Appears in Int. Symp. on FPGAs

A Representation for Dynamic Graphs in Reconfigurable Hardware and its Application to Fundamental Graph Algorithms

Lorenz Huelsbergen
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974

lorenz@research.bell-labs.com

ABSTRACT

This paper gives a representation for graph data structures as electronic circuits in reconfigurable hardware. Graph properties, such as vertex reachability, are computed quickly by exploiting a graph's edge parallelism—signals propagate along many graph edges concurrently. This new representation admits arbitrary graphs in which vertices/edges may be inserted and deleted dynamically at low cost—graph modification does not entail any re-fitting of the graph's circuit. Dynamic modification is achieved by rewriting cells in a reconfigurable hardware array. Dynamic graph algorithms are given for vertex reachability, transitive closure, shortest unit path, cycle detection, and connected-component identification. On the task of computing a graph's transitive closure, for example, simulation of such a *dynamic graph processor* indicates possible speedups greater than three orders of magnitude compared to an efficient software algorithm running on a contemporaneously fast uniprocessor. Implementation of a prototype in an FPGA verifies the accuracy of the simulation and demonstrates that a practical and efficient (compact) mapping of the graph construction is possible in existing FPGA architectures. In addition to speeding conventional graph computations with dynamic graph processors, we note their potential as parallel graph reducers implementing general (Turing equivalent) computation.

1. INTRODUCTION

Graphs are central data structures; algorithms to manipulate them must be fast. In circuit and hardware design, graph algorithms compute critical route and placement information. Across the computing landscape, graph algorithms deduce connectivity in networks, compute the closure of relations in databases, manage storage in operating and runtime systems—among many other applications. Since graphs naturally correspond to circuits, it seems beneficial to model a specific graph and algorithm as a hardware circuit using a *direct construction* [3; 1]. With direct constructions, however, the costs of compiling (fitting via place and route) a graph to a circuit are prohibitive in practice and sharply offset or nullify most gains in speed. Furthermore, since graphs are often dynamic structures—vertex/edge insertion and deletion occur frequently—their properties continually require recomputation and, with direct constructions, expensive recompilation.

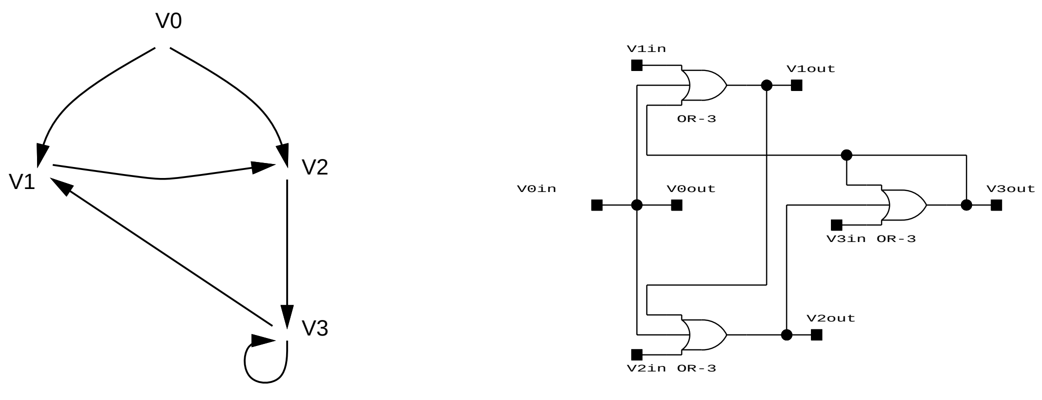
This paper contains a *novel hardware representation* for computing *dynamic graph algorithms*. That is, we describe a graph-to-circuit construction that can store, modify, and compute with arbitrary graphs. Since a graph is embedded in hardware, the size of this resource governs the size of the graphs that may be represented. (Graph representations require space quadratic in the size of the vertex set to hold the edges of an arbitrary graph on that number of vertices.) The representation is dynamic in that new edges and vertices can quickly be inserted and old edges and vertices quickly deleted. We give algorithms that use this construction to compute some fundamental graph properties (see, *e.g.*, [4]): reachability, transitive closure, shortest unit path, connected-component identification and cycle detection.

A graph $G = (V, E)$ maintained as a *graph circuit* is a collection of gates (representing the vertices V) and wires connecting the gates (representing the edges E). Circuit representations of graphs provide fast graph traversal since signals in the circuit may traverse *asynchronously* many graph edges in parallel. Comparable software algorithms—sequential and parallel—can only traverse a single, or small bounded number, of edges simultaneously (see Section 2). In graph circuits, furthermore, edge traversals and vertex visits occur at gate and signal propagation speeds; that is, at a significant fraction of the speed of light. We call a construction that implements dynamic graph algorithms in hardware a *dynamic graph processor* (DGP).

Compiling graphs into specialized hardware circuits in order to attain rapid property computation was initially proposed for integrated circuits [3] and then more directly for reconfigurable hardware arrays [1]. Section 3 contains a direct construction similar to those of the prior work.

This paper's dynamic construction is in sharp contrast to previous hardware [3; 1; 5] and conventional software (*e.g.*, [4]) approaches in which either:

- the hardware graph is completely *static* [3; 1] and re-



(a) (b)

Figure 1: A sample graph G_s with four vertices and six edges (a) along with a *static* circuit for computing reachability in G_s by direct construction (b). This paper gives a new construction for *dynamic* graphs that admits fast graph modification.

quires expensive initial compilation and recompilation when the target graph changes; or,

- large portions of the graph (such as the edge set) are stored in in a conventional memory [4; 5] which curtails utilization of a graph's parallelism, among other things.

In the next section we provide further details of the related work and a comparison of hardware graph algorithms to their conventional software counterparts. We now continue with a high-level overview of DGPs and an illustrative example.

Paramount to DGP feasibility is quick vertex/edge insertion into, or deletion from, the graph's hardware representation. Reconfigurable hardware (RCHW) enables dynamic modification of the graph representation.[1] DGP edge insertion, edge deletion, and vertex insertion are constant time, $O(1)$, operations. Deletion of a vertex v takes time $O(|I_v|)$ where I_v is the set of edges incident on v. (The time complexities count the operations that rewrite the reconfigurable hardware.) The bulk of this paper (Section 4) describes the new representation that enables such lightweight graph operations.

A DGP may be viewed as a form of *graph co-processor*—it mirrors the structure of a graph that is also maintained, but in some other form, on a conventional host processor. Operations on the graph are initiated by the host and reflected by the DGP. Queries of graph properties may be answered by consulting the DGP instead of traversing the host's representation with a software algorithm. After a modification of the graph and hence of the RCHW array, graph queries may again be run on the updated graph without re-fitting the circuit. Bookkeeping tasks—such as maintaining the set of free vertices in the DGP—will, for now, be relegated to the host, but could certainly be merged into a DGP. Host-processor interface overheads may force DGP control logic into (reconfigurable) hardware to make DGPs practical.

The sample graph G_s in Figure 1a serves to illustrate. It has four vertices v_i and six edges and contains cycles. During the course of an algorithm, one may wish to obtain *reachability* information about G_s: for example, is there a path such that vertex V_0 *reaches* ($\leadsto$) vertex V_3? If the graph is viewed as a circuit where edges are wires, a signal on an edge (wire) $V_0 \rightarrow V_1$, for example, will propagate to that edge's target vertex, V_1. At V_1, this signal is then routed onto V_1's out-edges ($V_1 \rightarrow V_2$); continuing in this manner, the reachability $V_0 \leadsto V_3$ is computed. Moreover, upon subsequent edge/vertex insertions and deletions that derive a modified graph G'_s one may want to recompute reachability without explicitly re-fitting G'_s. Section 5 contains—along with DGP algorithms for transitive closure, shortest path, cycle detection and connected component identification—a DGP reachability algorithm that accomplishes this goal.

Simulation of a non-trivial graph property (transitive closure) indicates that large speedups over fast uniprocessor software—exceeding two orders of magnitude—are possible in commercial FPGA cards (*e.g.*, VCC Hotworks [11]). A prototype DGP in a Xilinx XC6216 has been built; it can quickly compute properties of small graphs and also serves to substantiate our simulation parameters. Single contemporary FPGA designs, however, limit graph sizes to a few hundred vertices at best. To handle larger graphs, we hypothesize simple FPGA arrays that require connections only between adjacent boundary cells between any two neighboring FPGAs. Over a range of inter-die communication delays, simulation of transitive closure on such FPGA arrays indicates the potential for extremely large speedups, exceeding a thousand fold. Results of the implementation and simulation are in Section 6.

We sketch a potential future application of DGPs as graph reducers. *Graph reduction* is known to be a model of Turing-equivalent computation. Design of a DGP containing logic for graph reduction at its vertices could therefore perform general computation. In the graph reduction model, a pro-

[1] General reconfigurability—such as afforded by off-the-shelf field-programmable gate arrays (FPGAs)—is not strictly necessary for this approach; PLAs may suffice; custom circuits (ASICs) designed to have small (perhaps bit wide) addressable registers could also be used; Section 6.1.

gram (code and data) is represented by a graph; computation proceeds by rewriting vertices that are currently reducible; this process terminates when no reducible vertices exist. Since multiple reducible vertices may coexist, in a DGP implementation this form of computation extracts a program's parallelism by concurrently rewriting all such vertices in the computation graph. Other future directions and enhancements of DGPs are in Section 7 as well.

2. RELATED WORK AND COMPARISON TO SOFTWARE

In this section we first review the specific work related to converting graph algorithms to hardware circuits. Then we compare such approaches (including ours) to conventional software solutions.

Two prior approaches aim to completely encode graphs in static or reconfigurable hardware [3; 1]. Neither approach has been directly implemented in hardware. Both only address static graphs. That is, a fixed graph is first transformed into a circuit and then fitted to the hardware before queries about it may be answered. Though the resulting circuitry stands to speed the computation, the cost of fitting will most likely negate all such gains. Furthermore, unlike this paper's construction, such approaches cannot easily adapt a circuit once fitted even if its corresponding graph only changes slightly. DGP graphs, on the other hand, are inexpensive to change on the fly due to a general hardware representation (§4) and the partial reconfigurability of some modern FPGAs. Individual details of the two prior static approaches are as follows. Chakradhar and Agrawal [3] described a graph-circuit transformation for computing a graph's independent sets. They use a direct construction (*cf.* §3) to encode the input graph's connectivity as a static circuit. Babb, *et al.* [1] describe graph-circuit transformations for closed semiring problems: shortest path and transitive closure. For example, given an input graph they create VHDL that implements the Bellman-Ford shortest path algorithm. They synthesize and emulate the resultant circuits for FPGA arrays. They do not, however, include the large compilation times (several hours for relatively small graphs; $|V| \leq 512$) that their system requires in the reported speedups.

Dandalis, *et al.* [5] describe a processor that computes a graph's shortest path also using the Bellman-Ford algorithm. In their design, the graph's vertices form a pipeline in the FPGA; edges, along with their weights, are stored conventionally in memory; computation of the algorithm proceeds by feeding the edges through the pipeline. An advantage of this form of graph computation is that a graph's edge set may be quickly modified (by rewriting the edge memory). The authors do not describe how to add or delete vertices from the RCHW pipeline, however. A disadvantage of this approach, compared to this paper's and to [3; 1], is that a graph's edge parallelism is drastically restricted—a vertex v's out-edge set is, in the worst case, not fully processed until all edges have sequentially passed through v. Since a graph on $|V|$ vertices may have up to $|V|$ out-edges at vertex v, this serial computation limits Dandalis, *et al.*'s speedups. In other words, their approach does not exploit the graph–circuit correspondence of the other approaches. Since the Bellman-Ford logic at each vertex is large, they report fitting only four vertices in the 128x128 Xilinx XC6264; in contrast, this chip with our representation can hold graphs approaching 128 vertices (see Section 6.1). Dandalis, *et al.*'s [5] speedups over software implementations are, for the above reasons, small (a factor of four at best).

Perhaps most similar in spirit to the DGPs of this paper is Kean's idea [8] of using reconfigurable logic to discover plausible routes during circuit layout. Kean's scheme in effect solves reachability in a two dimensional mesh of cells given that some cells are missing (*i.e.*, in use elsewhere and hence disabled). Unlike DGPs, his scheme is not inherently graph based, computes cell reachability only in a restricted domain (2D meshes with certain prescribed connectivity), and does not allow for arbitrary insertion/deletion of wires (*i.e.*, edges) between circuit nodes without dismantling the current circuit.

Software algorithms to compute graph properties are well known and described in introductory algorithm texts (*e.g.*, [4]). Breadth-first search, for example, can efficiently compute reachability and shortest unit path in time proportional to the size of the input graph, $O(|V| + |E|)$. This time, however, includes a constant representing the costs of manipulating the graph's software representation (including the non-trivial costs of fetching this representation into the processor). This software constant is large relative to the constants (signal propagation, gate delay) involved in digital circuits in general and graph circuits in particular. Furthermore, uniprocessor software implementations can only traverse a single edge at a time even though the search could naturally proceed through multiple edges—and potentially in multiple sections of the graph—concurrently. Note that compiling a graph to hardware does not necessarily change the asymptotic time complexity of the target algorithm; graphs in hardware do, however, stand to significantly reduce the constant overhead that software representation of the graph entails and can thereby stand to greatly speed the algorithm.

A multiprocessor with parallel software, on the other hand, can explore some number of paths in parallel; this number being bounded by the (contemporarily small) number of parallel processing elements. Additionally, a multiprocessor must furthermore coordinate and synchronize the search, which also incurs expenses. Parallel graph algorithms are known for many processing networks (see, for example, [9; 6]), but require explicit synchronization and communication. Synchronization operations too have relatively large time-complexity constants. The circuits described here operate—for the most part[2]—asynchronously at the (unclocked) gate and wire level. (We alleviate concerns of races due to this asynchrony in the discussion of the implementation, Section 6.1.) DGPs share the rather large hardware resource requirements, $O(|V|^2)$, of graph algorithms implemented on *processor arrays* [9]. Unlike processor arrays, DGPs do not require construction of novel physical hardware since they reside completely in commodity reconfigurable hardware.

3. BACKGROUND: REACHABILITY AND THE DIRECT CONSTRUCTION

First we provide background on graph reachability. Then we describe modeling graphs with a direct construction and elucidate the weaknesses thereof. *This section is independent*

[2]Some DGP algorithms, such as for transitive closure, synchronously invoke asynchronous subroutines (§5).

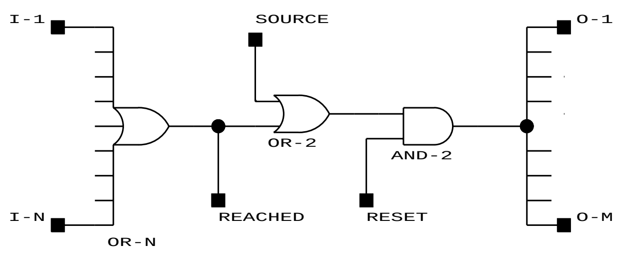


Figure 2: Vertex implementation that supports self reachability in the direct construction. This vertex supports n input edges and m output edges. V_{in} is set high if this vertex is the source of a reachability query; low otherwise. A vertex's V_{out} output registers high if it is reachable from the source vertex. This circuit also contains reset logic to clear cycles.

of the core result and only included for completeness.

Given a source vertex v_i in a directed graph G, graph reachability computes the set of vertices $R(v_i)$ reachable from v_i:

$$R(v_i) = \{v_j | v_i \rightsquigarrow v_j\}$$

Given reachability information, other graph properties (*e.g.*, transitive closure) may readily be computed. In Section 5 we use reachability as a "subroutine" to implement other graph algorithms.

We now describe how graph reachability can rapidly and simply be computed in hardware by wiring the target graph as a circuit, placing a signal on the desired source vertex, and monitoring whether this signal arrives at the sink vertex. A trivial *direct construction* transforms G to its corresponding reachability circuit—albeit not to one amenable to dynamic reconfiguration.

To construct directly a (static) circuit to solve reachability on graph G, replace G's edges with wires and every n-input m-output vertex in G with an n-input logical OR-gate fanning to the m out-edges (wires). Figure 1b illustrates the circuit resulting from the application of this construction to the sample graph of Figure 1a. To compute the set of vertices reachable from vertex v_i, assert a high value on the input to vertex v_i and low values on all other inputs v_j, $j \neq i$. After a time T sufficient to propagate the input signal along all paths and through all intervening gates, the set of vertices R reachable from v_i will be identified by high logic signals on their output. Concretely in the example, a high value on V_1's input will propagate to both V_2 and V_3's outputs since they are reachable from V_1. For nontrivial graphs, the required gate and wire propagation time T stands to be much shorter than the running time of a software implementation (see Section 5.1.2).

Three comments on the direct construction—one major and two minor—are in order. First, the direct construction does not admit efficient vertex and edge modification. This is because the circuit is "hardwired" and the placement of OR-gates (vertices) and the routing of wires (edges) is circuit dependent. Complex place and route algorithms are required to map the logical circuit to physical hardware. Gate or wiring changes require re-placing and re-routing the design. The difficulty in performing updates on this construction is basically that the resources between gates—*i.e.*, wires—may be restricted from future connection with new gates; making such connections may require gate relocation. Similarly, removing a route in the direct construction may require multiple, perhaps nonlocal, modifications. To sidestep this problem, we will represent the graph as its *adjacency matrix* in hardware—described in the next section.

The second comment on the direct construction is that it does not correctly handle the case of a vertex reaching itself. In particular, this construction will indicate that a source vertex v_i always reaches itself (through a zero length path) even if no proper path exists. The enhanced vertex implementation of Figure 2 solves this problem by separating the source of a vertex from its input edges with an additional two-input OR-gate. A vertex is now reachable from the source if its REACHED output is high.

The third comment concerns resetting the circuit (upon having computed reachability, for example). Cycles in the graph that lie on paths explored by the reachability computation will continue to exist in an active (high) state even *after* all source inputs are reset to low. This is because every OR-gate in the cycle has a high input which forces that gate's output high; hence cycles are stable upon removal of the original sources. (We exploit this "feature" in computing graph cyclicity information; Section 5.5.) The two-input AND-gate of Figure 2 provides a reset operation to clear cycles. A low value on the RESET line (normally high), clears cycles.

In the following, we omit details of reset circuitry and other easily resolved issues such as gate fan out, *etc.*

4. DGP GRAPH REPRESENTATION

This section describes our construction for mapping graphs to circuits while permitting rapid modification of the graph. The construction is based on the adjacency-matrix representation of a graph; we define it first before describing the DGP mapping proper.

The *adjacency-matrix representation* (see, for example, [4]) of a directed graph $G = (V, E)$ is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

That is, a_{ij} is "true" if a directed edge from i to j exists in G; "false" otherwise. Note that this representation requires $O(|V|^2)$ space since it must be able to hold a complete graph.

An *active adjacency matrix* (AAM) embodies this matrix in reconfigurable hardware and supplies interconnect and glue logic to implement a particular algorithm, *e.g.* reachability. The AAM locates vertices at fixed locations in the hardware array and permits a directed edge between any two vertices, thereby supporting complete graphs. Edge insert and delete operations in an adjacency matrix are $O(1)$ time operations (assuming constant-time array indexing). For an AAM, these operations will require addressing (selecting) a particular cell (or small fixed-size set of cells) in a RCHW array and replacing (writing) a new set of values to that cell (or set of cells). Some off-the-shelf FPGAs have this functionality, *e.g.*, [12].

Vertex deletion in the AAM representation requires $O(|V|)$ steps; more precisely, to delete vertex v, the row and column corresponding to v's in- and out-edges must be cleared ($2|V| - 1$ matrix elements). However, if, for every active

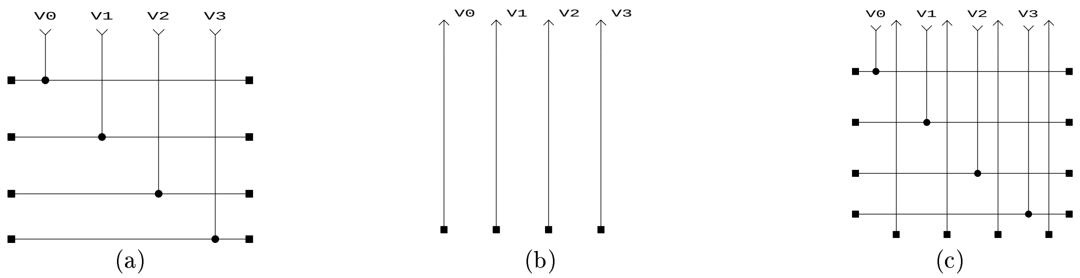


Figure 3: AAM distributor (a) and collector (b) networks. Together they form the complete AAM network (c). The source signal of a vertex v_i propagates down the ith column to every cell in the ith row; distributor/collector crossings locate graph edges; when an edge from $v_i \to v_j$ exists, crossing (j, i) will contain a physical connection. (c) represents a graph with no edges since there are no connections between distributor and collector.

vertex v, the host maintains an edge incidence set I_v of the edges entering and leaving v, deletion of v may be performed in $O(|I_v|)$ steps.[3] In practice, it is possible to use the capability of contemporary RCHW arrays (*e.g.*, [12]) to write an entire row or column in a single operation; to delete v, reset the row and column that correspond to v.

Inserting a vertex into the AAM representation requires either locating an unassigned vertex (row and column) in the matrix; or, failing that, extending the matrix by adding a row and column. Here, we restrict attention to the former case and fix the maximum vertex capacity of the matrix in advance. AAM vertex insertion is handled primarily by the host processor which maintains a list of free vertices in the DGP. Vertex deletion places the prior, now unused, DGP location on this list; new locations for vertex insertion are taken from this list. Vertex insertion can therefore be performed by the host and AAM in constant time.

Having given the time complexities of the various dynamic graph operations, we now describe construction of an AAM to support arbitrary dynamic graphs with up to N vertices ($|V| \leq N$). Here, we give the construction in terms of hardware gates and wires, independent of particular FPGA technologies; a prototype implementation (§6.1) indicates that this construction readily and efficiently maps to some FPGA designs.

The AAM consists of two pieces: a distributor network and a collector network. We first describe these networks. Then, in the context of graph reachability, we examine a sample graph's AAM and the dynamic AAM rewriting necessary to reflect edge/vertex insertions and deletions.

4.1 AAM Distributor

The first piece of an AAM is a *distributor network*. This network distributes the input signal at a particular vertex v along v's out-edges to its immediate successor vertices. Figure 3a depicts the distributor network for a four vertex graph. An arbitrarily large AAM distributor may be constructed by adding columns on the right and rows on the bottom.

[3]The incidence sets I can be maintained on a conventional host processor. Using dictionaries (see, for example, [4]) on the host, the incidence set for vertex v may be updated in worst-case time $O(\log_2 |E|)$ per edge insertion/deletion operation involving v. On average, this cost is $O(\log_2 |\hat{I}|)$ where $|\hat{I}|$ denotes the average size of the incidence sets.

In an AAM, the vertices are aligned along the top edge. Associated with every vertex is an input for the incoming value and an output for the outgoing value. The distributor network contains the output line. For vertex v_i in column i, the distributor routes the value from this line to row i of the AAM. That is, the value at v_i is made available at another vertex v_j by a (potential) connection at (j, i). As we will see, such a connection is made with another network—the collector, described below—that feeds vertex inputs. Note that the distributor is passive; it consists only of wires.

In what follows, we overload the notation V to denote not only the vertex set but also the vector of (binary) values at each vertex along an AAM's top edge. (V_0 denotes the value at the leftmost vertex.)

4.2 AAM Collector

The other piece of a generic AAM is the *collector network*. The collector serves to route values along in-edges into a vertex. Figure 3a depicts a four-vertex collector network; it can be made arbitrarily large by lengthening the collection wires and adding vertices (and wires) on the right. Values carried by the collector stem from the distributor and are routed onto the collector through graph-specific collector-distributor connections; this is illustrated in the reachability example below.

The collector, as the distributor, is passive and asynchronous. Signals propagate through either network at the propagation speed of the implementation medium.

4.3 Complete AAM

A complete AAM composed of distributor and collector networks is shown in Figure 3c. When representing a graph without edges (as here), collector and distributor networks are disjoint. Insertion of an edge will make a connection appropriate to the graph algorithm under consideration.

5. DGP ALGORITHMS

This sections contains DGP algorithms for vertex reachability, transitive closure, shortest unit path, connected component identification, and cycle detection. Since reachability is central to these algorithms, we describe it first.

5.1 Reachability

The additional logic to implement reachability with an AAM is described here and the DGP performance of reachability

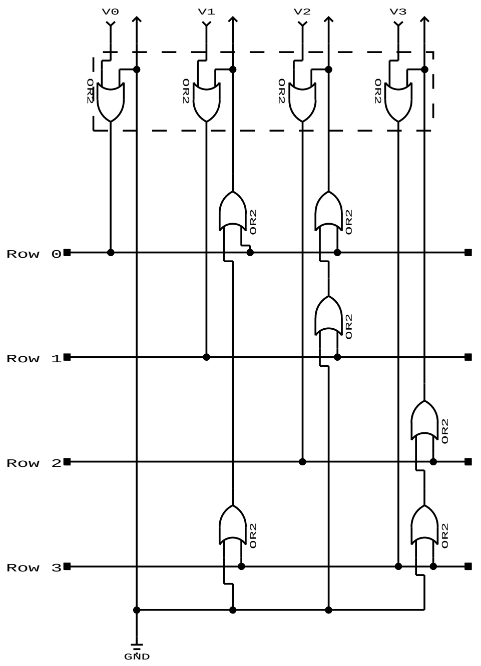


Figure 4: Full AAM for the sample graph of Figure 1a. If the graph contains an edge from $v_i \rightarrow v_j$, the distributor network connects to the collector network with an OR-gate at position (j, i). The logic in the dashed box implements the graph reachability function by propagating the value of a vertex's in-edges to its out-edges.

analyzed in terms of propagation delays. Additional circuitry is then presented that can further decrease the time of this computation.

5.1.1 Construction

We embed a graph $G = (V, E)$ in a general AAM (Figure 3) to obtain reachability queries as follows. An edge $e \in E$ from vertex $v_i \rightarrow v_j$ will connect the distributor subnet emanating from v_i to the collector subnet feeding into v_j. Since multiple edges may enter a vertex, OR-gates are used to combine values along multiple in-edges. Figure 4 contains the complete wiring for the four-vertex sample graph G_s of Figure 1a.

The OR-gates in the dashed box are specific to reachability and serve to combine the algorithm input (*i.e.*, source vertex) with the values on in-edges and then to propagate this combined value onto the vertex's out-edges. Algorithms other than reachability may require other circuitry at a vertex, but many graph algorithms have reachability at their core (Section 5). Note that for reachability (at least) the amount of per-vertex circuitry is quite small and scales with $|V|$.

The OR-gates internal to the AAM (not in the dashed box) form connections (edges) between vertices. An OR-gate in the AAM corresponds directly to a non-zero entry in a graph's regular adjacency matrix (§4). When the graph contains an edge from $v_i \rightarrow v_j$, the distributor network connects to the collector network with an OR-gate at position (j, i). (Note that $(0, 0)$ is the upper left corner.) The inputs to the gate are the collector/distributor lines at that position; its output is the collector line for position $(j, i-1)$. The OR-gates in the zeroth row, for example, connect the output of V_0 to the inputs of V_1 and V_2 respectively. The two gates in the third (rightmost) column serve to combine the values entering V_3 from itself (gate in third row) and from V_2 (gate in second row).

The initial value for a collector line is zero (ground) for reachability. This is because a vertex with no in-edges in essence always sees zeroes from these "non-edges."

Carrying the example forward, we now modify the depicted AAM and associated graph with edge insertions and deletions. Consider deletion of the edge $V_1 \rightarrow V_2$. In the AAM, the OR-gate at position $(2, 1)$ is rewritten with a wire that connects the (now dangling) left input from the V_2 collector to the left input of the gate at $(2, 0)$. The V_1 distributor wire to the rewritten gate's right input is also deleted by this operation. This modification therefore consists of indexing into the two dimensional AAM and rewriting the functionality of the logic and routing at the target location. Suppose now that the edge $V_0 \rightarrow V_3$ is inserted. This requires writing an OR-gate into position $(3, 0)$ with left input from the V_3 collector and right input from the V_0 distributor output. Figure 5 contains the updated graph (a) and the modified AAM (b) respectively.

5.1.2 Analysis

In contrast to conventional big-O analysis that counts algorithm steps, we analyze DGP graph reachability in terms of gate and signal propagation delays. (We use big-O when a graph algorithm requires a series of asynchronous computations to capture the constant overhead of invoking such a computation.)

Reachability in a graph is bounded by the number of vertices $n = |V|$. That is, in the worst case a DGP must propagate a signal along at most n edges serially. Worst-case propagation of a signal along a *single edge* in a DGP requires time

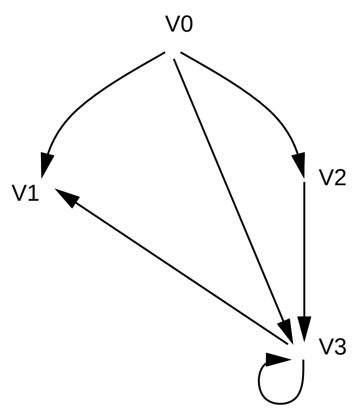
$$T = T_d(n) + T_c(n)$$

where $T_d(n)$ is an upper bound on the time to propagate along a distributor wire of length $O(n)$. Likewise, $T_c(n)$ is an upper bound on the time to propagate along a collector wire—containing up to n gates for distributor-collector connections—of length $O(n)$. Combined with the worst-case requirement of n serial edges, reachability therefore can be computed in time nT.

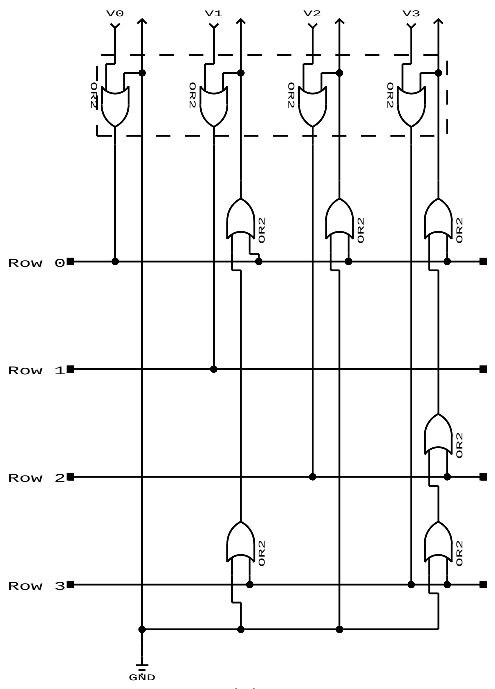
Two comments on this conservative analysis are required. First, $T_d(n)$ and $T_c(n)$ are expected to be extremely small. In a semiconductor chip, for example, $T_d(n)$ is essentially the propagation of electrons through a conductive substrate—typically at a significant fraction of the speed of light. $T_c(n)$ may contain $O(n)$ serial gate delays, each of which is typically on the order of nanoseconds. Even for large n, $T_c(n)$ is quite small.[4] That said, propagation delays will impose limits on attainable computation speeds as graph size increases.

Secondly, reachability paths may not span the $n = |V|$ vertices assumed by the above analysis. That is, for many

[4]In an FPGA implementation for example, $T_d(n)$ and $T_c(n)$ are, to within a small factor, the time it takes to propagate a signal the length of the die. $T_c(n)$ can perhaps be further reduced by using tristate logic to combine multiple values reaching a single collector.



(a)



(b)

Figure 5: (a) Graph of Figure 1a after deletion of $V_1 \to V_2$ and insertion of $V_0 \to V_3$. (b) Corresponding modified AAM.

graphs a smaller number of reachability operations may suffice. It is safe to terminate the reachability algorithm as soon as *stasis*—no changes during a time interval slightly greater than T—is reached.

5.1.3 Stasis Detection

Stasis detection can be used to reduce reachability time for graph queries from a vertex v_i that need not traverse the worst-case path; *i.e.*, one passing from v_i to v_j with fewer than $n-2$ intermediate nodes. As we shall see, stasis detection can significantly boost performance (§6).

At regular intervals, a *stasis detection circuit* (SDC) synchronously monitors the signal arriving at a vertex. The SDC captures the current signal and compares it to the signal captured in the previous interval. If the values differ at any vertex, the reachability computation is still incomplete. Key to proper SDC operation is the sampling interval—it must be larger than T, the maximum edge propagation time, to ensure that two consecutive samples allow time for traversal of the longest edge.

An SDC can be implemented with a register and an AND-comparator at every vertex (for storing and comparing the current and prior sample) along with a combining tree of AND-gates whose single output is asserted if the values at all vertices did not change from the last sample. The depth of this tree is $\log_2(|V|)$ and contains $O(|V|)$ gates. The clock period of the SDC must be slightly longer than T. Since the maximum propagation time grows faster than the depth of the SDC, the delay of the tree is less than T for all but the smallest values of $|V|$ (see Section 6).

5.2 Transitive Closure

Transitive closure (TC) for a directed graph $G = (V, E)$ is defined as the graph $G^* = (V, E^*)$ where

$$E^* = \{(i, j) : \exists \text{ path from } v_i \text{ to } v_j \text{ in } G\}$$

The adjacency matrix for E^* can be built in n reachability steps by computing the n vectors of vertices reachable from each v_i, for $0 \le i < n$. (DGP reachability computes a single such vector asynchronously and in parallel.) Using the complexity of DGP reachability derived above (§5.1.2), TC therefore has a DGP complexity of $O(n^2T)$. Note that this algorithm is cubic in complexity since T is dependent on wire lengths proportional to n. Simulation results of DGP TC are in Section 6.

5.3 Shortest Unit Path

Additional circuitry, described here, is required to compute the length of the shortest unit[5] path (SUP) from v_i to v_j. The OR-gate at each vertex (see Figure 4) that gates the collector signal to the distributor is replaced with a *clocked latch*. That is, the signal reaching a vertex v is clocked into the latch every clock period and, at the next clock, injected into v's distributor. This circuit operates synchronously. After initially asserting a signal at v_i, the circuit is clocked until either the signal propagates to v_j or $n = |V|$ clocks have been issued. In the first case, the number of issued clocks corresponds to the length of the shortest unit path from v_i to v_j; in the latter case, no path from v_i to v_j exists. Since a clock can be issued after waiting a time slightly greater than the worst-case propagation delay, T—as with

[5]All edges of length one.

stasis detection (§5.1.3)—computing the length of the SUP requires at most time $O(nT)$.

Note that this algorithm for SUP produces only the path length and not the path itself. Additional hardware in the AAM can be used for extracting the SUP; Section 7.2.

Design of a DGP algorithm to compute the shortest path in the presence of variable edge weights is an open problem.

5.4 Connected Components

A connected component (CC) in an undirected graph G is a set of vertices and edges $C \subseteq G$ such that for all $v_i, v_j \in C$ and $i \neq j$ vertex v_i reaches v_j. The DGP CC algorithm for undirected[6] G is as follows. Initially, no vertex is in a CC so a component C_{v_0} may be constructed first. For a vertex v_i that is not in a CC, construct the component C_{v_i} and place in it v_i and the vertices reachable from v_i; this set of reachable vertices is computed using DGP reachability defined above (§5.1). In the worst case (when the graph has no edges), computing all components requires $n = |V|$ reachability steps—hence the DGP complexity of computing G's connected components is $O(n^2T)$.

Either the host processor or the DGP can maintain state to identify vertices that are not yet component members. If handled by the host, DGP CC computation stands to suffer from overheads in the host-RCHW interface since it relies on many host operations which can be expensive. These operations can be accomplished in hardware as follows. Maintain an n-bit register A where bit i set indicates that vertex v_i is already in a component. Initially A is clear. While there is an unset bit u in A, reset the reachability AAM's vertices, assert vertex u in the AAM, output the result vertex vector as a new connected component, and update A with the "or" of the result vector and A. Circuitry to detect unset bits can be similar to that used by processor instructions that find the least significant set bits in a vector, for example.

A DGP algorithm for strongly connected components is an open problem.

5.5 Cycle Detection

DGP reachability readily determines whether or not a graph is acyclic. Recall that reachability uses `OR`-gates at vertices to merge multiple in-edge signals (§5.1). When a vertex lies on a cycle that is active, it receives and outputs a high value; that is, its output is connected to its input via a potentially circuitous path. Due to feedback, active cycles continue to be active even after removal of the source inputs since a high incoming signal suffices to set the outputs. DGP reachability, conducted with all source inputs initially high, reveals vertices that lie on a cycle (or multiple cycles) when the source inputs are subsequently lowered since such vertices retain a high output value. The set of vertices that share a cycle with a single vertex v can be computed by setting only v's source input and capturing the set of vertices that stay high after v's source input is lowered.

Cyclicity can thus be determined in time nT.

6. RESULTS

Results take two forms: an FPGA implementation of a DGP network and simulation of DGP transitive closure with comparison to software.

[6]An undirected graph in a DGP uses two directed edges, $v_i \rightarrow v_j$ and $v_j \rightarrow v_i$, to denote an undirected edge $v_i \leftrightarrow v_j$.

6.1 Implementation

To explore DGP layout issues, a 16x16 DGP network has been constructed and tested in a Xilinx XC6216 FPGA on a VCC Hotworks PCI board with Webscope [11]. The XC62xx [12] is a family of fine-grain highly reconfigurable FPGAs. Reconfiguration of a single cell takes on the order of a microsecond (8 bit, 50Mhz bus). A cell is connected directly to its nearest neighbors. Hierarchically, lines of length 4, 16, and the length of the chip are available to some cells. Where possible, our prototype uses the longest available line for efficiency.

Highly compact mappings of one vertex per row/column are possible in the XC62xx device family. Aside from the small amount of algorithm specific logic and the SDC that may sit along the top rows of a DGP, an XC62xx device of size NxN cells can hold graphs with nominally N vertices! Edge insertion/deletion is currently manual via Webscope as is extraction of the reachability result.

Since the DGP algorithms are asynchronous, the inexistence of race conditions must be established. For the DGP algorithms of this paper, races do not exist for the following reason. During a specific computation (say reachability) a gate changes state at most once. Gate states are reset only when the circuit is reset. Furthermore, once a line goes high, it stays high (until circuit reset). Hence, even though a graph and its corresponding DGP circuit may contain cycles, races leading to indeterminate results cannot occur.

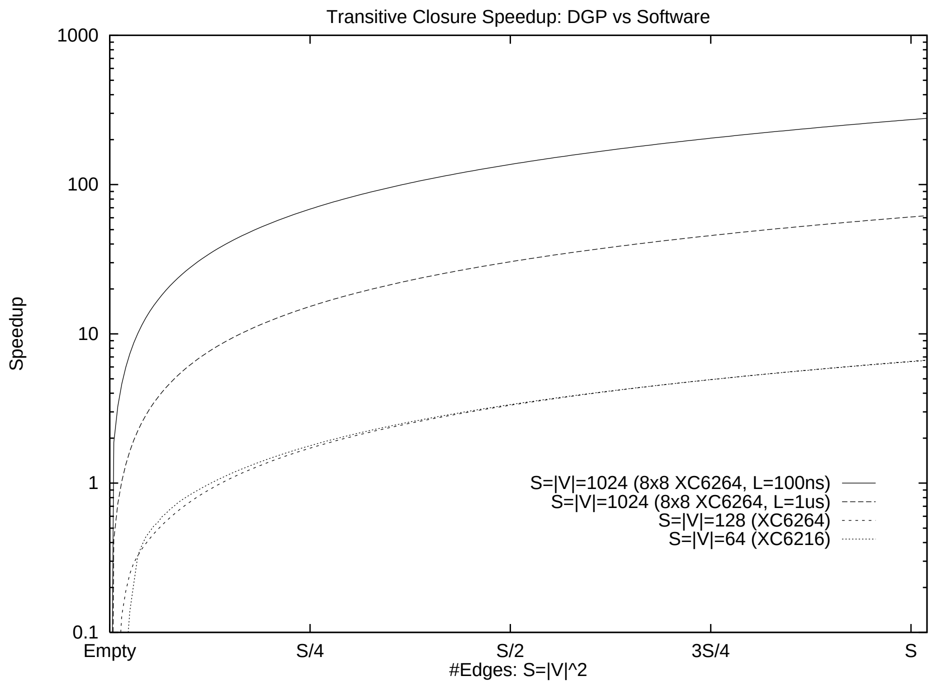
6.2 Simulation

To determine the potential speedup of a DGP graph algorithm over its software counterpart, we ran transitive closure simulations for graph sizes $|V| =$ 64, 128, and 1024. The first two sizes correspond to possible DGP implementations in the XC62xx family: XC6216 (64x64 cells) and XC6264 (128x128 cells). The 1024-vertex graph presupposes a hypothetical 8x8 array of XC6264s. In general, NxN surfaces of FPGAs could be constructed; we chose 8x8 since it is attainable in today's technology. For this hypothetical array, we require connection of edge cell inputs and outputs at all FPGA boundaries (but no hierarchical routing between FPGAs); since pin packaging requires multiplexing to support such connections, we presume the dies to be connected directly, perhaps as a multichip module. We consider a range (100ns to 1us) of inter-die communication delays for the 1024-vertex simulations. Xilinx timing specifications [12] for XC62xx[7] gate and routing connection delays are: logic gate, 2.5ns; length-1 wire, 1.0ns; length-4 wire, 1.5ns; length-16 wire, 2.0ns; chip-length wire, 3.0ns. Timing analysis of the implementation (§6.1) via Xilinx tools indicates that the simulation timing calculations are correct.

A simulation consists of computing reachability from every graph vertex on random graphs holding $0 < E \leq |V|^2$ edges. A simulation thus computes transitive closure over random graphs varying in edge-set size. Simulations were performed for the three graph sizes using DGP with and without stasis detection circuitry[8] (§5.1.3).

[7]Due to process improvements and a smaller feature size, the timings for the XC6264's internal interconnects relevant to this paper are very close to the timings of the XC6216 so the published XC6216 timings were used for both; private communication, Xilinx.

[8]For the DGP simulation with stasis detection, the period of the SDC (assuming 20ns per tree level) for the three graph



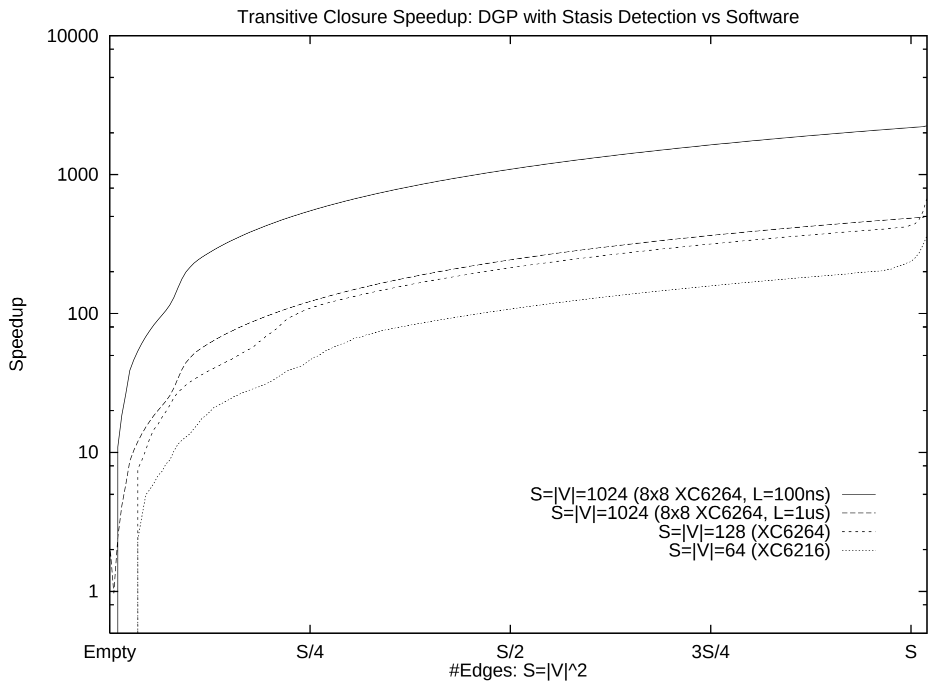


Figure 6: Transitive-closure speedup for DGP over software with and without DGP stasis detection. The Y-axis (log scale) is the ratio of DGP to software running times. (Note the Y-axis in the bottom graph extends an order of magnitude further than the top graph's Y-axis.) The X-axis' S parameter is $|V|^2$ for vertex set sizes of $|V| = 64$, 128, and 1024 corresponding to the XC6216, XC6264 and two 8x8 arrays of XC6264s (using two different values, L, for inter-die delays) respectively. The simulation is on random graphs; the text contains further description of this procedure and a discussion of the results.

Figure 6 contains the simulation results as DGP speedup relative to a uniprocessor software implementation. Note the log scale on the Y-axis on both plots and the increased Y-axis range on the bottom plot. For the hypothetical 8x8 FPGA array, each plot contains two curves representing 100ns and 1us inter-die communication delays. The software algorithm used for comparison is a C program (compiled -O3) for breadth-first search on adjacency lists that computes reachability from every vertex. The software ran on a 500MHZ DEC Alpha and cycle counts were obtained via `pixie` profiles. Note that such counts are conservative since they do not contain cache misses, among other things. For both DGP and software, only the times for the reachability computations were summed; the times to create the initial graph and to reset the graph state after a reachability query are not in the total.

From Figure 6 it is evident that DGPs with SDC stand to significantly outperform their software algorithm counterparts. Even without stasis detection, DGPs could provide useful speedup. In the 64x64 and 128x128 DGPs with SDC, maximum speedups exceed two orders of magnitude; in the hypothetical array, speedup exceeds three orders of magnitude. Actual speedups are potentially even greater since the simulations were conservative; for example, a AAM collector line (§4.2) always contained the maximum number of `OR`-gates independent of graph size. DGP behavior on small graphs first exhibits a dip (not visible) followed by a sharp rise (visible) and then another sharp rise as the graph becomes complete. These effects are due to the non-linearity of connectivity probabilities as a function of edges in random graphs [2].

7. FUTURE DIRECTIONS

Here we address a couple of seemingly surmountable obstacles that impede the practicality of DGPs: the quadratic hardware requirement and extraction of solution paths. Then we outline how DGPs might be used to realize general Turing-complete computation via graph reduction.

7.1 Virtual DGPs

Future device integration promises smaller and denser features; this should allow DGPs for larger graphs. Additionally, devices such as PLAs or ASICs could more effectively utilize semiconductor resources in implementing DGP functionality. (A DGP utilizes only a small fraction of the XC62xx's resources.) It still seems necessary however to seek other solutions to DGP's quadratic hardware requirements. Since DGPs have regular structure, they are promising candidates for virtualization. For example, one can divide an AAM (§4) in half vertically and alternately swap the left and right halves into a rectangular RCHW array (*e.g.*, 1x2 FPGAs) from a configuration memory. Signals that propagate out of one half are captured and retained for the other half in registers at array edges. Many other virtualization options are available. Since configuration memory is local to the RCHW, much bandwidth is available for reading/writing configurations from this memory to the RCHW array. Even so, the cost of swapping configurations can be large relative to the time spent computing in the currently resident configuration. Further study is required to understand the time/space tradeoffs involved in virtualizing DGPs.

sizes is much less than the propagation time T along the graph circuit's longest edge.

7.2 Path Extraction

For some graph-based applications, the *existence* of solution paths—*i.e.*, reachability information that is independent of the path's vertex enumeration—suffices. For example, transitive closure requires only reachability information without the reaching paths having to be explicit. Other graph applications, however, require the list of vertices that constitute the solution path.

For example, shortest path computations are often performed in order to identify such paths for future use. DGP SUP (§5.3) provides the length of the shortest path, but not the path itself. Additional hardware in the AAM connections can be used to extract the solution path at an extra, albeit small, space cost. In Section 5, `OR`-gates implement graph edges by making AAM connections from the distributor to the collector networks. Here we augment this per-connection circuitry to include a latch. This latch is set when the corresponding edge is traversed by a DGP computation. Furthermore, this latch is active only when the edge's target vertex has not yet been reached; *i.e.*, a high signal on the target vertex's distributor disables the latch. The latch's state therefore indicates whether or not a signal that propagated along its edge was in the set of edges to first reach the target vertex. After a SUP computation, the path is retraced (by a host processor or by specialized hardware) from the target vertex to the source vertex along in-edges that have their latch set. Note that multiple in-edges to a vertex may have their latch set in which case multiple SUPs exist; *i.e.*, the target vertex was reached in the same SUP clock cycle along multiple paths.

We anticipate that similar augmentation of the AAM's per-edge circuitry can be used to extract the solution paths computed by other DGP algorithms.

7.3 Parallel Graph Reduction

Parallel graph reduction represents a program (code and data) as a graph. This graph is continually reduced (or rewritten)—in parallel—until the program's result is obtained. Graph reduction via combinatory logic is equivalent to a Turing machine and forms the basis for some functional language implementations (see [10; 7] for background). Since DGPs support dynamic graphs, it may be possible to use vertices as combinators and edges as pointers in the combinator graph. Certainly, graph reduction is much more complicated than the graph algorithms given for DGPs in this paper. A DGP for graph reduction would require complex vertex and edge logic.

A simple combinator system requires only two rewrite operations

- $\mathbf{K}\ x\ y \Rightarrow x$
- $\mathbf{S}\ f\ g\ x \Rightarrow (f\ x)(g\ x)$

to denote any computable function. Here, **K** and **S** are combinator literals. Lowercase identifiers are variables bound to combinator terms built from the two rules. The identifiers on the left-hand side of the arrows are parameters to the combinators. When parameter arguments arrive, the left-hand side is rewritten (reduced) to the right-hand side. Juxtaposition of combinator terms denotes function application. A program's code and data is a combinator term that forms a graph. This graph contains cycles to express recursion (loops) and may share combinator terms. The **K** combinator simply returns its first argument, discarding the second. The **S** combinator takes two combinator expressions that must be functions (f and g) and applies both to x. The two results of **S**'s application of f and g are then themselves juxtaposed and evaluated as a function application.

In a hypothetical DGP graph reducer, vertices could represent combinator literals and application nodes. Evaluation of a **K** rule would result in the vertex bound to x to be propagated to all places in the combinator graph that reference this particular **K** rule. That is, vertices have edges to their containing combinator terms and, upon evaluation, would propagate their identity to these terms. Since node indices are required on node creation and are communicated among nodes during rewrite, edges must be able to propagate vertex indices; this requires the transmission of multiple bits along edges, either serially or in parallel. The **S** rule is more complicated since it requires allocation of application nodes and their initialization. A suitable DGP must therefore support parallel vertex allocation and an automatic means for reclaiming spent vertices (a process known as garbage collection).

If DGP graph reduction is indeed possible, multiple reducible nodes can be rewritten in parallel which gives fine-grain parallel execution of general programs.

8. SUMMARY

This paper described a novel transformation that converts graphs to circuits. Such circuits can solve a number of graph algorithms extremely quickly and are, using reconfigurable hardware, amenable to dynamic vertex and edge sets that change over time. Simulation on random graphs indicates that speedups over fast software algorithms—a factor >1000 in some cases—are possible with dynamic graph processors. Implementation of a prototype DGP in an FPGA substantiates the simulation results and demonstrates that compact DGP mappings are possible in off-the-shelf reconfigurable hardware. Future work includes extension to larger graphs, design of more DGP algorithms for other graph properties, and exploration of DGP applications such as for general computation via graph reduction.

Acknowledgments

Thanks to Miron Abramovici and Jose T. de Sousa for insightful discussions of this work. Bob Slous at Xilinx supplied important details of the XC62xx. Vishwani Agrawal provided useful comments on a draft of this paper. The reviewers' comments improved the exposition and unearthed a technical bug.

9. REFERENCES

[1] J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. In *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development & Computing*, pages 225–236, November 1996.

[2] B. Bollobas. *Random Graphs*. Academic Press, 1985.

[3] S. T. Chakradhar and V. D. Agrawal. A novel VLSI solution to a difficult graph problem. In *Fourth International Symposium on VLSI Design*, pages 124–129. IEEE, January 1991.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] A. Dandalis, A. Mei, and V. K. Prasanna. Domain specific mapping for solving graph problems on reconfigurable devices. In *6th Reconfigurable Architecture Workshop*. IEEE, April 1999.

[6] W. D. Hillis. *The Connection Machine*. MIT Press, 1985.

[7] S. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[8] T. Kean. Using CAL to accelerate maze routing of CAL designs. In *Proceedings of 2nd Int. Workshop on Field Programmable Logic and Applications*. 1992.

[9] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. Morgan–Kaufmann, 1992.

[10] D. A. Turner. A new implementation technique for applicative languages. *Software Practice & Experience*, 9:31–49, 1979.

[11] Virtual Computer Corp. *H.O.T. Works User's Guide, Ver. 1.0*, 1997.

[12] Xilinx Inc. *The Programmable Logic Data Book; XC6200 product specification V1.0*, 1996. `http://www.xilinx.com`.