Development and Synchronous FPGA Implementation of a Parallel Accelerated Propagation-Delay Algorithm for Shortest-Path Problem

Intel Science Talent Search
(Computer Science)
November 2009

Jacob Cole

Torrey Pines High School
San Diego, CA 92130

Abstract

This paper describes the development, optimization, simulation, and practical FPGA (Xilinx Spartan-3E X3S500E) implementation of a new parallel algorithm for the NSSP (single source shortest path problem with nonnegative edge weights). Its run time has an upper bound of O(*min*(n, ε)), and it uses hardware resources on the order of O(m), the theoretical optimum. It was applied to standard benchmark problem instances and its performance was compared to that of the fastest general case implementation of Dijkstra's algorithm – O($m + n \log n$). For practical instances of the problem, the propagation delay algorithm required on the order of 200-300 times fewer clock cycles. The hardware implementation achieved is fully scalable, and the paper proposes a second-generation chip architecture which, when implemented, will make the device efficiently problem-reconfigurable in real-time. The relatively low cost of the chip combined with its power and flexibility make it broadly applicable in a wide variety of laboratory and field situations. Moreover, the underlying algorithm is the product of a new parallel computing paradigm, which will be termed "accelerated propagation delay" because it is based on controlling and recording the relative speed of signals propagating in parallel through a network. This paradigm is generalizable to solve other problems that are even more computationally intensive than pathfinding, including the NP-complete subset sum and Hamiltonian path problems. An accelerated design to solve the first of these problems was developed as part of this project and is proposed briefly in the discussion.

1 Introduction

1.1 Overview

Finding the shortest paths between points on a graph is one of the most fundamental and widely applicable optimization problems in computer science. Directly or as subproblems, shortest path problems are critically important in fields ranging from network routing to autonomous system control to computational biology to embedded system design [1], [2], [3]. Frequently, the speed with which shortest path problems can be resolved limits the rate of much larger operations.

Previous study of the shortest path problem has largely been limited to sequential algorithms and, more recently, parallel algorithms implemented on general-purpose architectures [4], [1]. However, the former are restricted by an upper bound of O($m + nD$)[1] [5], and attempts at the latter either are only applicable to specific instances of the problem, or parallelism degrades as problem size increases. Furthermore, fully fledged CPUs are bulky and expensive – this renders large-scale parallelism impractical for many applications.

The introduction of FPGAs (Field-Programmable Gate Arrays, See Section 1.2) [6] now provides a platform for low-cost, massively parallel processing. Recently, a number of VLSI (Very Large Scale Integration) solutions to the shortest path have been proposed, but their complexity (time and/or space) is inferior to that of algorithms implemented on general-purpose architectures. Moreover, their actual performance against standard benchmarks has not been not extensively studied [2].

[1] Here, n is $|V|$, the number of vertices (nodes) in the graph, m is $|E|$, the number of edges (weighted arcs connecting nodes), and D is the additional complexity the data structure backing the algorithm incurs.

This paper describes the development, optimization, simulation, and practical FPGA (Xilinx Spartan-3E X3S500E) implementation of a new, massively parallel algorithm to solve the NSSP (single source shortest path problem with nonnegative edge weights). Its run time has an upper bound of O($min(n, \varepsilon)$),[2] and it uses hardware resources on the order of O(m), the theoretical optimum. It was applied to standard benchmark problem instances [5] and its performance was compared to that of the fastest general case implementation of Dijkstra's algorithm – O($m + n \log n$) [7]. For practical instances of the problem, the propagation delay algorithm required on the order of hundreds of times fewer clock cycles. The hardware implementation achieved is fully scalable, and this paper details a second-generation chip architecture which, when implemented, will make the device efficiently problem-reconfigurable in real-time. The relatively low cost of the chip combined with its power and flexibility make it broadly applicable in a wide variety of laboratory and field situations.

Moreover, the underlying algorithm is the product of a new parallel computing paradigm, which will be termed "accelerated propagation delay" because it is based on controlling and recording the relative speed of signals propagating in parallel through a network. This paradigm is generalizable to solve other problems that are even more computationally intensive than pathfinding, including the NP-complete subset sum and Hamiltonian path problems [10]. An accelerated design to solve the first of these problems was developed as part of this project and is proposed briefly in the discussion.

[2] ε is the eccentricity of the source node in the single source shortest path problem. See Section 1.3

1.2 Field-Programmable Gate Arrays (FPGAs)

Field-programmable gate arrays, or FPGAs, are reconfigurable, programmable logic devices that easily allow for parallel processing. They consist of an array of configurable logic blocks (CLBs) wired together via a set of programmable interconnect modules. By making use of the fact that all combinational logic can be represented as a Boolean sum of products, these devices achieve hardware-efficiency even while retaining the ability to be programmed to represent nearly any synthesizable digital microcircuit [6].

1.3 Shortest Path Problems: Formal Definitions

There are many classes of shortest path problems. Although the algorithm described in this paper can be generalized to solve other instances, this paper focuses on its performance in resolving the single source shortest path problem with non-negative edge weights, or NSSP. The NSSP is easily understood in terms of the single *pair* shortest path problem (SPSP), the general problem of finding a path from one node of a graph to another such that the sum of the weights of its constituent edges is minimal. Rigorously, given a weighted, directed graph $G = (V, E)$ (that is, a set V of vertices and a set E of edges), a real-valued edge weight function $\ell : E \rightarrow \mathbf{R}$, and one vertex $s \in V$, the SPSP seeks a path P from s to some vertex $v \in V$ such that the following expression ("length" of path P) is minimized: $\sum_{p \in P} \ell(p)$

The NSSP extends this: it seeks the shortest paths between a source node s and *every* vertex v of the graph. The solution is typically represented with a shortest paths tree (SPT). The SPT of G is defined to be a spanning tree rooted at s such that the reversal of any path v to s is a shortest path from s to v. Also, NSSP restricts the range of ℓ to nonnegative real numbers [7].

1.4 Prior Research

Of the many sequential techniques to solve the NSSP, Dijkstra's algorithm [7] provides the best general-case performance (amortized run time of O($m + n \log n$) when implemented with a Fibonacci heap) [4], [5]. Hence, it is currently the standard algorithm employed to solve shortest path problems that arise in many fields (very notably, it is employed by the OSPF network routing protocol) [1]. As a result of its ubiquity, many attempts have been made to outperform it.

Recently, various parallel approaches to the NSSP have been proposed [1], [2], [9] and some were demonstrated to perform better than Dijkstra's algorithm. This paper describes the development of one such method, based on the paradigm of propagation delay. Although the paradigm and method were created independently in the course of this project in 2009, subsequent literature search revealed that other investigators, apparently unknown to one another, hit upon certain aspects of the same approach between 2006 and 2008.

Prasad et al. described NATR, an FPGA-based shortest path solution [9]. The algorithm they derived is logically similar to the non-accelerated version of the propagation delay algorithm presented below, so in most cases it requires significantly more clock cycles than the accelerated version of this algorithm (performance is O(ε) vs. O($min(n, \varepsilon)$) for NSSP).

Ishikawa et al. [1] arrived at an algorithm that is, in the abstract, equivalent to the accelerated propagation delay technique put forth in Section 2.3, but they implemented it in a different form on a general purpose parallel DAPDNA-2 processor instead of an FPGA. Their logic required the generation of an n by n matrix circuit, and their high-level hardware platform had only had a limited number of processing elements (376), so they had to divide the problem into segments then recombine the results. Therefore, their implementation was less hardware-efficient (O(n^2) as opposed to O(m)), less scalable, and more expensive.

Oltean et al. recognized the general concept that many difficult computational problems can be solved by analyzing delayed signals. They employed it to build devices that used delays in propagation of light through fiber optics to solve certain NP-complete problems (including subset sum, Hamiltonian path, and exact cover) [10]. These implementations have considerable practical limitations however (e.g., difficult to reconfigure, no acceleration possible).

2 Parallel, Propagation-Delay Algorithm for Shortest Path

2.1 Propagation Delay

The general paradigm of propagation delay considers a graph as a network through which a signal may permeate, starting at a source node (or nodes in some applications). The signal's propagation through each edge of the graph is delayed by an amount proportional to the edge's weight. In outline form, the paradigm is: (1) Express the problem (in the abstract) as a weighted graph. (2) Start a signal at a particular node (or multiple nodes for some applications), and allow it to propagate through nodes making some record of signal arrivals at each node from each edge. (3) Analyze recorded arrival information to solve the problem. Frequently, it is necessary to trace the progression of the signal backwards through the graph to find the answer.

Details of the structure of each individual problem can be used to optimize performance. In general, since delayed signals are not sequentially dependent upon one another and are easily modeled in straightforward hardware (either synchronously or asynchronously[3]), algorithms derived from this paradigm are extremely amenable to massively parallel execution.

2.2 Naïve Algorithm for NSSP

Direct application of the propagation delay paradigm to the NSSP yields a parallel algorithm that executes rapidly for graphs with small, integer edge weights. While far from optimal, the naïve algorithm is straightforward to understand and forms the basis for the accelerated algorithm.

To comprehend its behavior, consider the following example (Fig. 1). In the sample graph[4], *A* has been chosen as the source vertex. The *signal* is said to begin at *A*. A node that the signal has reached will be considered *active*. All edges departing from an active node will also be considered active. Two numbers correspond to each edge $e \in E$. The first represents the value of its *waiting function* $w(e)$, which reflects the position of the signal along the edge, and the second represents its assigned weight, or length, $\ell(e)$. Initially, before any cycles of the clock, $w(e) = \ell(e)$. On each clock cycle, the value of $w(e)$ for each active edge decrements by 1. This decrease corresponds to the propagation of the signal along the edge. When the value of the waiting function of a given edge reaches 0, the signal propagating along it can be said to have reached the next node (at this point, it ceases counting down). That target node then becomes active, and the waiting functions of the edges departing it begin to decrement. Thus, the signal eventually will reach every node of a connected graph.

[3] A synchronous circuit is orchestrated by universal clock; an asynchronous circuit either has no clock or is divided into independent, autonomous components [8].
[4] An undirected graph is shown for simplicity. The actual algorithm uses symmetric pairs of directed edges to simulate undirectedness.

When a node *v* is first activated by an edge, the identifier of the node at the originating end of that edge is logged by *v* as its `PREV` (the node preceding it in the final SPT).[5] Once every node has been reached, the list of `PREV` values for the entire graph is outputted. The list constitutes an SPT – any shortest path from *s* can be identified by following `PREV` values backwards from a given vertex to *s*.

Since every clock cycle brings a signal one unit farther away from the source node, and this algorithm solves the NSSP when every node in the graph is reached, the run time complexity of this algorithm is O(ε), where ε is the (weighted) eccentricity[6] of the source node – by the time the node farthest from the source has been reached, all other nodes necessarily have been. Hence, for graphs with small edge weights, this algorithm is acceptable (even rapid), though it balks at larger cases.

The memory (and as will be shown, hardware) footprint of this algorithm is extremely small. O(1) values must be registered for each of the *m* edges in the graph, and O(1) values must be registered for each of the *n* nodes, so it appears that the hardware/memory usage is on the order of O($n + m$). However, since to be considered at all, a node must be connected to at least one edge, the *m* term can be seen as subsuming the *n* without loss of generality. Thus the complexity can actually be considered to be O(m). This is the theoretical minimum resource usage to resolve the shortest path problem: no more data must be stored than that which is required to represent the problem and solution.

[5] Note: If multiple signals arrive at a node at the same time, a single `PREV` value is arbitrarily chosen from among them. Also, since `PREV` is never overwritten once set, edges incident to a node that has already been reached are deactivated – the arrival of a later signal from them would do nothing; only the first signal is logged.

[6] The *eccentricity* of a vertex *v* is defined to be the "maximum graph distance between *v* and any other vertex *u*" [http://mathworld.wolfram.com/GraphEccentricity.html]

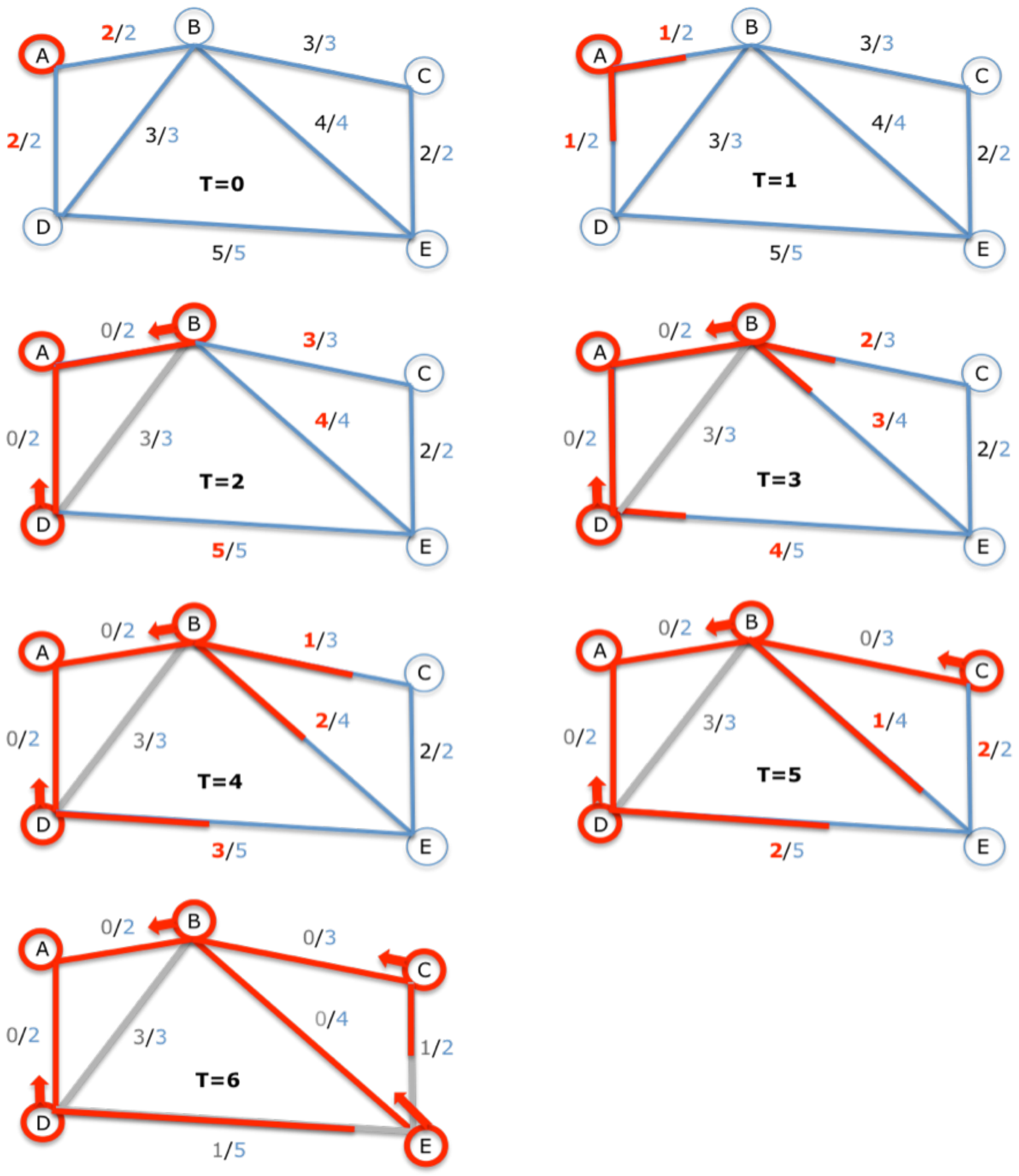

Fig. 1: **Naïve NSSP solution by propagation delay in a 5-node graph**. The source node is *A*. Initial state and states over the course of 6 clock cycles are shown (elapsed "ticks" – clock cycles – denoted by *T*). Active nodes are depicted as red. The position of the signal in the graph is depicted by the red lines extending along the edges, and the value of the waiting function *w*(*e*) is shown by the numerators in the fractions beside the edges. The denominators reflect $\ell(e)$ – edge weights. Edges that are gray have been deactivated because their destination has already been reached from another edge. The arrows extending from nodes that have been activated graphically depict `PREV` – following them backwards as if they constituted a vector field will trace the shortest path to *s* from any node. By *T*=6, the set of arrows constitutes an SPT.

2.3 Accelerated Algorithm

For many problems with large edge weights, the naïve algorithm is clearly impractical due to its O(ε) run time. Fortunately, by re-examining the nature of the signal delays in the graph, performance can be improved by an order of magnitude. Conceptually, the accelerated algorithm possesses only one difference from the naïve: on each tick of the clock, $w(e)$ for all active edges is not simply decremented by 1, but by the *maximum number possible*. This maximum number, which will be used as the weight decrement, is equal to the minimum, nonzero value of w from among all active edges of the graph. Thus, the algorithm behaves as if time has skipped forward

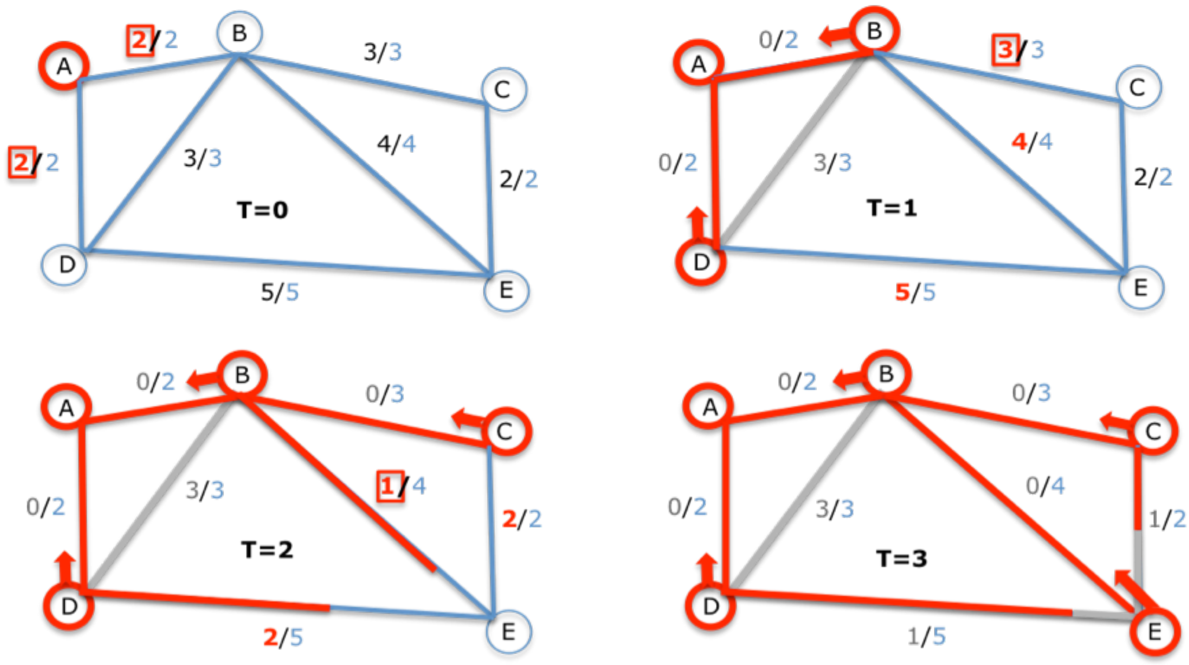

Fig. 2: **Accelerated NSSP solution by propagation delay in a 5-node graph**. Graph is identical to that in Fig. 1. Again, s = A. The minimum waiting function value(s) from among active, nonzero edges is boxed. This number constitutes the decrement for the waiting function. Acceleration solves this instance of the problem in three clock cycles instead of six.

to the next "activation event." This guarantees that on each clock cycle, at least one edge (the active edge of minimal w) will signal its destination node. Since the problem is solved when n

nodes have been activated (at this point, all `PREV` values will have been assigned), the accelerated algorithm has an additional upper bound on its run time of O(n).[7] This makes its overall worst-case[8] run time O($min(n, \varepsilon)$), where the function *min* returns the lesser of its two arguments. Furthermore, when implemented properly (see Section 4), its memory/hardware usage is, like the naïve algorithm, on the optimal order of O(m) – no additional data must be stored. The accelerated algorithm is illustrated in Fig. 2.

3 FPGA Architecture: Accelerated Propagation Delay Algorithm

3.1 Static Architecture

It is relatively straightforward to configure an FPGA to apply the accelerated propagation delay algorithm to a specific graph. In circumstances where on-the-fly reconfigurability of the graph's topology is not necessary, the static architecture described in Fig. 3 is practical and efficient.

In cases where multiple edges activate a node at the same time, the previous node identifier from any edge would give a valid SPT, but only one is needed. A state machine within each node is employed to select a single one. As soon as a node is activated, the state machine, executing in parallel with the rest of the program, begins sequentially searching through the node's incident edges to find a single one that has reached zero. PREV then assumes the value of the identifier passed in through that edge. Since the state machine executes in parallel with the rest of the device, it does not negatively impact run time complexity. It is of O(1) size and does

[7] The minimum edge weight can be found in O(1) with appropriate FPGA architecture. See Section 3.

[8] If more than one node is activated on each tick of time, as is frequently the case when edge weights are small or when the graph is not sparse, run times can be far lower. See Section 4: Performance.

not generate significant clock skew as a daisy chain of combinational comparators would. This is a practical improvement over Prasad [9].

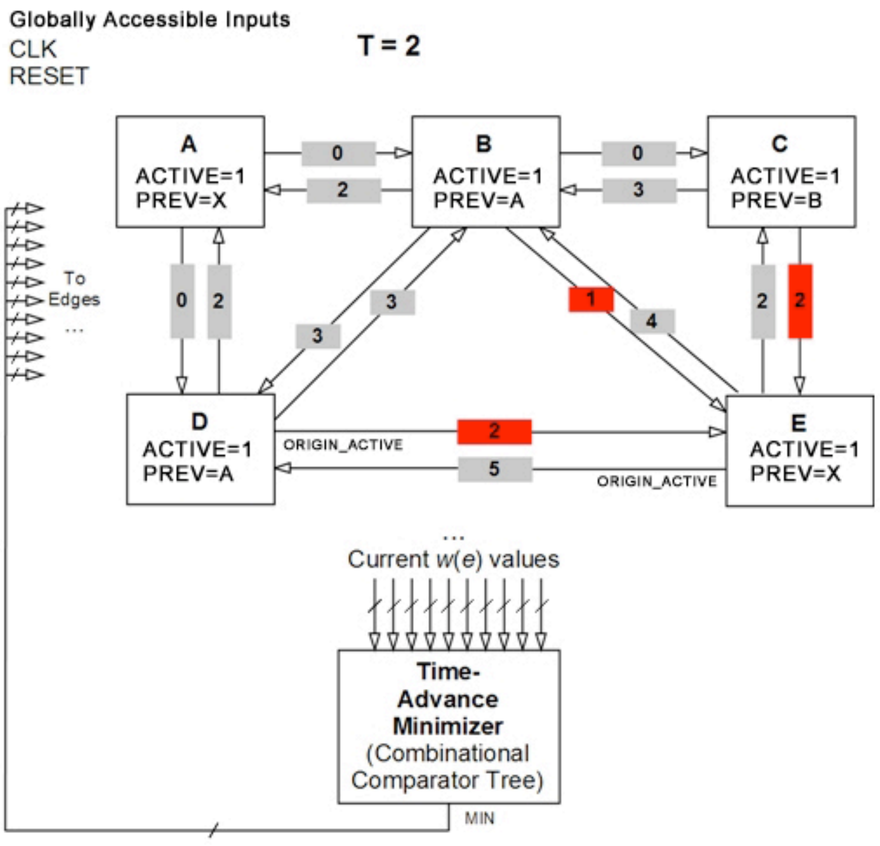

Fig. 3: **Static Architecture for Accelerated NSSP Solver for a 5-Node Graph (simplified)**. Graph represented by above block diagram is same as that in Fig. 1 and Fig. 2. Again, *s* = A. State *T*=2 from Fig. 2 is depicted. FPGA configuration is directly isomorphic to graph represented. Node modules (labeled A, B, C...) are connected by edge modules, the shaded boxes embedded within the arrows between nodes. Edge modules possess a register containing *w*(*e*) for each edge. (The initial value of each *w*(*e*) is ℓ (*e*).) An input wire on each edge (ORIGIN_ACTIVE – above labeled only for DE and ED) reads the ACTIVE register of the node from which it originated. Another wire (TARGET_ACTIVE), not shown, connects the ACTIVE value of each node to each incident edge. Whether an edge will decrement is determined by the value of the expression ORIGIN_ACTIVE && !TARGET_ACTIVE. Above, edges that will decrement on the next tick are depicted in red, those that will not are gray. The current *w* decrement for active edges (minimum value from among all red nodes) is calculated in real-time by the combinational, comparator tree-based Time-Advance Minimizer module (see Fig. 4).

Additionally, each node, *v*, possesses a multi-bit register PREV. As discussed above, it holds the identifier of the node that activated *v* (X represents a "don't care" state). When an edge's *w* value reaches 0, a bus transmits the value of PREV as well as a signal indicating that its down-counting has completed to its destination node.

3.2 Dynamically Reconfigurable Architecture

An efficient FPGA architecture that can be reconfigured at runtime to solve problems in graphs of differing topologies is presented in simplified form below.

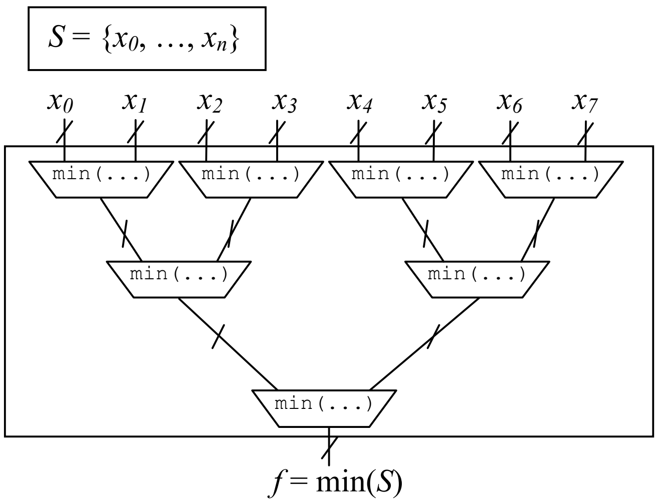

Fig 4: **Architecture for Time-Advance Minimizer Module**. Uses a combinational logic binary comparator tree to find minimum element in an input set (in this application, minimizes $w(e)$). n input buses map to 1 output bus. $n = 8$ shown. Gate count is on the order of $O(n)$ and depth (and hence skew incurred) is $O(\log n)$. However, in terms of clock cycles, run time is effectively $O(1)$.

The top level architecture consists of a fully interconnected set of modules that represent nodes. Nodes internally handle the task of representing the edges departing from them. To ensure that the architecture can represent any possible graph, all nodes have the potential to activate one another. This wiring can be done efficiently on an FPGA because its programmable interconnects are natively structured to allow for all possible connections. As depicted in Fig. 5, for every possible pair of nodes (x, y) a line from the output bus ACT_NODES of x feeds into the ACTIVATE input bus of y.

The Time Advanced Minimizer module computes the "accelerated time" step size (weight decrement), which is used internally by the node modules. It uses a comparator tree as in Fig. 4 to calculate the minimum TIME_OUT value (the minimum w value) from among all nodes and sends the number to each node via the TIME_IN bus. Since, in this architecture, nodes are internally aware of the current minimum w value from among all their departing edges (this is outputted via TIME_OUT – see below), only n values must be compared by the tree. This reduces tree depth, and hence clock skew, to O(log *n*), allowing for faster maximum clock frequency than in the static design.

Within the Node

The edges departing a given node *n* are represented within the node module (Fig. 6). Their weights and destinations are stored in a FIFO (First-In, First-Out) data structure, ordered as a min priority queue keyed by edge weight ℓ (thus, the smallest $\ell(e)$ from among edges departing a given node is listed first). The value corresponding to each edge weight key is the list of node identifiers representing the destinations of the edges departing *n* that have a weight of ℓ.[9] Thus, all edges of a given weight are listed beneath the key corresponding to that weight in the FIFO. This data structure is populated at initialization or on-the-fly via the input bus INIT_CONF. This representation requires one unit of FPGA resources per edge; hence, the hardware usage of the architecture is still on the order of O(*m*).

When an activate signal enters a given node *n* (i.e. if any line in the ACTIVATE bus is pulled high), an internal (Mealy) state machine registers the event. It initiates a minor state machine to find PREV, as in the static architecture. It also enables the local time accumulator, which, on every subsequent clock cycle increments its current value by the current "accelerated time" step size (the weight decrement), which is received from the TIME_IN input bus.

The local time accumulator is wired to the negative input of a subtractor module. The positive input of the subtractor module reads the current weight of the item at the head of the FIFO. Thus, the output of the subtractor (which is fed to the state machine and to TIME_OUT) is equal to the minimum *w* value from among edges departing *n*. This means that the event of the

[9] For graphs with a high ratio of edges to nodes, it is more efficient to simply represent this list as a bitset.

subtractor's output reaching 0 corresponds to the event of a signal's arrival at some other node, brought about by way of an edge originating at *n*.

This zeroing event is detected by the node state machine, which signals the FIFO to pop the head element of the queue and also sends a signal through the output bus ACT_NODES to bring about the activation of the destination nodes enumerated in the value portion of that head element. The new head element of the FIFO represents the edges of next smallest ℓ (weight) value – the edges whose waiting functions will next reach zero.

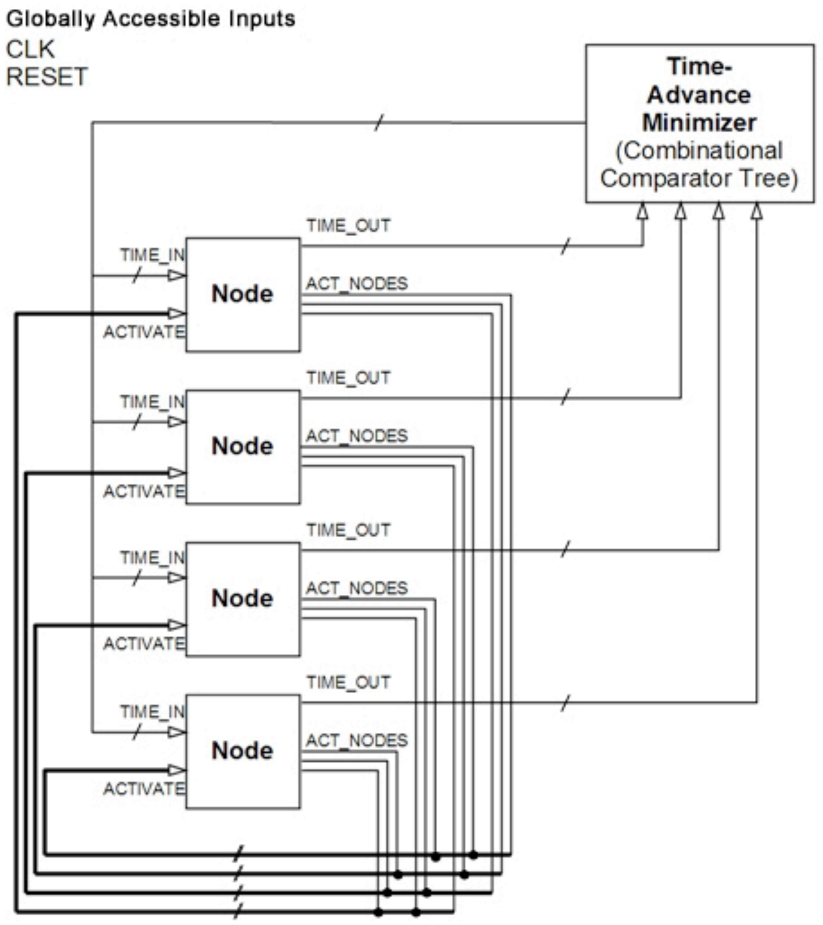
Globally Accessible Inputs
CLK
RESET
Time-Advance Minimizer (Combinational Comparator Tree)
TIME_OUT
TIME_IN
Node
ACT_NODES
ACTIVATE

Fig. 5: **Dynamically Reconfigurable Top-Level Architecture (simplified)**. Instance above holds 4 nodes. ACT_NODES buses are depicted as expanded to reveal connections. ACTIVATE buses keep signals from different input nodes separate – this is necessary when finding `PREV` values.

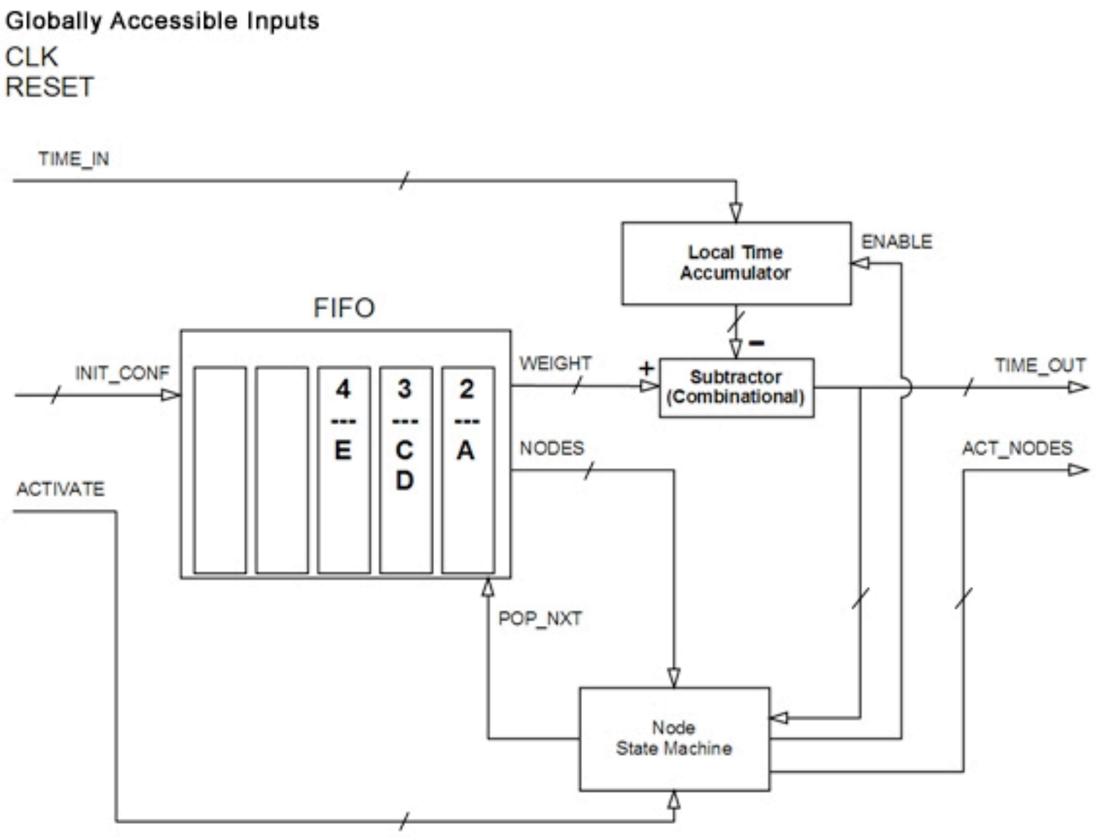

Fig. 6: **Dynamically Reconfigurable Architecture**: **Node Block Diagram (simplified)**. Above image diagram depicts initial state *T*=0 (after graph has been loaded) of node *B* from same graph in Fig. 1, 2, and 3. CLK is the clock input signal.

The dynamically reconfigurable architecture can be scaled to the necessary size simply by adding and wiring in more node modules. The design also enables efficient hardware reuse. Since many graphs do not have every node connected to every node [5] (e.g roadmap modeling [2]), few, if any nodes require a FIFO of length *n*. Thus, resource usage can be improved by limiting the lengths of a certain subset of the FIFOs. The extra resources can be used to enable the device to support graphs with greater numbers of nodes.

4 Testing and Results: Performance

As proof of concept, the static architecture was coded in Verilog, simulated and verified in the behavioral simulation program ModelSim, then implemented on a Xilinx Spartan-3E X3S500E. Test data for numerous small cases of the NSSP (fewer than 30 nodes) were run and output to a custom C++ program via a serial line using the RS-232 protocol. It was demonstrated that the results were accurate, and that the number of clock cycles required matched those predicted in theory and simulation.

To determine the speed-up afforded by the parallel propagation delay algorithm as implemented on the FPGA, an isomorphic Java-based simulation was developed and applied to the standard Random4-n[10] class of test cases [5]. The NSSP SPT was computed from 10 randomly selected source nodes and the run time results were averaged. Performance was compared with that of a standard, fast, Fibonacci heap implementation of Dijkstra's algorithm. To perform a fair comparison, a Java implementation of Dijkstra's algorithm was created that counted a virtual "clock cycle" for every group of O(1) operations.[11] The Fibonacci heap used was implemented per Cormen et al. [11] to have O(1) decrease_key and O(log n) amortized remove_min.

Results of the simulation are shown in Table 1. The propagation delay algorithm solved the NSSP on all graphs using on the order of 200-300 times fewer clock cycles. As n increased, run time for the propagation delay algorithm increased approximately linearly, as predicted, whereas Dijkstra's algorithm increased at a rate of approximately O($m + n \log n$), also as

[10] Random4-n uses random graphs with $m = 4n$. Range of arc lengths is [0, . . . ,n]. Values of n used were 2^i for $i = \{10, . . . , 17\}$.

predicted. Thus, the more nodes present in the graph, the greater the advantage of the propagation delay algorithm over Dijkstra's.

Table 1: **Propagation Delay vs. Dijkstra's Algorithm: Run Time Comparison on Random4-n Benchmark Cases**

Nodes (*n*)	Edges (*m*)	Clock Cycles		
n	m	Dijkstra*	Prop Delay†	Ratio
1,024	4,096	159,296	750	212.39
2,048	8,192	353,580	1,493	236.83
4,096	16,384	779,182	2,998	259.90
8,192	32,768	1,699,358	6,025	282.05
16,384	65,536	3,681,307	12,028	306.06
32,768	131,072	7,930,726	24,079	329.36
65,536	262,144	16,985,653	48,282	351.80
131,072	524,288	36,226,452	97,825	370.32

*Estimated Dijkstra Clock Cycles
†Actual Propagation Delay Clock Cycles

5 Discussion

This project demonstrated the theoretical viability and practical utility of the new parallel computing paradigm of accelerated propagation delay by applying it to create an algorithm to solve the NSSP shortest path problem in O($min(n, \varepsilon)$) time with O(m) resources, and implementing it in a practical, scalable form on an FPGA. It also described in detail how to extend the architecture to make the device efficiently reconfigurable on-the-fly.

This work has both near and long-term consequences. It is immediately applicable to a host of common computational problems – processors such as this could be employed in devices including those for gene analysis, missile interception, wheelchair navigation, network routing, embedded system synthesis, and more [2], [3], [12]. Furthermore, the paradigm is readily

[11] It is not possible to directly compare the performance of algorithms implemented on different platforms since differing languages and processors may take varying amounts of time to perform the same operations.

generalizable to accelerate the resolution of certain NP-complete problems important in routing, cryptography, compiler design, and other fields [13].

Among near-term solutions: shortest path problems in which each node has many connected edges (such as those which appear in synthesizing complex microcircuitry) can be solved particularly rapidly by the propagation delay approach since its complexity is not dependent upon *m*. In these situations, the observed 200-300 times speed-up ratio over the fastest sequential algorithms, whose performance degrades linearly as *m* increases, could be significantly improved.

If a graph is too large to fit on the device, the propagation delay approach allows it be broken into independent subgraphs to be solved separately, then recombined using the method of Ishikawa et al. [1]. This is particularly applicable to network routing because relatively isolated subnetworks – which can be recombined into the main graph – naturally arise in computer network topology.

Furthermore, the reconfigurable architecture is readily implementable on a dedicated, high-speed application-specific integrated circuit (ASIC), because a fixed configuration can load graphs of any topology and the design decreases clock skew to O(log *n*). This makes it practical to implement on devices with higher clock speeds than an FPGA.

The propagation delay paradigm's long-term implications began to reveal themselves even during the development of the solution to the shortest path problem. It became apparent that the same type of "accelerated delay" network, with minor modifications, could be used to rapidly solve the NP-complete subset sum problem. The subset sum problem asks, in essence: "Given a set *S* of positive integers and a target number *T*, does there exist a subset of *S* that sums to *T*?"

Conceptually, the main modifications that have to be made are 1) allow nodes to become activated multiple times (by every incoming signal), 2) implement edges in such a way that they are able to queue and delay *all* incident signals, and 3) record all signal arrival times at nodes. Any signal that arrives at a dedicated target node after *T* units of delay have elapsed can be traced backward through the graph to its origin – the set of edge weights thus traversed is equivalent to the set of integers that solves the problem. (For further information, see, Oltean et al. [10], who independently developed a part of this solution – for the yes/no decision problem only – and implemented it in a non-accelerable optical network to solve the subset sum problem and other, even more difficult NP-complete problems.)

Whether or not this paradigm ultimately proves widely usable in practical situations, it offers inroads and insights into the nature of the computable universe.

References

[1] H. Ishikawa, S. Shimuzu, Y. Arakawa, N. Yamanaka, and K. Shiba. (2007). New Parallel Shortest Path Searching Algorithm based on Dynamically Reconfigurable Processor DAPDNA-2. *Proc. IEEE.* [Online]. Available: http://biblio.yamanaka.ics.keio.ac.jp/file/S05S05P02.pdf

[2] T. K. Priya, P. R. Kumar, and K. Sridharan. (2006, Nov.). A hardware-efficient scheme and FPGA realization for computation of single pair shortest path for a mobile automaton. *Microprocessors and Microsystems*. [Online]. 30(7), pp. 413-24. Available: http://linkinghub.elsevier.com/retrieve/pii/S0141933106000548

[3] J. Gebert, M. Laetsch, E. Ming Poh Quek, and G. W. Weber. (2004). "Analyzing and Optimizing Genetic Network Structure via Path-Finding." *Journal of Computational Technologies*. [Online)]. Available: http://144.122.137.13/iam/images/a/a3/Preprint7.pdf

[4] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak, ``Parallel Shortest Path Algorithms for Solving Large-Scale Instances," *9th DIMACS Implementation Challenge -- The Shortest Path Problem*, DIMACS Center, Rutgers University, Piscataway, NJ, Nov. 13-14, 2006.

[5] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. (2006 Oct. 29). 9th DIMACS Implementation Challenge: Core Problem Families. Available: http://www.dis.uniroma1.it/~challenge9/download.shtml

[6] A. Chandrakasan. (2008, Spring). L12: Reconfigurable Logic Architectures. *6.111 Introductory Digital Systems Laboratory* [Online]. Available: http://web.mit.edu/6.111/www/s2008/LECTURES/l12.pdf

[7] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, “Shortest paths algorithms: Theory and experimental evaluation,” *Mathematical Programming*, vol. 73, no. 2, pp. 129–174, May. 1996.

[8] A. Chandrakasan. (2008, Spring). L4: Sequential Building Blocks (Flip-flops, Latches and Registers). *6.111 Introductory Digital Systems Laboratory* [Online]. Available: Available: http://web.mit.edu/6.111/www/s2009/LECTURES/l4.pdf

[9] G. R. Prasad, K. C. Shet (Dr.), and N.B. Bhat (Dr.), “NATR: A New Algorithm for Tracing Routes,” in Innovations and *Advanced Techniques in Systems, Computing Sciences and Software Engineering*, Dordrecht, Netherlands: Springer, 2008, pp. 399–408.

[10] M. Oltean, O. Muntean, “Solving the subset-sum problem with a light-based device,” *Natural Computing*, vol. 8, no. 2, pp. 321-331, 2009.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Ed*. MIT Press and McGraw-Hill, 2001, Chapter 20: Fibonacci Heaps, pp.476–497.

[12] N. Sudha and A.R. Mohan, “Design of a hardware accelerator for path planning on the Euclidean distance transform,” *Journal of Systems Architecture*, 54, (2008) 253–264.

[13] Gilmore PC, Gomory RE (1965) Multistage Cutting Stock Problems of Two and More Dimensions, Operations Research, Vol. 13, No. 1, pp. 94-120