A Prototype of a Dynamically Reconfigurable Processor Based Off-loading Engine for Accelerating the Shortest Path Calculation with GNU Zebra

Sho SHIMIZU, Taku KIHARA, Yutaka ARAKAWA, Naoaki YAMANAKA
Keio University, 3–14–1 Hiyoshi, Kohoku-ku, Yokohama, 223-8522, Japan
Email: shimizu@yamanaka.ics.keio.ac.jp

Kosuke SHIBA
IPFlex Inc., 2–27–1 Kamiosaki, Sun felista Meguro 6F, Shinagawa, Tokyo, 141-0021 Japan
Email: shiba@ipflex.com

Abstract—A hardware off-loading engine to speed up the shortest path calculation in OSPF (Open Shortest Path First) has been developed. The developed system is co-designed with both hardware and software to optimize an architecture of a router for highly functional Traffic Engineering (TE). To speed up the shortest path calculation, we employ a dynamically reconfigurable processor, IPFlex DAPDNA-2, as a hardware off-loader, and newly structured a novel high-speed parallel shortest path algorithm, called MPSA (Multi-route Parallel Search Algorithm). The proposed algorithm consists of simple processing, in which multiple paths are simultaneously searched by multiple Processor Element (PE) of DAPDNA-2. Therefore, it reduces the execution time of shortest path calculation to 2.8% compared with the popular shortest path algorithm, Dijkstra’s algorithm. Our prototype works together with a famous software-based router, GNU Zebra, on commodity Linux PC. The proposed architecture and prototype system can be applied to future network sophisticated TE.

I. INTRODUCTION

Generalized Multi Protocol Label Switching (GMPLS) [1] is a key technology to control and manage next generation IP backbone networks. It enables not only high speed and large capacity networks but also QoS controls and traffic engineering (TE). TE is an essential technique to utilize network resources efficiently and to avoid network congestion. Routing in GMPLS networks employing TE is based on multiple metrics such as the number of hops, link bandwidth, and transmission delay [2]. In current IP networks, shortest path routing which is based on interface cost only is often used [3]. Path calculation in GMPLS networks will be more complex due to the consideration of multiple metrics.

In addition, in GMPLS networks, topology of IP layer is affected by a lightpath in optical layer. Lightpath establishment leads to change topology of IP layer, and re-calculation of the shortest paths is essential. Therefore, each router frequently re-calculates the shortest paths to create a routing table in GMPLS networks. However, the conventional method for calculating the shortest path in OSPF [3] has $O(n^3)$ computational complexity where $n$ is the number of nodes. Therefore, in large-scale networks, the complexity of the shortest path search become more difficult. Ultra fast shortest path calculation can adopt to huge-size networks.

Conventional approaches to calculate the shortest path is Dijkstra’s algorithm [4] which is suitable for Neumann-type sequential processors and widely used, for example in creating routing tables in OSPF. The way to speed up Dijkstra’s algorithm is to make clock cycle of a processor high since Dijkstra’s algorithm is a sequential algorithm. However, it has limitations to speed up clock cycle because of the power consumption. To solve this problem, reconfigurable processors which is based on new architectures have been studied [5]. In this paper, we employ a new approach that parallel shortest path algorithm is executed on reconfigurable processors to make the breakthrough for speeding up shortest path search. It is a software and hardware mixed approach.

We propose a parallel shortest path algorithm called MPSA (Multi-route Parallel Search Algorithm) based on parallel data-flow type dynamically reconfigurable processors. MPSA is suitable for parallel data-flow machines since it can be expressed as matrix operations. MPSA searches for multiple paths in parallel. Current positions are proceeded by 1 cost unit with each step, and the first path to reach the target node is the shortest path to the node. After the current position reaches the target node, all links of the node are added to the discovered links. We show the proposed algorithm is theoretically less execution time than Dijkstra’s algorithm by about 97% since the proposed algorithm is $O(\sqrt{n})$ while Dijkstra’s algorithm is $O(n^2)$.

We implement a hardware off-loading engine prototype to speed up the shortest path calculation in OSPF. MPSA is implemented as hard-wired logic on DAPDNA-2, which is a commercially available dynamically reconfigurable processor developed by IPFlex Inc., Japan [6]. Our prototype works together with GNU Zebra, a famous software-based router, running on commodity Linux PC. When Zebra creates a routing table in the process of OSPF, our off-loading engine
calculates the shortest path.

II. Related works

A. Shortest path algorithm

Let \( G = (V, E) \) be a directed graph where \( V \) is the set of nodes, and \( E \) is the set of edges. Let \(|E| = m, |V| = n, \) let \( s \) be the source node, and \( c \) be a function assigning a non-negative valued weight to each edge of \( G \). The cost of a link can be thought of as the distance between the two nodes. For each \( v \in V \), \( d(v) \) represents the cost of the shortest path from the source node \( s \) to \( v \). The theoretically most efficient sequential shortest path algorithm is Dijkstra’s algorithm [7]. It calculates the shortest path between the source node and all other nodes. It is expressed as follows.

1) Set \( S \) to empty, where \( S \) is a set of nodes whose shortest paths from the source node \( s \) have already determined.
2) Add the source node \( s \) to \( S \), and \( d(s) = 0 \). If there is a link from \( s \) to \( v \), \( d(v) = c(s, v) \), for all other nodes, \( d(v) = \infty \).
3) Add a node \( u \) to \( S \), where \( d(u) \) is the smallest in \( V - S \). If \( S = V \), complete the algorithm.
4) If there is a link from \( u \) to \( v \in V - S \), \( d(v) = \min(d(v), d(u) + c(u, v)) \). Then go to 3).

Routing in IP networks is based on the shortest paths, a router calculates the shortest path when it creates a routing table. Dijkstra’s algorithm is suitable for Neumann-type processors, and it is employed as the shortest path algorithm in the real routers, for example GNU Zebra [8] and XORP (eXtensible Open Router Platform) [9]. The computational complexity of Dijkstra’s algorithm is \( O(n^2) \) where \( n \) is the number of nodes in a network. Therefore, in large-scale networks with TE functionalities, Dijkstra’s algorithm becomes heavy task for a router.

B. Reconfigurable processor

It has limitations to speed up clock cycle of processors because of the power consumption. To cope with the problems about power consumption, performance, and rapid development, dynamically reconfigurable processors have been developed. Recent dynamic reconfigurable devices have been developed to achieve high performance and flexibility [5].

DAPDNA-2 is a commercially available dynamically reconfigurable processor developed by IPFlex Inc., Japan [6]. DAPDNA-2 consists of two processors, DAP (Digital Application Processor) and DNA (Digital Network Architecture). These processors have different architectures. DAP is a 32-bit RISC CPU which is a Neumann-type processor, and DNA is a parallel data-flow machine. DNA consists of 376 small computing unit called PEs (Processor Elements), which is arranged in an array pattern.

We can design the connections between PEs when implementing an algorithm on DAPDNA-2. Connection structure of PEs can yield a parallel data-flow machine. Each structure is called a configuration. DNA can keep three configurations in own cache. These configurations can be switched within one clock. Thus, this chip combines the advantages of the high-speed processing of hardware and the flexibility of software.

III. Multi-route parallel search algorithm

A. Summary

The basic idea of our proposed algorithm is that the shortest path is the first path reached to a node when we traverse all edges from the source node at the same pace. First, the current positions are set at the source node. After this, all the current positions are simultaneously proceeded by 1 cost unit per process. All the current positions are equally distant from the source node. When the current position reaches a node, we find that the path on which we traverse is the shortest path to the node.

Here, the following notations are used to explain the algorithm.

\( V_f \) The set of nodes whose shortest paths from the source node have already determined.
\( V_r \) The set of nodes which is just reached.
\( E_f \) The set of edges which are presently traversed.
\( E_f \) The set of edges which are determined that they are on the shortest path tree.
\( i, j \) A node in the network
\( r(i, j) \) The remaining cost to reach Node when \( (i, j) \) is traversed where \( (i, j) \in E \).

The following is an accurate procedure of our proposed algorithm.

1) Set \( V_f \), \( E_f \), and \( E_f \) to empty. Set \( V_f = \{s\} \), and \( r(i, j) = c(i, j) \) for \( (i, j) \in E \), \( r(i, j) = 0 \) for \( (i, j) \notin E \).
2) Add node \( i \in V \) to \( V_f \). If \( V_f \) is equal to \( V \), complete the algorithm.
3) Remove the edge \( (i, j) \) from \( E_f \), and set \( r(i, j) \) to zero for all \( i \) where \( j \in V_f \). Add the edge \( (i, j) \) to \( E_f \), where \( (i, j) \in E_f \), \( i \in V_f \), and \( r(i, j) \neq 0 \). Set \( V_f \) to empty.
4) \( r(i, j) = r(i, j) - 1 \) for all the edge \( (i, j) \in E_f \). If \( r(i, j) \) becomes 0, add node \( j \) to \( V_f \). Also add the edge \( (i, j) \) to \( E_f \). Then go to 2).

Figure 1 shows an example of our proposed algorithm. The denominator represents \( c(i, j) \) and the numerator represents \( c(i, j) - r(i, j) \) in edge \( (i, j) \). First subfigure is the initial state of the proposed algorithm. First, the algorithm adds edge \( (1, 2) \) and \( (1, 3) \) to \( E_f \). Next, the current position is proceeded by 1 cost unit. Then \( r(1, 2) = 0 \) and \( r(1, 3) = 2 \), so edge \( (1, 2) \) reaches node 2. Node 2 is added to \( V_f \), and edge \( (1, 2) \) is added to \( E_f \). In the next step, edge \( (2, 3) \), and \( (2, 4) \) are added to \( E_f \). After the current positions are proceeded, \( r(1, 3) = 1 \), \( r(2, 3) = 3 \), and \( r(2, 4) = 4 \). In this step, no node are reached. The algorithm runs similarly until all shortest paths are determined, \( V_f = \{1, 2, 3, 4\} \).

B. Matrix representation

Network topology can be expressed as a matrix \( A = a_{ij} \) whose size is \( n \times n \). It has row and column corresponding to every node, and its \( i/\text{th} \) entry \( a_{ij} \) equals the cost of the edge
(i, j) if (i, j) ∈ E, otherwise 0. The matrix is called the node-
node adjacency matrix, or simply called the adjacency matrix.
Equation 1 shows the adjacency matrix for the network shown in
Fig. 1. In Fig. 1, the edge between node 1 and node 2 is 1,
so a_{12} and a_{21} is 1.

\[
A = \begin{pmatrix}
0 & 1 & 3 & 0 \\
1 & 0 & 4 & 3 \\
3 & 4 & 0 & 2 \\
0 & 3 & 2 & 0
\end{pmatrix}
\] (1)

MPSA can be expressed as matrix representation. Matrix
representation is space-inefficient, but suitable for hardware
implementation. In addition, matrix operations offer high par-
allelism since the operations of the elements are independent.
For these characteristics, we express MPSA as matrix-based
operations to implement it on DAPDNA-2.

Matrix representation version of MPSA uses five matrices,
called network matrix N, search matrix S, fix matrix F, reach
matrix R, and path matrix P. Here, we assume that an element
is an l-bit unsigned integer data.

The followings are definitions of these matrices. Network
matrix \( N = n_{ij} \) corresponds to \( r(i, j) \). For example, if edge
(i, j) is now traversed and 5 cost units remain to reach node
j, \( n_{ij} = 5 \). Search matrix \( S = s_{ij} \) is a matrix representation
of \( E_r \). If edge (i, j) is not now traversed, \( s_{ij} = 0 \). Otherwise,
\( s_{ij} = 2^l - 1 \). This number is the maximum unsigned integer
value for l-bit data, and all of its bits are 1. In the later, we
denote \( 2^l - 1 \) as MAX_VALUE. Fix matrix \( F = f_{ij} \) expresses
\( V_f \). In the case that node j ∈ \( V_f \), \( f_{ij} = \text{MAX}\_\text{VALUE} \)
for all \( i ∈ E \). On the contrary, \( j \notin V_f \), \( f_{ij} = 0 \) for all \( i ∈ E \).
Reach matrix \( R = r_{ij} \) is a matrix representation of \( V_r \). Like fix
matrix \( F \), \( r_{ij} = \text{MAX}\_\text{VALUE} \) for all \( i ∈ E \) when \( j ∈ V_r \),
and \( r_{ij} = 0 \) for all \( i ∈ E \) when \( j \notin V_r \). Finally, path matrix \( P = p_{ij} \)
corresponds to \( E_f \). If edge (i, j) ∈ \( E_f \), \( p_{ij} = \text{MAX}\_\text{VALUE} \).
Otherwise \( p_{ij} = 0 \).

In addition, we denote the matrix operations as follows
where \( X = x_{ij} \) and \( Y = y_{ij} \) are matrices. And \( Z = z_{ij} \) denotes the result of each operation.

- **TRANS(X)**
  Transpose \( X \), \( z_{ij} = x_{ji} \) for all \( i, j \).

- **OPEA(X)**
  Called Operation A later. If \( x_{ij} ≠ 0 \), \( z_{ij} = \text{MAX}\_\text{VALUE} \).
  Otherwise, \( z_{ij} = 0 \).

- **OPEB(X)**
  Called Operation B later. If \( x_{ij} ≠ 0 \), \( z_{ij} = 1 \).
  Otherwise, \( z_{ij} = 0 \).

- **OPEC(X)**
  Called Operation C later. \( z_{ij} \) is the result of bitwise OR
operation for all element in column \( j \). Figure 2 shows an
example of Operation C.

Using above matrices and operations, MPSA can be ex-
pressed as follows. We assume that \( T_s = t_{sij} \) is a temporary
matrix used in the following description.

1) Set fix matrix \( F \), search matrix \( S \), and path matrix \( P \)
to zero matrix. Reach matrix \( R \) is set to be \( r_{is} = \text{MAX}\_\text{VALUE} \)
for all \( i ∈ E \) where \( s \) is the source
node. Set network matrix \( N \) to the adjacency matrix
representing the network topology and edge weight, i.e.
\( n_{ij} = c(i, j) \) if \( (i, j) ∈ E \), otherwise \( n_{ij} = 0 \).

2) \( F = \text{OR}(F, R) \). If \( f_{ij} = \text{MAX}\_\text{VALUE} \) for all \( i, j \),
complete the algorithm.

3) \( T_1 = \text{AND}(\text{NOT}(R), S) \), \( T_2 = \text{AND}(\text{NOT}(R), N) \). These
operations is to set all elements in column \( i \) to zero
where \( i ∈ V_r \). \( T_3 = \text{AND}(\text{TRANS}(R), T_2) \), and set
\( T_4 = \text{OPEA}(T_3) \). \( T_5 = \text{MAX}\_\text{VALUE} \) if node \( i \) is the
reached nodes. Adding \((i, j)\) to \( V_f \) in 3) of Section III-A
corresponds to \( S = \text{OR}(T_1, T_4) \). Set \( T_5 = \text{NOT}(S) \).

4) \( N = \text{SUB}(T_2, \text{OPEB}(S)) \). If \( n_{ij} \) become 0, it
indicates that node \( j \) is just reached. To pick up
edge \((i, j)\) where node \( j \) is just reached, set \( T_6 = \text{NOT}(\text{OPEA}(\text{OR}(T_5, N))) \). If edge \((i, j)\) is on the shortest
path tree, \( R_{ij} \) is \( \text{MAX}\_\text{VALUE} \). Finally, \( P = \text{OR}(P, T_6) \),
and \( R = \text{OPEC}(T_6) \). Then go to 2).

Figure 3 shows the data-flow of above matrix representa-
tion version of MPSA. MPSA consists of the simple matrix
operations, AND, OR, NOT, SUB, TRANS, OPEA, OPEB,
and OPEC. They are suitable for hardware implementation
because of their simplicity. In addition, their elements are
independent except for TRANS and OPEC. By assigning a
PE to a operation of an element, we can take advantage of
parallel processing of dynamically reconfigurable processors.

IV. EVALUATION

In this section, we evaluate the execution time of MPSA
on a dynamically reconfigurable processor. We assume the
maximum distance in the network is \( d_{\text{max}} \), and the execution
time of the sequence in Fig 3 is \( t_{\text{seq}} \). The sequence shown in
Fig.3 repeats until all shortest paths from node \( s \) is determined,
in other words, all element in $F$ become MAX\_VALUE. Therefore the sequence must be repeated $d_{max} + 1$ times to obtain all shortest paths. The execution time $T_{exe}$ is expressed as follows.

$$T_{exe} = (d_{max} + 1) \times t_{seq}$$

(2)

The above result indicates the execution time is $O(d_{max})$. In square-mesh networks,

$$d_{max} = 2(\sqrt{n} - 1) \times c_{ave}$$

(3)

where $n$ is the number of nodes, and $c_{ave}$ is the average cost for all edges $(i, j) \in E$. From Equation (2) and (3), we obtain

$$T_{exe} = 2(\sqrt{n} - 1) \times c_{ave} \times t_{seq} + t_{seq}.$$  

(4)

Equation (4) indicates the execution time is $O(\sqrt{n})$ in square-mesh networks. On other topologies, $d_{max}$ is at most $O(n)$, so the computational complexity of MPSA is at most $O(n)$. This is lower than that of Dijkstra’s algorithm.

To measure $T_{seq}$, we design the matrix operation units on DAPDNA-2. They corresponds to the operations used in the operation, respectively. In these design, an element in a matrix is 16-bit data, and the size of a matrix is $4 \times 4$. The latency for each matrix operation is shown in Table I. The latencies are between 2 clocks and 4 clocks. We obtain $t_{seq} = 27$ (clocks) from the result of the latencies and Figure 3.

We compare the execution time of Dijkstra’s Algorithm and MPSA. Figure 4 shows the execution time versus the number of nodes where the network topology is square-mesh. We assume the average cost $c_{ave}$ is 3, and the reconfigurable processor has enough PEs and memories. Dijkstra’s algorithm is run on the PC whose processor is a Intel Pentium 4 3.0GHz, and which has 1024MB RAM. The plots of Dijkstra’s algorithm in Fig. 4 is the actual measurement. On the other hand, that of MPSA is calculated from Equation 4 and the assumption that DAPDNA-2 runs at 166 MHz. In the proposed algorithm, the execution time increases slowly as $n$ increases because MPSA on a reconfigurable processor runs at $O(\sqrt{n})$. When $n = 169$, 96.7% fewer time are used than with Dijkstra’s algorithm.

V. HARDWARE OFF\_-LOADING ENGINE PROTOTYPE

Hard-wired logic is a good way to speed up an algorithm, but it has less flexibility than software-based approach. In this paper, we employ an off\_-loading technique which is a hardware and software mixed approach. In this approach, only a heavy task is executed on the hardware specialized to a certain algorithm. It leads to reduction of the total execution time. On the other hand, a light task is executed by software. It leads to having flexibility of software. To speeding up the shortest path calculation of an actual router, we make a prototype of a hardware off\_-loading engine working together with GNU Zebra software router. The off\_-loading engine is made on IPFlex DAPDNA-2 using the matrix-based operations shown in Section III-B. Our implementation is done on the evaluation board, DAPDNA-EB4 shown in Fig 5.

A. MPSA IMPLEMENTATION ON DAPDNA-2

In the implementation, we decide the size of an element in a matrix is 16 bits since the bit\_-length of metric in a header of OSPF is 16 bits [3]. One word of DAPDNA-2 is 32-bit length, so two element can be packed in a word. And we determine...
the size of the matrix is $32 \times 32$. It is lead to be able to calculate 32-node network at a time. The data size of the matrix is 2KB ($32 \times 32 \times 2$).

The data of the matrices is loaded from external memory, and input into the shortest path calculation unit. When the data pass through the unit, the operations shown in Fig. 3 are executed. Therefore, the unit runs multiple times until all the shortest paths from the source node are determined. After the output data is stored into external memory. In the implementation, the data of each matrix is input/output in serial. Serial I/O makes required memory bandwidth low, and reduces PE consumption compared to parallel I/O. Especially, reduction of PE consumption is a merit because of easy fitting.

Figure 6 shows the high level function constituting MPSA. The implementation of MPSA consists of three parts: pre-processing, main processing, and post-processing stage. The operation in Fig. 6 corresponds to several matrix operations.

Four operations exist in pre-processing stage as explained below.

- **Update fixed nodes**
  Add newly reached nodes which is determined in the previous processing to the set of fixed nodes.

- **Clear unnecessary remaining costs**
  In order not to add the links towards the reached nodes to the searched nodes, we clear the corresponding values in the network matrix.

- **Exclude unnecessary searched links**
  We exclude the links towards the reached nodes from the set of searched links.

- **Update searched links**
  Add the links from newly reached nodes to the set of searched links.

In main processing stage, we executes the operations shown in Fig. 3. Finally, post-processing stage has two operations as explained below.

- **Update the shortest path tree**
  The links on the shortest path tree are determined in main processing stage. Add the links to the shortest path tree which is expressed as path matrix.

- **Detect the nodes which is just reached**
  The newly reached nodes in main processing stage is detected in this operation. The reached nodes are used in next pre-processing stage.

The implementation employes reconfiguration to reduce usage of PEs. As shown in Figure 6, all the functions constituting MPSA is splitting three configurations. Configuration 1 includes loading data and pre-processing stage except updating searched edges. Configuration 2 includes updating searched edges, main processing stage, updating the shortest path tree in post-processing stage, and storing the data. It is main configuration in our implementation. Configuration 3 includes detecting the nodes which is just reached and storing the part of the data. We split into three configurations at the point before the operation TRANS and OPEC. This is because there are dependencies in TRANS and OPEC.

Figure 7 shows the flowchart of reconfigurations. Configuration 1 to 3 run until all the shortest paths are determined, so they run $d_{\text{max}} + 1$ times.

B. Integration with GNU Zebra

We integrate our hardware off-loading engine explained in previous subsection with GNU Zebra. GNU Zebra is a famous software-based router which runs on UNIX-like operating system, for example Linux. Zebra works as a daemon, and it processes many routing protocol such as RIP, and OSPF, etc. To be able to off-load the calculation, we modify Zebra version 0.94’s source code. We add the off-loading architecture in ospfd which is a daemon processing OSPF included in Zebra.

Fig 8 shows the architecture of our DAPDNA-2 based hardware off-loading engine integrated with Zebra. As operating system, we employ RedHat Enterprise Linux 4 which is running on a commodity PC. The PC has Intel Pentium 4 3.0GHz and 1024MB RAM. MPSA is implemented on DAPDNA-2 and evaluation board DAPDNA-EB4 is used. DAPDNA-
EB4 is Full-size PCI board and plugged into a PCI slot of the PC as shown in 9. The device driver makes DAPDNA-EB4 work on RedHat Enterprise Linux. The modified ospfd usually collects link-state information, and it triggers the shortest path calculation on DAPDNA-2 when re-calculation is required. Before executing MPSA, link-state informations are transformed to the matrix-based data format as show used in MPSA. Network matrix, Reach matrix, Fix matrix, Search matrix, and Path matrix are initialized and set to RAM on DAPDNA-EB4. The result is represented as matrix-based format, so ospfd transforms the result to Zebra’s internal data structure and makes a routing table.

VI. Conclusion

In this paper, we implement a prototype of a hardware off-loading engine for speeding up the shortest path calculation working together with GNU Zebra. In the next generation networks, TE with many functionalities is required to utilize network resources efficiently. Under this situation, the shortest path calculation is frequently occurred and become a heavy task. We proposed a new approach to calculate the shortest path. It is a parallel shortest path algorithm running on a reconfigurable processor. To achieve fast shortest path calculation, in this paper, we have proposed parallel searching method named MPSA suitable for parallel data-flow machines. We can take advantage of parallel processing by using MPSA since it can be expressed as simple matrix operations. In addition, the implementation of MPSA on DAPDNA-2 is integrated with a software-based router, GNU Zebra. We show that the proposed algorithm requires far less execution time than Dijkstra’s algorithm theoretically.

Acknowledgment

The authors thank to Tomomi Sato, CTO, IPFlex Inc., and other staff of IPFlex Inc. This work is supported in part by a Grant-in-Aid for the Global Center of Excellence for High-Level Global Cooperation for Leading-Edge Platform on Access Spaces from the Ministry of Education, Culture, Sport, Science, and Technology in Japan. One of the authors, Sho Shimizu, is grateful for Yoshida Scholarship Foundation for its financial support.

References