On the Feasibility of Using Network Processors for DNA Queries

Herbert Bos
Vrije Universiteit†
Amsterdam, The Netherlands
H.Bos@few.vu.nl
† formerly affiliated with Universiteit Leiden

Kaiming Huang
Universiteit Leiden
Leiden, The Netherlands
khuang@liacs.nl

Abstract

So far network processors units (NPUs) are exclusively used in networking systems. In this field they form an attractive solution as they offer high throughput, full programmability and are expected to scale with link speed. We observe that bio-computing fields like DNA processing share many of the properties and problems of high-speed networks. At the same time, we observe that DNA processing suffers from sub-optimal performance when implemented on general-purpose processors. Many of the problems are caused by the fact that the potential parallelism in DNA processing cannot be exploited by general purpose processors. For this reason, we evaluate the suitability of NPUs for implementing well-known DNA processing algorithms, such as 'Blast'. To achieve a realistic performance measure, the first (parallelisable) stage of Blast has been implemented on an Intel IXP1200 network processor and used to process realistic queries on the DNA of a Zebrafish. It is shown that the performance of a 232 MHz IXP1200 is comparable to that of a 1.8 GHz Pentium-4.

1 Introduction

The term network processor unit (NPU) is used to describe novel architectures designed explicitly for processing large volumes of data at high rates. Although the first products only became available in the late 1990s, a huge interest, both industrial and academic, has led to there now being over 30 different vendors worldwide, including major companies like Intel, IBM, Motorola, and Cisco [9]. The prime application area of NPUs has been network systems, such as routers and network monitors that need to perform increasingly complex operations on data traffic at increasingly high link speeds. Indeed, to our knowledge, this has been the *only* application domain for NPUs to date. We observe, however, that bio-computing fields like DNA processing share many of the properties and problems of high-speed networks. For example, in well-known algorithms like *Blast* and *Smith-Waterman*, a huge amount of data (e.g., the human genome) is scanned for particular DNA patterns [2]. This is similar to inspecting the content of each and every packet on a network link for the occurrence of the signature of an 'internet worm' (a typical, albeit rather demanding intrusion detection application [13]).

We also observe that many bio-computing fields suffer from poorly performing software when run on common processors, which to a large extent can be attributed to the fact that the hardware is not geared for high throughput or the exploitation of parallelism. For example, in our lab the performance of the Blast algorithm was improved significantly when implemented on an FPGA compared to the identical algorithm on a state-of-the-art Pentium [12]. Similarly, using an array of 16 FPGAs in the Biocceleratorproject, the Smith-Waterman algorithm improved by a factor of 100 – 1000 [8]. Unfortunately, FPGAs are hard to program and VHDL programmers are scarce. NPUs are an interesting compromise between FPGAs and general purpose processors; they can be programmed in C and exploit parallelism to deal with high rates. Indeed, their design is optimised explicitly to deal with such data streams.

For this reason, we evaluated the suitability of NPUs for implementing well-known DNA processing algorithms, in particular 'Blast'. To achieve a realistic performance measure, the first (parallelisable) stage of the Blast algorithm has

been implemented on an Intel IXP1200 network processor and used to process realistic queries on the DNA of a Zebrafish. The implementation will be referred to as *IXPBlast* throughout this paper.

The contribution of this paper is two-fold. First, to our knowledge this is the first attempt to apply NPUs in an application domain that is radically different from network systems. Second, it explores in detail the design and implementation of one particular algorithm in bio-informatics on a particular platform (the IXP1200). The hardware that we have used in the evaluation is by no means state of the art. For example, the IXP1200 clocks in at a mere 232MHz and provides no more than 24 microengine hardware contexts, while more modern NPUs of the same family operate at clock speeds that are almost an order of magnitude higher and offer up to 128 hardware contexts. Nevertheless, we feel confident that the results, bottlenecks identified, and lessons learned apply to many different types of NPU, including the state of the art. The ultimate goal of this work, of which the results presented in this paper represent only a first step, is to empower molecular biologists, so that they will be able to execute complicated DNA analysis algorithms on relatively cheap NPU boards plugged into their desktop computers. This contrasts sharply with current practices, where biologist often have to send their queries to a specialised analysis department which then runs the algorithm on a costly cluster computer (such as the BlastMachine2 [11]) and subsequently returns the results.

The remainder of this paper is organised as follows. The software and hardware architecture is discussed in Section 2. Section 3 discusses implementation details. The implementation is evaluated in Section 4, while related work is described in Section 5. In Section 6, we conclude and attempt to evaluate the suitability of (future generations of) NPUs for DNA processing.

2 Architecture

In this section, we discuss the problem area and the Blast algorithm, as well as the hardware and software configuration to implement the algorithm.

2.1 Scoring and aligning

IXPBlast is an implementation on Intel IXP1200 NPUs of the most time-consuming stage of the Blast algorithm for genome searches. Blast (Basic Local Alignment Search Tool) allows a molecular biologist to search nucleotide and protein databases with a specific *query* [2]. For illustration purposes, we assume in this paper that all relevant information consists of DNA nucleotides (of which there exist exactly four, denoted by 'A', 'C', 'G' and 'T' respectively). Nevertheless, the same implementation can be used for other types of sequences as well (e.g., amino acids or proteins).

Blast is designed to discover the *similarity* of a specific sequence to the query, by attempting to align the two. For example, if the query exactly matches a subset of the sequence in the database, this will be a perfect match, resulting in the highest possible score in the similarity search. However, even if two sequences differ at various places, they might still be "similar" and hence interesting to the molecular biologist. For instance, researchers may have located a gene (a sequence of DNA nucleotides) on the DNA of a Zebrafish that controls, say, the thickness of the Zebrafish's spinal cord and wonder whether a similar gene exists in the human genome. Even though the match may not be perfect, a strong similarity might hint on similar functionality. The differences might be very small (e.g. the sequences are basically the same except for a few strands of 'dead' - non-functional - DNA), or more complicated (e.g. while most of the DNA is present, some subsequences of the nucleotides have moved). Such sequences should also score high on the similarity test.

Denoting the large sequence in the database as DNA_{db} (e.g. the human or zebrafish genome), and the (smaller) sequence that is searched for in the DNA_{db} as the *query* (e.g., the DNA sequence of a specific gene), we can now distinguish two phases in the Blast algorithm. In Phase 1, the *scoring* is done. Given a query, this phase will keep track of the location where a (possibly small) subset of the query matches a subset of DNA_{db} to which it is compared. In genome searches, this phase is the most compute intensive and exhibits a lot of potential parallelism. For example, given n computers, each processor could be used to score one n^{th} of DNA_{db}. In Phase 2 the 'scores' are used to perform the actual *alignment* by searching for the area with maximum overall score (for instance because a number of consecutive substrings have matched). This area represents the part of DNA_{db} that is most 'similar' to the query and the overall score indicates how similar the two sequences are. By its nature, this phase is predominantly sequen-

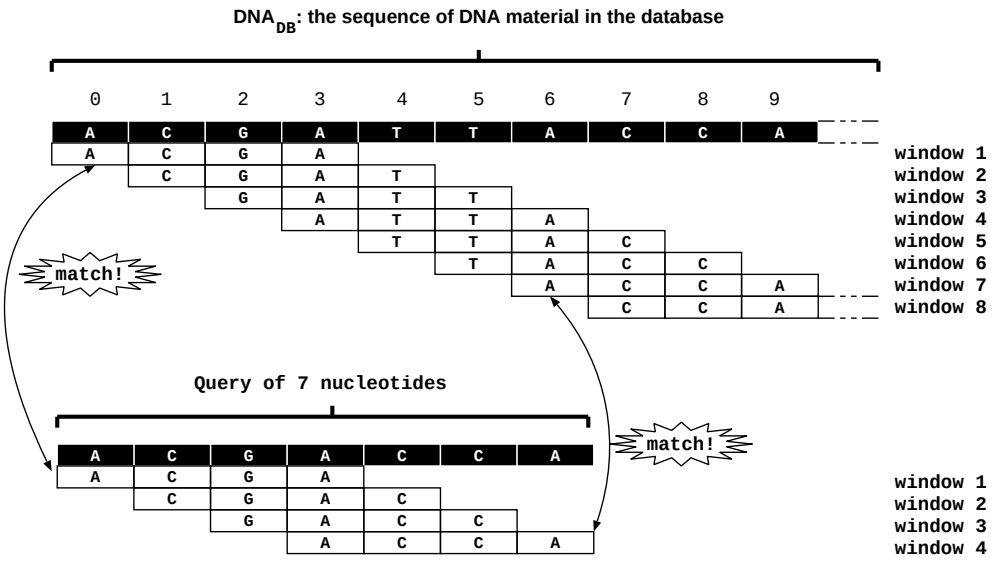


Figure 1. Sequences, queries and windows

tial. We therefore concentrate solely on Phase 1 of the Blast algorithm, and do not consider Phase 2 at all, other than to remark that it is handled using a conventional implementation on a Pentium processor.

Although Blast resembles certain algorithms in networking (e.g. programs that scan the contents of a packet for the signature of a worm), there are some differences. Where 'signature search' algorithms often try to find an exact match for an entire string, Blast also performs 'partial' matches. In other words, whenever a substring of a certain minimum size in the query matches a substring of the same size in DNA_{db}, this is logged. In this paper, this minimum size is referred to as the *window size*.

For example, in Figure 1 the DNA_{db} that is shown at the top is scanned for the occurrence of a pattern that resembles the query shown at the bottom. As partial matches also count, the query is partioned in overlapping windows which are all individually searched for in DNA_{db}. Whenever a window matches, the position where the match occurs is recorded. Figure 1 illustrates the use of a window size of 4. As a trivial example, suppose that the genome at the top of the figure is 100 nucleotides long. Parallelism can be exploited by searching for the query in the first segment of 50 nucleotides of DNA_{db} on one processor and in the remaining segment of 50 nucleotides on a second processor. Note that with a window size of four, three windows that start in segment 1 will end in segment 2, so care must be taken to handle these cases correctly. In the figure, two matches are shown: window 0 (ACGA) matches at position 0, and window 3 (ACCA) matches at position 6. So the pairs (0,0) and (3,6) are recorded. Phase 2 looks at where each of the windows was matched (scored) and attempts to find an area with 'maximum score' (e.g., where many consecutive windows have matched, but which may exhibit a few gaps or mismatches). Every window in the query should be compared to every window in DNA_{db}. Denoting the window size as W, the number of nucleotides in DNA_{db} as N and the number of nucleotides in the query as M, the number of nucleotide comparisons that is needed to process DNA_{db} in a naive implementation is given by: $W \cdot (N - W + 1) \cdot (M - W + 1)$.

A more efficient matching algorithm, in terms of number of comparisons, is known as Aho-Corasick [1]. In this algorithm, all the windows in the query are stored as a single trie structure. After setting up the trie, DNA_{db} is matched to all the windows at once, one nucleotide at a time (see also Section 2.4). Aho-Corasick may be said to trade the number of comparisons for the number of states in the trie. For example, given a window size W, a (theoretical) upperbound to the number of states in the trie (independent of the query) would be $\sum_{i=0}^{W} 4^i$ (capable of representing all possible combinations of sets of W nucleotides). In prac-

tice, however, the number of states would be much fewer. In the query used in this paper, known as MyD88 among molecular biologists, the number of nucleotides is 1611 and the number of states is a little over 10^4. The number of comparisons (and state transitions) in Aho-Corasick equals the number of nucleotides in DNA_{db}.

Both the naive strategy with direct comparisons (**IXPBlast**$_{direct}$) and Aho-Corasick (referred to simply as **IXPBlast** in this paper) were implemented on the network processor. Results are discussed in Section 4. IXPBlast$_{direct}$ was included to provide insight in the performance of the IXP1200 when performing a fixed and large set of operations and as a baseline result. The main focus of our research, however, is on IXPBlast, due its greater efficiency.

2.2 Hardware Configuration

The IXPBlast hardware configuration is shown in Figure 2. A general-purpose processor (Pentium-III) on a host is connected to one or more IXP1200 evaluation boards plugged into the PC's PCI slots. Each of the boards contains a single IXP1200 NPU, on-board DRAM and SRAM (256MB and 8MB respectively) and two Gigabit Ethernet ports. In our evaluation we used only a single (Radisys ENP-2506) NPU board.

The NPU itself contains a two-level processing hierarchy, consisting of a StrongARM control processor and six independent RISC processors, known as *microengines* (MEs). Each ME supports 4 hardware contexts that have their own program counters and register sets (allowing it to switch between contexts at zero cycle overhead). Each ME runs its own code from a small (1K) instruction store.

MEs control the transfer of network packets to SDRAM in the following way. Ethernet frames are received at the MACs and transferred over the proprietary IX bus to an NPU buffer in 64 byte chunks, known as *mpackets*. From these buffers (called RFIFOs by Intel, even though they are not real FIFOs at all) the mpackets can be transferred to SDRAM. MEs can subsequently read the packet data from SDRAM in their registers in order to process it. While this description is correct as a high-level overview of packet reception, the actual reception process is a bit more involved. In Section 2.3, a detailed explanation of packet reception in IXPBlast is given.

On-chip there is also a small amount of RAM, known as 'scratch memory' (4KB on the IXP1200)

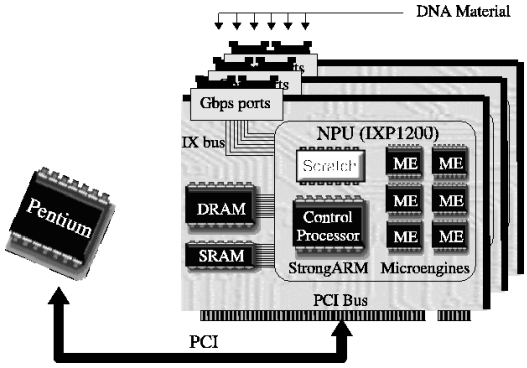


Figure 2. The IXP hardware configuration

memory	size	access time (cycles)
general purpose registers	128 * 4 bytes per ME	-
scratch memory	1K words	12-14
SRAM	8MB	16-20
SDRAM	256MB	33-40

Table 1. IXPBlast memory hierarchy

that is somewhat faster than SRAM, but considerably slower than registers. Table 1 summarises the memory hierarchy and the access time for the most important types of memory (timings obtained from [5]). Interestingly, the differences in access times between Scratch and SRAM (and SRAM and SDRAM), is 'only' a factor of $1.5 - 2$. All buses are shared and while the IX bus is fast enough to sustain multiple gigabit streams, memory bus bandwidth might become a bottleneck. Fortunately, the IXP provides mechanisms for latency hiding, e.g., by allowing programs to access memory in a non-blocking, asynchronous way.

In our configuration, practically all the code runs on the MEs. This means that the code needs to be compact, with an eye on the small instruction stores. The size of the instruction store also means that MEs cannot afford to run operating system code. Instead, programmers run their code directly on the hardware. The StrongARM is used for loading the appropriate code in the ME instruction stores, for starting and stopping the code and for reading the results from memory. In our configuration it runs a Linux 2.3.99 kernel, which we log on to from the Pentium across the PCI bus. In the first stage of the Blast algorithm, the Pentium is not used for any other purposes. The second stage, however, runs solely on the Pentium and does not concern the NPU at all.

2.3 Software architecture

Given that the Blast *program* was to be implemented on an IXP1200, the first problem that needed to be solved was that of accessing the *data*, e.g., determining where to store DNA_{db} and where to store the query.

Accessing the DNA_{db} sequence

Starting with the former, many of the options are immediately ruled out, due to the size of the data sequences, which can be several gigabytes. In fact, the only viable storage locations for this data are the IXP's SDRAM, and/or some form of external storage (e.g. the host's memory, or an external database system). As the ENP2506 only has 256 MB of SDRAM memory (of which some is used by Linux), it is not possible to load DNA_{db} in its entirety in the IXP's memory. On the other hand, since Blast requires access to the entire data, it needs to be 'as close as possible', for efficiency. This effectively rules out any solution that depends on DNA_{db} remaining on external storage throughout the execution (e.g. continuously accessing data in host memory across the PCI bus). So, data has to be streamed into the IXP from an external source.

Again, there are only two ways of streaming DNA_{db} into the NPU: (1) across the PCI bus, (2) via the network ports. We have chosen the latter solution for a number of reasons. First, IXPBlast is intended as a 'network service for molecular biologists'. For this reason, the ability to stream network packets to network cards from various places on the network is quite convenient. Second, in our case, the PCI bus that connects the IXP1200 to the Pentium is not very fast (64/66) and hence *might* be stretched by the gigabit speeds offered by the 2 gigabit network ports. Third, even if a faster bus had been available, it is commonly stated (true or not) that network speed will outgrow bus speed (for example, OC-768 operates at 40 Gbps, which is beyond the capacity of next-generation PCI buses like PCI-X 533). As our aim was to evaluate NPUs with an eye on applying it in a variety of DNA processing fields, not just Blast, the solution that (in potential) provides the highest throughput seemed the best. Fourth, this decision seems in line with what NPUs were intended for: processing data packets coming in from the network. Future versions of NPUs might try to optimise the handling of network packets even more and we should therefore aim for a design that would benefit from such new features.

The price that we pay for this, of course, is that we now have to dedicate valuable processing time on the IXP to packet reception, stripping off protocol headers, etc. Since the Blast algorithm performs a fair amount of computation per chunk of data (and on an IXP1200 is not able to cope with gigabit speeds anyway), it would have been significantly faster, in retrospect, to let the Pentium write the data in the IXP's SDRAM directly.

It should be mentioned that in IXPBlast_{direct}, the naive implementation, the encoding is deliberately suboptimal in size. That is, although each nucleotide *can* be encoded with as few as two bits, we found that this may yield less than optimal performance, as a result of all the extra bit shifting/bit extraction that is required. This is explained in more detail in Section 2.5.

Accessing the query

While DNA_{db} does not fit even in the IXP's SDRAM, this is not the case for the query. Meaningful queries are generally much smaller than full DNA_{db}. Ideally, the query would be fully contained in the NPU's registers. However, this is only possibly for very small queries[1]. In practice, query sizes range from a few hundred to a few thousands nucleotides. This will easily fit in SRAM and possibly even in the 4KB of scratch memory, depending on how nucleotides are encoded.

For now, we assume that in IXPBlast_{direct} the query is stored in scratch memory. We may reasonably expect scratch memory to be sufficiently large, especially considering that newer versions of the IXP have much larger scratch spaces (e.g. 16kB on the IXP2800). Even so, if a query becomes even larger than that, it is trivial to use SRAM instead. As indicated by Table 1 the difference in access between scratch memory and SRAM in the IXP1200 is less than 50%.

In the Aho Corasick implementation of IXPBlast we are not so fortunate. The number of nodes in the trie quickly expands to a size that exceeds the capacity of scratch memory. For this reason, the entire query in IXPBlast is always stored in SRAM.

[1] Assuming we use all 128 general-purpose registers and a 2-bit encoding for the nucleotides, the theoretical upperbound would be a query of $128 \times 32/2 = 2048$ nucleotides, which is still realistic. However, this would leave no registers for the computation

Overview of packet processing

In Figure 3, we have sketched the lifecycle of packets containing DNA material that are received by IXPBlast and the way that the implementation processes this data. The MEs are depicted in the center of the figure. Collectively, they are responsible both for receiving DNA_{db} and for matching the DNA material in this sequence against the query. We will now discuss the numbered activities in some detail.

1. *Packet reception.* In this step, DNA_{db} is sent to the NPU across a network in fixed-length IP packets (of approximately 500 bytes payload) and received (in chunks of 64 byte mpackets) first in the NPU's RFIFO and subsequently in SDRAM. We have dedicated one ME (ME_0) to the task of packet reception. The ME is responsible for (a) checking whether a packet has arrived from the network, (b) transferring the DNA material to a circular buffer in SDRAM, and (c) indicating to the remaining 5 MEs that there is DNA material to process.

2. *Query in SRAM (or Scratch).* The query is stored in faster memory than the packets. In the implementation of Aho-Corasick the query is stored in SRAM (2a), as it does not fit in the 4KB scratch memory, except for very small queries. In the naive implementation (IXPBlast_{direct}), the entire query is stored in on-chip scratch memory for efficiency (2b). As Scratch memory is only 4KB on IXP1200s, this limits the size of the query. In our implementation, as discussed in Section 3, we are able to store more than 10^4 nucleotides in scratch memory, far exceeding the length of most queries in practice.

3. *Scoring.* While ME_0 is dedicated to packet reception, the threads on the remaining MEs implement the scoring algorithm described in Section 2.1, i.e., every window in the query is compared to every window in DNA_{db} (and an administration is kept for all matches). Phase 2 of the Blast algorithm (the alignment, which takes place on a Pentium processor) starts only after the entire DNA_{db} has been 'scored'.

   Note that when processing a packet, a thread really needs to access *two* packets. The reason is that the last few windows in a packet in DNA_{db} straddle a packet boundary. For example, assume the window size is 12 nucleotides and a thread T processing packet p_i is about to process the last 11 nucleotides in the packet. In this case, the window 'spills' into packet p_{i+1}. As a result, T must access upto 11 nucleotides of p_{i+1}. Conversely, all packets except the first need to be accessed by *two* threads: the one responsible for this packet and the one responsible for processing the *previous* packet. We will call these threads the packet's 'main thread' and 'spill thread', resepctively.

   Processing takes place in batches of B packets. For example, if $B = 100.000$, this means that 100.000 packets are received and processed in their entirety before the next batch is received (this is not to say, however, that all B packets first have to be received before the processing can start). For this purpose, every thread maintains a packet index in Scratch memory. Each of these indices has to point to the end of the batch buffer, before the next batch of B packets arrives. Otherwise, there is a risk that the entire batch cannot be stored. So if any of the indices lags behind, an error indication is generated and the system dies. Obviously, this is not very efficient. For example, if an index points to packet number $B-1$, there would be $B-2$ buffer slots available for receiving packets and there is no need to stop the application. In other words, the current implementation is over-conservative. We plan to fix this in a future implementation.

   It should be stressed that each of the MEs processes only 20% of all packets in the circular buffer. Finally, as shown in Figure 3, consecutive MEs do not need to process consecutive packets. For instance, ME_3 in the figure has overtaken ME_4 and ME_5.

2.4 Aho-Corasick

The code running on microengines ME_1-ME_5 executes the Aho-Corasick string matching algorithm. It is not our intention to explain the algorithm in detail and interested readers are referred to [1]. Nevertheless, for clarity's sake a brief description of how the algorithm works at runtime is here repeated. To simplify things, we assume that the window size is 3. Consider a query consisting of the following windows: `{acg,cgc,gcc,ccg,cga}`. This is a query of length 7 that consists of 5 windows. The be-

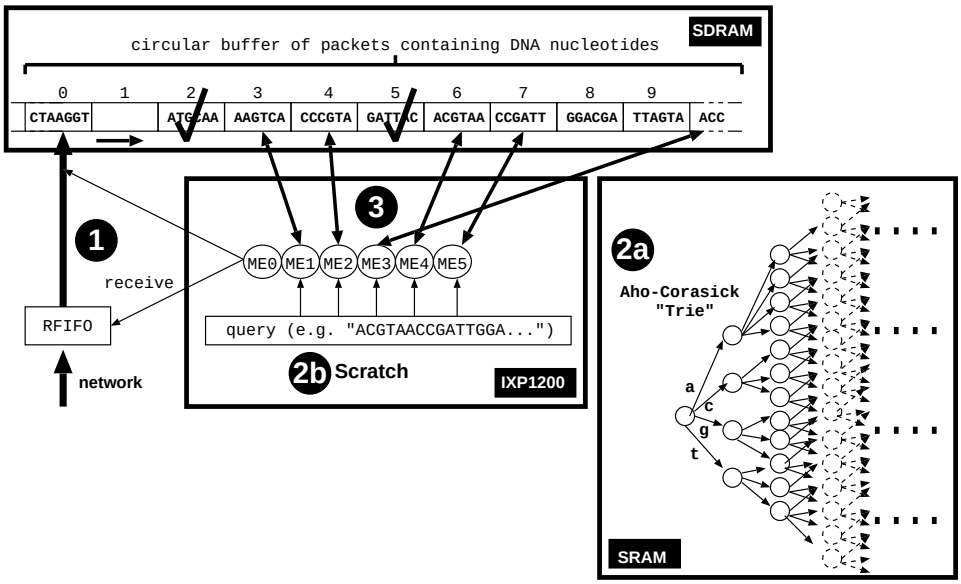


Figure 3. Processing packets in IXPBlast

haviour of the pattern matching machine is determined by three functions: a goto function g, a failure function f and an output function $output$. For the example query, these functions are shown in Figure 4.

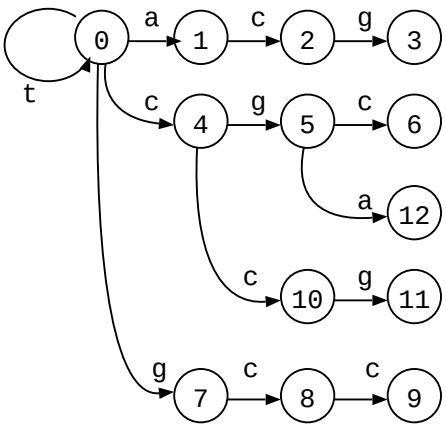
State 0 is the start state. Given a current state and an input symbol from DNA_{db} (the next nucleotide to match), the goto function g makes a mapping to a 'next state', or to the message *fail*. The goto function is represented as a directed graph in the figure. For example, the edge labelled a from 0 to 1 means that $g(0, a) = 1$. The absence of an arrow indicates *fail*. Whenever the *fail* message is returned, the *failure* function is consulted which maps a state to a new state. When a state is a so-called *output state*, arriving here means that one or more of the windows have matched. In IXPBlast, this translates to recording both the corresponding positions in DNA_{db} and the window(s) that matched. This is formalised by the output function that lists one or more windows at each of the output states.

The pattern matching machine now operates as follows. Suppose the current state is s and the current nucleotide of DNA_{db} to match is x.

1. If $g(s, x) = s'$, the machine makes a *goto transition*. It enters state s' and continues with the next nucleotide in DNA_{db}. In addition, if $output(s) \neq empty$, the windows in $output(s')$ are recorded, together with the position in DNA_{db}.
2. If $g(s, x) = fail$, the failure function is consulted and the machine makes a *failure transition* to state $s' = f(s)$. The operating cycle continues with s' as the current state and x as the current nucleotide to match.

Suppose for instance that the following subset of DNA_{db} is processed: `tacgcga`. Starting in state 0, the algorithm first processes `t` which brings about no (real) state transistion. Next, `a` is processed which lead to a move to state 1 on a goto transition. Subsequent goto tranistions for the following two nucleotides, `c` and `g` respectively, move the algorithm to state 3. State 3 happens to be an output state, so the window `acg` has matched. The next nucleotide in the input is `c`. As there is no goto transition from state 3 for this nucleotide, the algorithm makes a failure transition to state 5, continueing with the same input nucleotide `c`. From state 5 it makes a goto transition to state 6, which again is an output state (the window `cgc` was matched). The next nucleotide in the input is `g`. Again, a failure transition is made, this time to state 8 and processing continues with nucleotide `g`. The next failure transition leads to state 4 and from there a goto transition *can* be made bringing us to state 5. The nucleotide to match is `a` for which, again, a goto transition is possible. The new state is 12, which is an output state for the window `cga`. At this point, all three windows hidden in the input string have been matched.

The explanation above describes the essence of



(a) Goto function

i	1	2	3	4	5	6	7	8	9	10	11	12
f(i)	0	4	5	0	7	8	0	4	10	4	5	1

(b) Failure function

i	*output(i)*
3	acg
6	cgc
9	gcc
11	ccg
12	cga

(c) Output function

Figure 4. The Aho-Corasick pattern matching machine.

the algorithm. In their paper Aho and Corasick describe how it can be implemented even more efficiently as a finite automaton which makes exactly one state transition per input nucleotide. There are a few observations to make from the above. First, the operation of the algorithm at runtime is extremely simple. Indeed, the complexity of Aho-Corasick is not the operating cycle, but the generation of the appropriate trie. The construction of the trie consists of executing a sequence of three straightforward algorithms. The precise nature of these algorithms is beyond the scope of this paper and interested readers should refer to the original paper by Aho and Corasick. Second, the algorithm matches each nucleotide of DNA_{db} against *all* windows at once. This means that the algorithm scales well for increasing numbers of windows and window sizes. Indeed, the size of the query has hardly any impact on the processing time, in contrast to most current implementations of the Blast algorithm. Even without considering implementation on NPUs, this is an interesting property. Third, the algorithm requires a relatively large amount of memory. Recent work has shown how to make the Aho-Corasick algorithm more memory efficient [14], but in this paper the original algorithm was used, as it is faster and a few MB of SRAM is more than enough to store even the largest BLAST queries.

2.5 Nucleotide encoding

In DNA there are only 4 nucleotides, A, C, G and T, so 2 bits suffice to encode a single nucleotide. In fact, in the Aho-Corasick version of IXPBlast both the query and DNA_{db} are encoded in this fashion.

When encoding DNA_{db} in the naive implementation, however, some encoding efficiency was sacrificed for more efficient computation (making the implementation slightly less naive). The advantage of an alternative encoding is that it enables us to reduce the number of shift and mask operations that are needed in the comparisons on the mciroengines. The way this is done is by performing these operations *a priori* at the sender's side. For example, if in a hypothetical system the window size is two nucleotides, the minimum addressable word size of the memory is 4 bits and DNA_{db} is '`ACGT`', several shift/mask operations may be necessary to get to the right bits if an optimal 2-bit encoding is used. Let's assume that A=00, C=01, G=10, and T=11. In this case, DNA_{db} is encoded as `0001 1011` and to get to window '`CG`' some shifting and masking would be hard to avoid. However, no shift/masks operations are needed if the following encoding is used: `0001 0110 1011 11`... (to each 'original' encoding of a nucleotide the 'original' encoding of the next nucleotide is appended). Now each consecutive window is found simply by reading the next 4-bit word and comparing this word to the query window *directly*.

Of course, a less efficient encoding means more overhead in transferring the data in and out of memory and microengines. In practice, we discovered that an encoding of DNA_{db} of 4 bits gave the best performance.

3 Implementation details

DNA_{db} is sent to the IXP from a neighbouring host across a gigabit link. The sender uses `mmap` to slide a memory mapped window over the original DNA_{db} file. The content of the file that is currently in the window is memory mapped to the sending application's address space, allowing the sender to read and send the data efficiently. Packets are sent in fixed-size UDP/IP Ethernet frames of 544 bytes where the UDP header is immediately followed by a 6 bytes pad. The pad ensures that the DNA payload starts at an 8 byte boundary (14B Ethernet + 20B IP + 8B UDP + 6B pad = 48), which is convenient as the minimum addressable unit of SDRAM is 8 bytes.

Careful readers may have noticed that this UDP setup is problematic. As we have said that the current IXPBlast implementation on the IXP1200 cannot keep up with the linespeed, without some feedback mechanism the sender has no way of knowing how fast it can send the data without swamping the receiver. In the current version, we have ignored this problem and manually tuned the sender in such a way that packets are never dropped by the receiver.

In the implementation of IXPBlast, 2 threads on ME_0 are dedicated to receiving packets from the network and storing them in SDRAM. A third thread synchronises with the remaining MEs. Dedicating more than 2 threads to packet reception does not improve performance. The implementation is much simpler than in most networking applications: ME_0 loads data in SDRAM in batches of 100.000 packets. As soon as packets are available the processing threads start processing them. They synchronise with ME_0 by reading a counter each time they have finished processing a packet. If no new packet is available, they wait. On ME_1-ME_6 this is essentially the only synchronisation that is necessary for a block of 100.000 packets to be processed. In network systems, such as routers, such simplistic synchronisation is generally not permitted.

On the remaining MEs two threads are used to process the data, resulting in 10 packet processing threads in total. Again, using more threads did not improve the performance and even resulted in performance degradation (possibly due to the additional resource contention). Each $thread_i$ ($0 \leq i < 10$) is responsible for its own subset of the packets as follows: $thread_i$ processes packets $i, i+10, i+20, i+30, ...$ The thread compares each of the windows in the query to each window in the packet for which it is the 'main thread' and to each window that starts in this packet and spills over into the next packet. For $IXPBlast_{direct}$, this means a direct comparison with each of the windows in the query, while for IXPBlast it means parsing one molecule at a time and comparing to all the windows at once.

IXPBlast works with a window size of 12 (or 3 amino acids), a size suggested in the literature [6]. Each window has a *score list*, kept in SRAM that tracks the positions in DNA_{db} that match the window. Whenever a window match is found, the location in DNA_{db} is noted in the window's scorelist. After DNA_{db} has been processed in its entirety, the scores are the 'end-result' of Phase 1. They are used as input for the remaining sequence alignment algorithm (Phase 2) on the Pentium.

In IXPBlast, all initialisation of memory, receiving of packets, storing of packets in SDRAM and synchronisation with the remaining MEs takes up 235 lines of microengine C code. The packet processing functionality on ME_1..ME_5 was implemented in 292 lines of code.

4 Results

To validate our approach, IXPBlast was applied to the current release of the Zebrafish shot gun sequence, a genome of a little over $1.4 \cdot 10^9$ nucleotides, using a query that consisted of the 1611 molecules long cDNA sequence of the Zebrafish MyD88 gene [7]. The window size is 12 molecules, so the number of windows is 1600. The results are compared to an equivalent implementation of the algorithm on a Pentium.

Throughout this paper, the process of obtaining DNA_{db} (either from disk or from the network) and storing it in memory is referred to as 'DNA_{db} *loading*'. As explained in Section 3, the loading process in the current implementation of IXPBlast is somewhat inefficient. The reason is that, due to the lack of a feedback mechanism, the sender is forced to send at an artifically low rate to prevent the IXP from being swamped. As a result, when transferring the Zebrafish genome (approximately 1.4×10^9 nucleotides) the sender spends a small amount of its time sleeping. For the implementation on the Pentium, the sequence was read from a local disk and stored in host memory. In the measurements reported in clock cycles, the overhead of loading is not included. Arguably, doing so introduces a bias in the results to our advantage. We think this is not very serious, for two

reasons. First, we are trying to evaluate the use of network processors for DNA processing and for this the loading across the network is an orthogonal activity (as mentioned earlier, one could use the same loading method as used for the Pentium). Second, including loading in the current implementation of DNA_{db} in the comparison does not make much sense since a significant number of cycles are 'idle cycles', that are spent waiting for the sender to wake up.

However, the total time in seconds is also reported and this does include the loading time. For the Pentium implementation this only involves reading from the hard drive. For the implementation on the IXP, this involves both reading from the hard drive and transmission across the network, so in this case the results would seem to biased in the advantage of the Pentium. The reason that the total time on the Pentium is longer than the one on the IXP is probably caused by the fact that in the former case reading DNA_{db} and processing the data are done sequentially, while in the latter case these are fully parallelised activities. The results are summarised in Table 2.

Experiments 4–8 in Table 2 were obtained from the cycle-accurate IXP simulator, provided by Intel. Experiments 5–8 will be explained below. Experiment 4 is listed to get a handle on how many cycles are typically spent per packet by a single thread. We can use this result to verify the total processing time. DNA_{db} consists of a total 810754 packets of 1792 nucleotides each, so that the total time to process the entire DNA_{db} by a *single* thread (not including loading) can be estimated as $(301028 \times 810754)/(232 \times 10^6) = 1051$ sec. For 10 threads the time would be roughly 105 seconds. If we include the overhead incurred by the loading (including sleeps), a total result of 129 seconds as measured is very close to what we expected.

The number of cycles per molecule is $301028/1792 = 168$ cycles. As the current implementation has not focused on optimisation much, we feel confident that this number can be brought down in future versions of the code. It should be mentioned that for the measurement, only a single packet was processed, as simulating the processing of the entire genome takes an exceedingly long time. Interestingly, the performance of the 1.8GHz P4 is very close to that of the 232MHz IXP1200. The number of clock cycles in experiment 1 on a Pentium 4 does not include loading. Converting clock cycles into seconds, the processing time on a Pentium comes to 97 seconds (slightly better than the estimated actual processing time on the IXP). With the naive implementation, IXPBlast_{direct} (shown as Experiment 1 of Table 2), processing even a mere 10.000 packets (containing only 792 nucleotides each) takes as long as 140 seconds. With this implementation it would take approximately 359.5 minutes (almost 6 hours) to process the entire genome. Recently, we also implemented the more efficient implementation of Aho-Corasick in which all failure transitions are eliminated and the entire algorithm is reduced to a deterministic finite automaton. The total processing time on the IXP now comes to just 90 seconds.

Observe that the processing time in IXPBlast is determined by the size of DNA_{db} and to a much lesser extent by the size of the query. The explanation is that Aho-Corasick always leads to a state transition when a new nucleotide is processed (eventhough the old and new state may be the same). The only difference is in the size of the trie in memory and the number of matches of windows that are found for a query. A 'match' results in slightly more work than 'no match', as the score-keeping must be done in SRAM.

In the simulator, subsets of the original 1611 nucleotides MyD88 query were used to construct queries of 200,400,...,1400 nucleotides. All queries were used to process two packets each with a length of 1611 nucleotides: (1) where there were no matches, and (2) where the packet was exactly the same as the 1611 nucleotide query. Both experiments are run at the same time with one thread on one ME processing case (1) and another thread on a different ME processing case (2). As a result there will be some interference between the 2 experiments (but very little as each experiment runs on a separate ME). If anything, the results will be better when run in complete isolation, rather than worse. Case (1) incurs no updates of SRAM, and is considered the fastest possible case. Case (2) represents the maximum workload as it will match every window in the 1611 nucleotide query. It also exhibits the greatest difference between the queries, as the short sequences will only encounter a larger number of matches in the beginning of the packet, but much fewer towards the end (see also Table 3). The two extremes (200 and 1611 nucleotides) are shown as Experiments 5–8 in Table 2. As shown in Figure 5, there is hardly any difference in performance for the packet with no matches. These results are for the implementation of the Aho-Corasick algorithm as a deterministic finite automaton. We speculate that the varying sizes of the tries would have made more of a differ-

Exp	Query size (nucleotides)	DNA_{db} size (nucleotides)	Implementation	Cycles	Time (s) (incl. loading)
1	792	8.96×10^6 (10^4 pkts)	$IXPBlast_{direct}$, IXP1200 (232 MHz)	not measured	140 seconds
2	1611	1.4×10^9	Pentium-4 (1.8 GHz)	174741844482	132
3	1611	1.4×10^9	IXPBlast, IXP1200 (232 MHz)	not measured	129
4	1611	1792 (1 *real* pkt)	IXPBlast, IXP1200, 1 thread only (simulated)	301028	not measured
5	1611	1792 (1 no-match pkt)	IXPBlast, IXP1200, 1 thread only (simulated)	288655	not measured
6	1611	1792 (1 max-match pkt)	IXPBlast, IXP1200, 1 thread only (simulated)	418047	not measured
7	200	1792 (1 no-match pkt)	IXPBlast, IXP1200, 1 thread only (simulated)	288373	not measured
8	200	1792 (1 max-match pkt)	IXPBlast, IXP1200, 1 thread only (simulated)	303821	not measured
9	1611	1.4×10^9	IXP1200, 232MHz with implementation of Aho-Corasick as deterministic finite automaton		90

Table 2. Results for various implementations of Blast

Aspect measured	Results
pkt size (including headers)	544 bytes
number of packets in DNA_{db}	810754
pkt reception (for 1 pkt of 544 bytes)	1677 cycles
size of trie for 1611 nucleotide query	10312 states (≈240 KB)
no. of matches for 400 nucleotide query	109142
no. of matches for 1611 nucleotide query	524856

Table 3. Related IXPBlast measurements

ence in the presence of a cache, but in the IXP1200 no caching takes place, so all accesses to the trie are accesses to SRAM. For this reason, the size of the trie has little impact on total performance.

Recall that in these experiments an old version of the IXP was used, while newer versions offer clock speeds comparable to Pentiums (in addition to faster memory and more MEs). For example, an IXP2800 runs at 1.4GHz and offers 16 MEs with 8 threads each. It is tempting to say that the 'competition' between the IXP and the Pentium will be won simply by the IXP2800's faster clock alone, but this conclusion may be premature. Operating at higher clock rates may increase stalls, e.g. due to memory bus contention. Nevertheless, considering the state-of-the-art in NPUs, it is safe to say that the speed of even the naive implementation may be improved signifantly by better hardware and, given that results of IXPBlast are competitive even with the current hardware, the results are promising.

5 Related work

To our knowledge there is no related work in applying NPUs to DNA processing. In bio-informatics there exists a plethora of projects that aim to accelerate the Blast algorithm, using one of the following methods (1) improving the code (e.g. [10]), (2) use of clusters for parallel processing (e.g. Paracel's BlastMachine2 [11]), and (3) implementation on hardware such as FPGAs (e.g. Decypher [4, 8, 12]). While IXPBlast is related to the work on cluster computers, it has an advantage in that it is cheaper to purchase and maintain than a complete Linux cluster. Implementations of Blast on FPGAs are able to exploit even more parallelism than IXPBlast. On the other hand, they are harder to program (and programmers with the necessary VHDL skills are scarce). Because of this FPGAs are not very suitable for rapid development of new types of DNA processing tools. Moreover, NPUs expect to increase their clock rates at roughly the same pace as current microprocessors such as the Pentium, meaning that the same code experience a performance boost 'for free'. The clock speeds of FPGAs, on the other hand, are not expected to grow quite so fast.

Variations of the Aho-Corasick algorithm are frequently used in network intrusion detection systems like the newer versions of Snort [13]. While we are not aware of any implementation of the algorithm on network processors, there is an implementation of Snort version 1.9.x on an IXP2400 with a TCAM daughter card from IDT which implements the Boyer-Moore algorithm [3]. Boyer-Moore is similar to Aho-Corasick, but is limited to searching for a single pattern at a time. A more memory efficient implementation of Aho-Corasick is presented in [14].

6 Conclusion

In this paper, it was demonstrated that NPUs are a promising platform for implementing certain algorithms in bio-informatics that could benefit from the NPU's parallelism. We have shown how the Blast algorithm was successfully implemented on a fairly slow IXP NPU. The performance was roughly equivalent to an implemen-

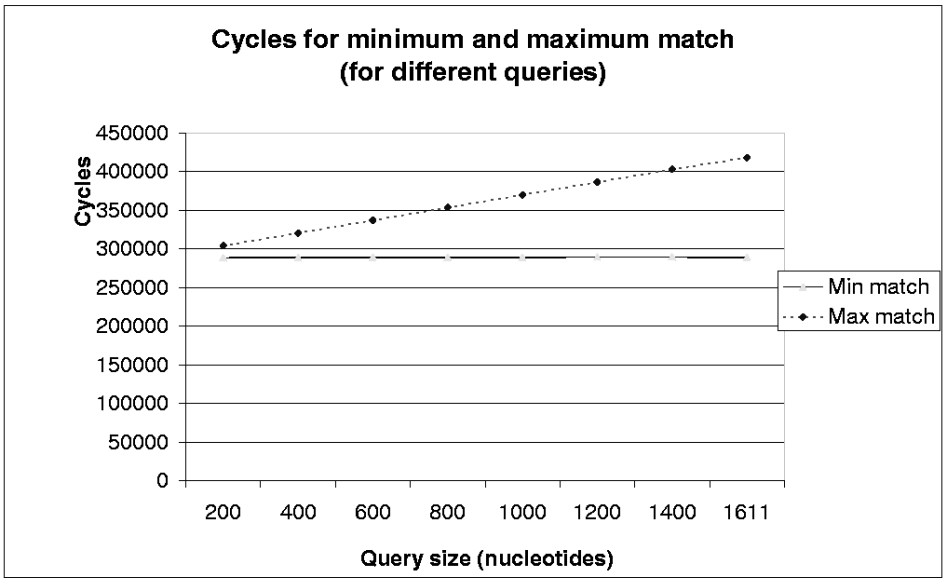


Figure 5. Cycle counts for different queries

tation on a Pentium processor running at a much higher clock rate. As we have made few attempts to optimise our solution, we believe there can be no doubt that NPUs are an interesting platform in fields other than networking in general and in DNA processing in particular.

Although the Aho-Corasick algorithm itself is fairly efficient, many improvements of Blast have been proposed in the literature. Moreover, several simple changes to IXPBlast are expected to boost performance significantly. First, a trivial improvement for IXPBlast is to switch to amino acids (effectively looking at 3 nucleotides at a time). Second, the Aho-Corasick algorithm that was used in IXPBlast can itself be improved. Research projects exist that make the algorithm either faster or more memory efficient (e.g., [14]). So far, we have not exploited any of these optimisations. Third, while implementing the feedback mechanism between sender and IXP will not change the number of cycles spent on processing the data, it will decrease the duration of the process as experienced by the user. Fourth, multiple network processors can be used in parallel by plugging in multiple ENPs in the PCI slots of a PC to speed up the process even more.

We conclude this paper with a speculation. So far, we have described the application of an architecture from the world of networking to the domain of bio-informatics. It may well be that the reverse direction is also useful. In other words, it would be interesting to see whether a similarity search algorithm such as Blast could be fruitfully applied to scanning traffic, e.g. for a 'family of worms' exhibiting a degree of family resemblance. Instead of looking for exact patterns in a flow, this would look for patterns that are similar to the query. For this to work, more research is needed to investigate whether worms that exploit a certain security hole sufficiently resemble each other to allow Blast to separate them from normal traffic.

Acknowledgements

Our gratitude goes out to Intel for providing us with IXP12EB boards and to the University of Pennsylvania for letting us use one of its ENP2506 boards. We are indebted to Bart Kienhuis for discussions about this idea in the early stages of the work and to Fons Verbeek for providing the required knowledge about bio-informatics and for

supplying the DNA material. Shlomo Raikin has been helpful in comparing our approach to the ones that use FPGAs and suggesting reasonable parameters for the experiments. A massive thanks is owed to Hendrik-Jan Hoogeboom for pointing us to the Aho-Corasick algorithm which resulted in a speed-up of several orders of magnitude.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] S.F. Altschul, W. Gish, W. Miller, and D.J Lipman E.W. Myers. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[3] White Paper DeCanio Engineering, Consystant. The snort network intrusion detection system on the intel ixp2400 network processor. http://www.consystant.com/technology/, February 2003.

[4] M. Gollery. Tera-BLAST - a Hardware Accelerated Implementation of the BLAST Algorithm, White Paper. `http://www.timelogic.com/company_articles.html`, August 2000.

[5] Erik J. Johnson and Aaron R. Kunze. *IXP1200 Programming*. Intel Press, 2002.

[6] J. W. Kent. BLAT - The BLAST-like Alignment Tool. *Genome Research*, 12(4):656–664, 2002.

[7] K.A. Lord and D.A. Liebermann B. Hoffman-Liebermann. Nucleotide sequence and expression of a cDNA encoding MyD88, a novel myeloid differentiation primary response gene induced by il6. *Oncogene*, 5(7):1095–7, July 1990.

[8] Compugen Ltd. Bioccelerator product information. `http://eta.embl-heidelberg.de:8000/Bic/docs/bicINFO.html`, 1995.

[9] Kurt Keutzer Niraj Shah. Network processors: Origin of species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, October 2002.

[10] TimeLogic White Paper. Gene-BLAST - An Intron-Spanning Extension to the BLAST Algorithm. `http://www.timelogic.com/company_articles.html`, 2001.

[11] Paracel. Blastmachine2 technical specifications. `ttp://www.paracel.com/products/blastmac`ine2.php, November 2002.

[12] Shlomo Raikin, Bart Kienhuis, Ed Deprettere, and Herman Spaink. Hardware implementation of the first stage of the Blast algorithms. *Submitted for review to the Oxford Journal on Bioinformatics*, 2003.

[13] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*, 1999. Available from `http://www.snort.org/`.

[14] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms foıntrusion detection. In *Proceedings of the IEEE Infocom Conference* [1], pages 333–340.