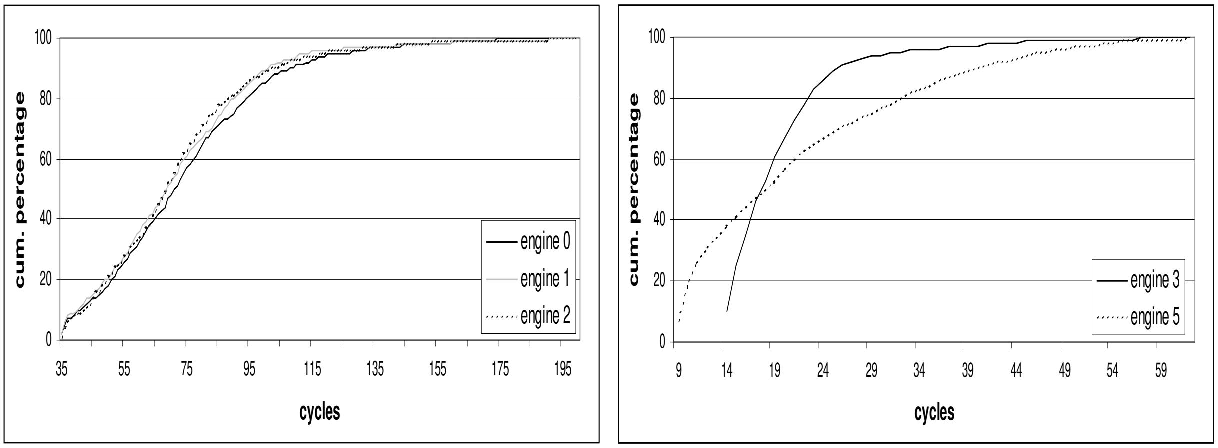
Figure 6: Latencies for FBI CSR and receive FIFO buffer.

for the receive FIFO buffer, it takes at least 30 cycles. The memory bandwidth is about equally shared between the four (three) microengines (the microengines not shown in the figures do not access the corresponding resources). The minimum latency for accessing the FBI CSR and the receive FIFO buffer is 9 and 15 cycles, respectively.

It becomes worse when accessing SDRAM. As shown in Figure 7 (a), 50% of the SDRAM accesses take at least 75 cycles to finish. The minimum latency is 35 cycles, but it can take up to 200 cycles. The latency to access the scratchpad RAM in the FBI unit is fairly small as shown in Figure 7 (b). 50% of the requests finish within 18 cycles. However, as opposed to the previous cases, for the scratchpad RAM case, the two engines do not share the bandwidth equally. The reason for this behavior is not clear.

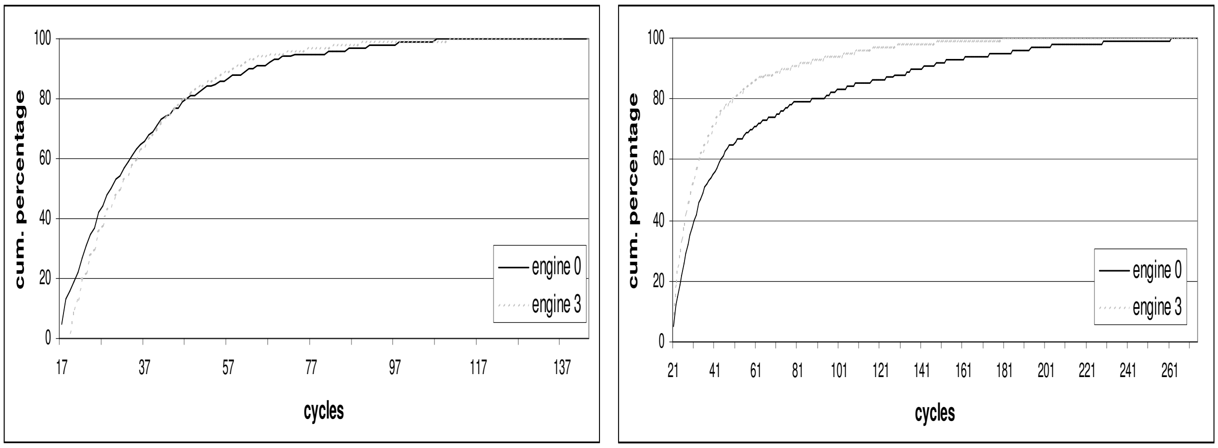
There are two ways to access the SRAM memory: locked and unlocked. Both cases are presented in Figure 8. We show only the data for microengines 0 and 3, since the results for engines 1 and 2 are very close to the ones for engine 0. Engines 4 and 5 do not access SRAM. The SRAM latencies are smaller than the SDRAM latencies. For the unlocked case, 50% of the request take at most 29 cycles to complete, which is much faster than the access times for SDRAM. For the locked case, the corresponding number is 36 cycles, which is still faster than SDRAM. The minimum latency for accessing unlocked and locked SRAM memory is 17 and 21 cycles, respectively. On the other hand, accessing locked SRAM memory can take more than 250 cycles in rare cases. Similar to the scratchpad RAM case, the access bandwidth is not equally distributed among all microengines.

As can be seen from the various graphs, the IXP1200 has some very long latencies in accessing its various memory devices. However, the IXP1200 still manages to have a very high IPC count even though its memory operations have such high latencies. The reason for this is that the IXP1200 has multiple threads running on each microengine. Thus when one thread is stalled waiting for a memory access, another thread can do useful work. With this feature, the IXP1200 is able to maximize the amount of useful work it does.



(a) SDRAM latency. (b) Scratchpad RAM latency.

Figure 7: Latencies for SDRAM and Scratchpad RAM.



(a) SRAM latency (unlocked). (b) SRAM latency (locked).

Figure 8: Latencies for SRAM.

5 Related Work

Programmable network processors are currently of big interest to the networking community and new such processors are being announced on a nearly monthly basis. Apart from Intel, programmable network processors have also been announced by Agere [6] (bought by Lucent), C-Port (bought by IBM), Lexra [4], and Sitera [5] (bought by Vitesse Semiconductor). However, evaluations of these platforms still have to appear. Although the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) has released a networking benchmark suite [3], no vendor has yet published test results.

6 Conclusions and Future Work

Our work has shown that the IXP1200 succeeds in achieving high IPC values although it misses features like out-of-order execution and speculation. The crucial feature that allows it to perform so well is hardware support for multithreading and fast context switches. This support also successfully hides the high (in terms of the number of cycles the instruction has to wait before the memory access is completed) access latencies for SRAM and SDRAM memory.

Because the IXP1200 has appeared only recently, it has not been extensively studied yet. As such, it presents a great opportunity for future research work. One area of future work could be evaluating its performance in a real network environment. The applications available with the IXP1200 simulator do not completely implement a real routing application as they make certain simplifying assumptions in their packet handling by only catering to a subset of Ethernet and IP packets. For example, only Ethernet packets of the 802.3 format are handled, Ethernet packets of the "Ethernet-2" format are neglected. Even within the 802.3 format, the applications only support a very non standard version of the protocol. Also, IP packets with options enabled (e.g., record route) are not dealt with at all. The functionality of the applications is also rather limited in that they offer only IP forwarding and that there is no support for more advanced features such as Intserv or Diffserv. However, with its programmable microengines, it is exactly these kinds of applications that the IXP1200 is targeted towards. We are confident that such solutions will appear in the near future, especially since the programmability and low cost of the IXP1200 makes it a very appealing platform for companies other than Intel to develop applications for. Another area of research that can be looked into is the interaction of the microengines with the StrongArm core.

References

[1] Z. Cvetanovic and R.E. Kessler. Performance Analysis of the Alpha 21264-based Compag ES40 System. In *Proceedings of ISCA-2000*, pages 192–202, June 2000.

[2] T.R. Halfhill. Intel Network Processor Targets Routers. *Microprocessor Report*, pages 1–10, September 1999. http://developer.intel.com/design/network/IXP1200.htm.

[3] T.R. Halfhill. EEMBC Releases First Benchmarks. *Microprocessor Report*, pages 38–45, May 2000.

[4] T.R. Halfhill. Lexra's NetVortex Does Networking. *Microprocessor Report*, pages 37–41, July 2000.

[5] T.R. Halfhill. Sitera Samples its First NPU. *Microprocessor Report*, pages 53–56, May 2000.

[6] K. Krewell. Agere's Pipelined Dream Chip. *Microprocessor Report*, pages 34–36, June 2000.