Solving graph problems with dynamic computation structures

Jonathan Babb, Matthew Frank, Anant Agarwal

MIT Laboratory for Computer Science
Cambridge, MA 02139

ABSTRACT

We introduce *dynamic computation structures* (DCS), a compilation technique to produce dynamic code for reconfigurable computing. DCS specializes directed graph instances into user-level hardware for reconfigurable architectures. Several problems such as shortest path and transitive closure exhibit the general properties of closed semirings, an algebraic structure for solving directed paths. Motivating our application domain choice of closed semiring problems is the fact that logic emulation software already maps a special case of directed graphs, namely logic netlists, onto arrays of Field-Programmable Gate Arrays (FPGA). A certain type of logic emulation software called virtual wires further allows an FPGA array to be viewed as a machine-independent computing fabric. Thus, a virtual wires compiler, coupled with front-end commercial behavioral logic synthesis software, enables automatic behavioral compilation into a multi-FPGA computing fabric.

We have implemented a DCS front-end compiler to parallelize the entire inner loop of the classic Bellman-Ford algorithm into synthesizable behavioral verilog. Leveraging virtual wire compilation and behavioral synthesis, we have automatically generated designs of 14 to 261 FPGAs from a single graph instance. We achieve speedups proportional to the number of graph edges – from 10X to almost 400X versus a 125 SPECint SparcStation 10.

Keywords: dynamic computation structures, user-level hardware, FPGA computing, reconfigurable architecture workstation, logic emulation, virtual wires, closed semirings, shortest path.

1 INTRODUCTION

Configurable computers based on Field Programmable Gate Arrays (FPGAs) are capable of accelerating suitable applications by several orders of magnitude when compared to traditional processor-based architectures (see Splash[7] and PAM[3]). This performance is achieved by mapping a user application into a gate-level netlist that may be downloaded onto programmable hardware. However, the programming paradigm to achieve such performance is often no more than a set of hand-crafted circuits, one per FPGA in the system. Programmers must explicitly account for machine-level details, such as FPGA capacity and interconnect, prohibiting the development of automatically-compiled, architecturally-independent applications.

This research addresses compilation issues in reconfigurable computing as part of MIT's Reconfigurable Architecture Workstation (RAW) project. The RAW system leverages previous multi-FPGA work, virtual wires,[1] in conjunction with behavioral compilation technology, to view an array of FPGAs as a machine-independent

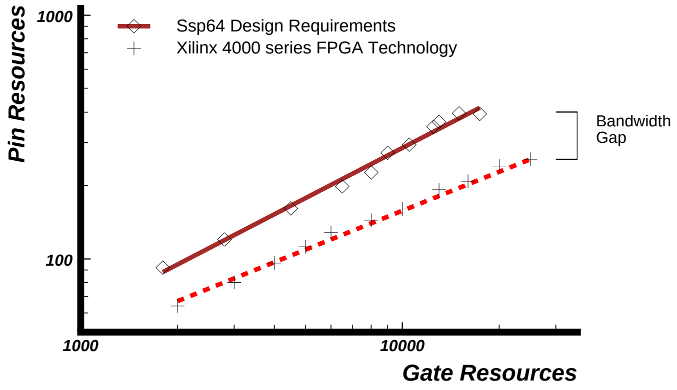

Figure 1: Pin Bandwidth Gap

computing fabric. Given this viewpoint, we have developed a new software system which automatically generates dynamic computation structures based on user input for particular directed graph problems. These structures are specific to an application problem instance, yet are independent of any underlying architectural details. This new software operates in a framework which allows these structures to be automatically compiled onto a large array of FPGAs (an IKOS VirtuaLogic Emulator[8]) without user intervention.

The rest of this paper is organized as follows: Section 2 provides motivation based on available virtual wires technology. Section 3 then describes dynamic computation structures within the context of a complete reconfigurable compiler system. After Section 4 overviews the application domain, Section 5 describes the prototype hardware system. Section 6 then presents results for three sets of graph problems. Finally, Section 7 makes concluding remarks and presents future research directions.

2 MOTIVATION

Reconfigurable architectures are frequently composed of more than one FPGA. Previous research in multi-FPGA systems has shown that there exists a *bandwidth gap* between the required inter-chip routing resources of a typical architecture and the pin resources available in FPGA technology.[1] That is, naive mapping of architecture gates to FPGA gates and architecture signals to FPGA pins results in severe pin limitations and low gate utilization. Figure 1 shows this gap for various partition sizes of the Ssp64 shortest path structure reported in the results section in comparison to the Xilinx 4000 series FPGAs. Previous techniques required the user to carefully hand craft individual FPGA logic to overcome these pin limitations. One solution, *virtual wires*, has eliminated this pin limitation barrier of multi-FPGA systems by intelligently multiplexing each physical wire among multiple logical wires and pipelining these connections at the maximum clocking frequency of the FPGA. These connections are synthesized over a pipelined, statically-routed communication network, eradicating the need for expensive crossbars and esoteric PC-board technology, allowing the construction of scalable, direct-interconnect FPGA fabrics composed of solely of commodity FPGAs.

The development and commercial availability of virtual wires compiler technology enables the efficient combining of multiple FPGAs for use as a single, giant sea of gates by higher-level synthesis compilation steps. While this technology has primarily been applied to in-circuit emulation and logic simulation acceleration, it has also been applied effectively to reconfigurable computing of *hardware subroutines*, where an FPGA array implements a verilog[4] version of a subroutine in a C program and connects to the software by remote calls from a host work-

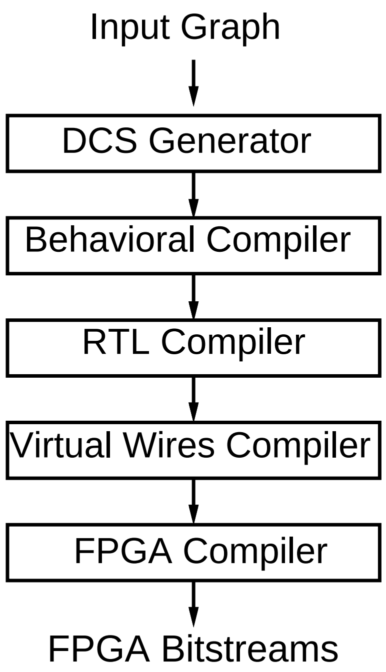

Figure 2: Compiler Flow

station.[2] This paper presents further results on the use of virtual wires technology for reconfigurable computing and validates its effectiveness for massive FPGA computing systems.

3 DYNAMIC COMPUTATION STRUCTURES

To explore user-level hardware for reconfigurable architectures, we introduce *dynamic computation structures* (DCS), a compilation technique in which we dynamically modify the underlying hardware architecture as a function of the user's input. DCS compilation can be viewed as an extreme case of generation of dynamic code,[6] where new processor instructions are dynamically generated based on the input data-set. Dynamic code generation is a software technique that allows specialization and optimization of code based on program input. Because the program input is available, compiler optimizations like constant folding, dead code elimination and removal of branches to expose instruction parallelism enable the code generator to create significantly faster code.

Dynamic computation structures extend dynamic code generation to reconfigurable architectures by creating hardware configurations, rather than software routines, to solve a specific problem instance. To specialize below the instruction set abstraction, this technique automatically creates new operators, distributes memory accesses, minimizes data widths, and reduces many complex data structures to wire permutations. The resulting structures are thus tailored to a specific problem instance.

3.1 Software Tool Flow

Figure 2 shows the overall software tool flow for the compilation system. In comparison with standard compilation technology for microprocessor computing, the algorithmic complexity of the transformations involved

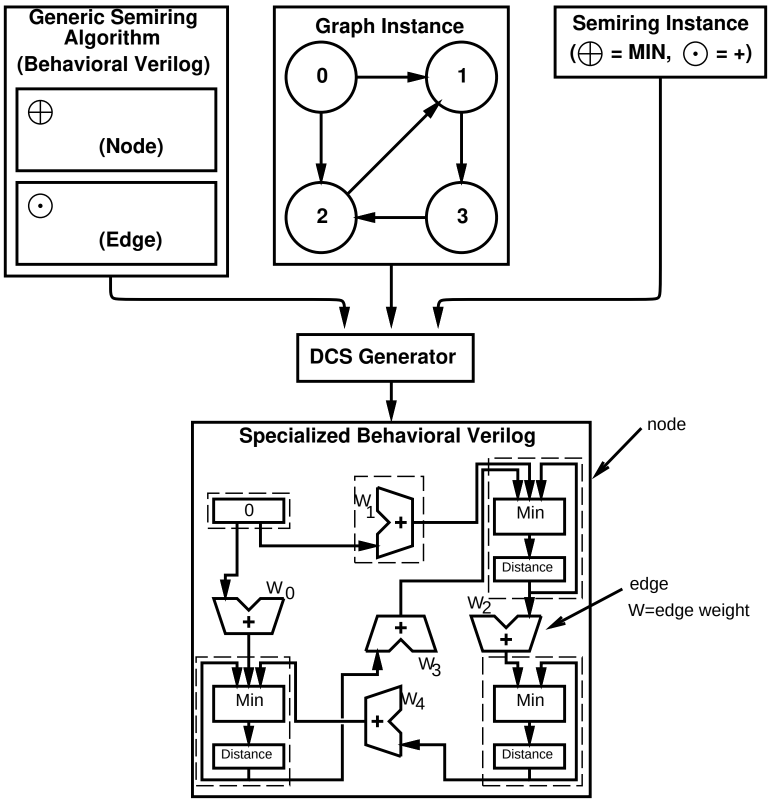

Figure 3: DCS Flowchart

is much higher. This system begins with the DCS generator, a compiler written in C, that takes an input graph problem specification in a high-level form and generates specialized behavioral verilog code. This verilog code describes a hardware instance that solve the given problem at a high-level, with no references to any underlying technology. The next two tools consist of a behavioral compiler and an RTL compiler which together map the input verilog into a single, gigantic netlist composed of generic logic gates in a reference technology. The virtual wires compiler then maps this netlist into multiple FPGAs. More specifically, the virtual wires compiler treats this netlist as a design to be emulated, partitioning, placing, and scheduling inter-FPGA connections to produce individual netlists for each FPGA in the emulator. Each individual netlist is then processed by the an FPGA compiler that places and routes the netlist for the target FPGAs, producing FPGA bitstreams. These layers hide successive levels of detail from the DCS generator: the FPGA layer hides the internal FPGA details; the virtual wires layers hides the inter-FPGA topology and communications as well as the FPGA gate and pin capacity; the RTL layer hides the reference technology libraries; and the behavioral layer hides state machine and datapath details.

3.2 The DCS Generator

Given specific user input, the DCS generator creates computation structures in synthesizable behavioral verilog. This verilog code is application-specific and architecture-independent. It can be targeted at any multi-FPGA system. Figure 3 demonstrates an example of the operation of the DCS generator for the semiring applications introduced in Section 4.

The DCS generator takes as input a library of generic descriptions (in behavioral verilog) of node and edge computation structures, a topological description of a specific user input graph instance, and the semiring parameters for the $\oplus$ and $\odot$ operators discussed in Section 4. The generic structures have parameters, including data widths, input edges per node, and semiring operators, that are set whenever a particular node or edge is instantiated. The DCS generator uses these structures to instantiate hardware with the proper bus widths and operations corresponding to the input specification. In addition the DCS generator specifies connections between these structures corresponding to the edges in the input graph. The DCS generator thus creates a description of a single circuit of higher level functions which can be synthesized to gates, partitioned, and compiled to specific FPGA technology by the lower levels of the compilation hierarchy.

In addition to the specialized behavioral verilog code created by the DCS generator, we also synthesize an interface through which the software running on a host workstation and the emulated design can communicate. Section 5 discusses this host interface in more detail. Briefly, in the circuit shown in Figure 3 the edge weight registers (marked by a W_i) are connected to the host processor interface by a writable bus. The bus is not shown in the figure. Likewise the node distance/path registers (marked "Distance" in the figure), are connected to the interface by a readable bus. On the host side these buses are mapped into the processor address space so that each edge weight can be written, and each node path read by the host software program.

4 APPLICATION DOMAIN

A large class of applications can be solved with the general strategy of compiling computation structures for a given problem instance. That is, given an input data set and a generic solution algorithm, we may compile a computation structure that is specialized to rapidly solve a particular problem instance. For a Von-Neumann style processor, such compilation results in specialized instructions, *compiled-code*, that when executed solves the problem at hand. For reconfigurable architectures, the computation structures are hardware descriptions to be downloaded into reprogrammable technology. This application domain broadly includes certain graph problems, logic simulation and verification, cellular automata, neural networks, N-body simulation, fluid dynamics, and discrete-event computer network simulation.

Within this domain, this paper focuses on the use of a reconfigurable architecture to solve high-speed computing problems involving directed graphs. Motivating our choice of this focus is the fact that logic emulation software is already capable of mapping a special case of directed graphs, netlists, onto an FPGA array. The specific problems we solve, including transitive closure and shortest path, exhibit the general properties of *closed semirings*[9] (Figure 4), an algebraic structure for solving directed paths. Closed semirings allow iterative solutions of directed path problems. These problems are often iterative subproblems of flow and matching linear programs, which in turn often form the inner loop of complex, combinatorial optimizations for power generation and transmission, water resource allocation, and silicon compilation.

For experimental cases, we consider three semiring cases: transitive closure: $S_{tc} = (\{0,1\}, OR, AND, 0, 1)$, shortest path: $S_{sp} = (\{0 \ldots 32767, +\infty\}, MIN, +, +\infty, 0)$, and a multiplicative version of shortest path: $S_m = (\{0 \ldots 32767, +\infty\}, MIN, \times, +\infty, 0)$. These problems can all be solved with variations of the Bellman-Ford shortest path algorithm (Figure 5) in order $O(nm)$, where n is the number of nodes and m is the number of edges, on a sequential processor. However, by parallelizing the inner loop, we will show solutions to semiring problems in order $O(n)$, given a sufficiently large reconfigurable architecture.

5 HARDWARE SYSTEM

A *closed semiring* is a system $(S, \oplus, \odot, 0, 1)$ where S is a set of elements, and $\oplus$ and $\odot$ are binary operations on S, satisfying the following five properties:

1. $(S, \oplus, 0)$ is a *monoid*, that is, it is *closed* under $\oplus$ [i.e., $a \oplus b \in S$ for all a and b in S], $\oplus$ is *associative* [i.e., $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for all a, b, c in S], and 0 is an *identity* [i.e.,$a \oplus 0 = 0 \oplus a = a$ for all a in S]. Likewise (S,$\odot$,1) is a monoid. We also assume 0 is an *annihilator*, i.e., $a \odot 0 = 0 \odot a = 0$

2. $\oplus$ is *commutative*, i.e., $a \oplus b = b \oplus a$, and *idempotent*, i.e., $a \oplus a = a$.

3. $\odot$ *distributes* over $\oplus$, that is, $a \odot (b \oplus c) = a \odot b \oplus a \odot c$ and $(b \oplus c) \odot a = b \odot a \oplus c \odot a$.

4. If $a_1, a_2, \ldots, a_i, \ldots$ is a countable sequence of elements in S, then $a_1 \oplus a_2 \oplus \cdots \oplus a_i \oplus \cdots$ exists and is unique. Morever, associativity, commutativity, and idempotence apply to infinite as well as finite sums.

5. $\odot$ must distribute over countable infinite sums as well as finite ones (this does not follow from property 3).

Figure 4: Closed Semiring Definition

Given:
    G : graph with vertices V[G] and edges E[G]
    w : edge weights
    s : source vertex number

Produce:
    distance[v] : weighted distance from source for vertex v
    path[v] : the previous vertex along best path from s to v

```
Bellman-Ford (G,w,s)
    for each vertex v on V[G]
        distance[v] = infinity
        path[v]= nil
    distance[s] = 0
    for i=1 to |V[G]| - 1
        for each edge (u,v) in E[G]
            cost = distance[u] + w[u,v] // ⊙ operator
            if cost < distance[v] // ⊕ operator
                distance[v] = cost
                path[v]= u
```

Figure 5: Bellman-Ford Shortest Path Algorithm

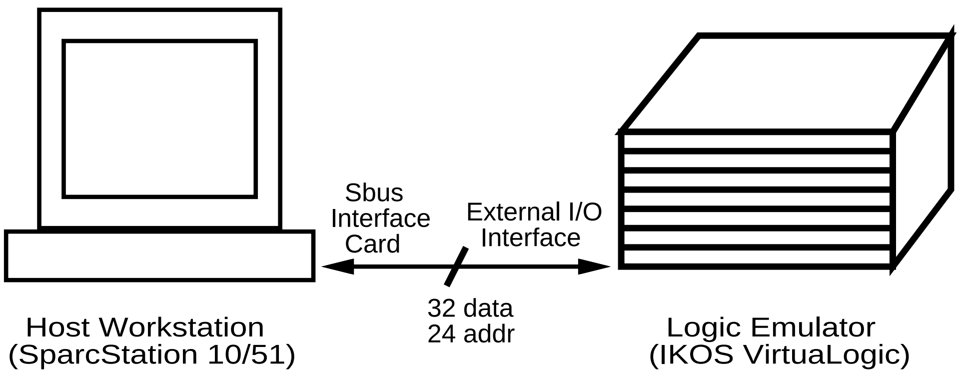

Figure 6: Prototype Reconfigurable Computing System

The prototype hardware system consists of a VirtuaLogic Emulator (VLE) from IKOS Systems (formerly from Virtual Machine Works, Inc.) coupled with a Sun SparcStation 10/51 via an S-bus interface card[5] (Figure 6). Not shown is a SCSI interface to the emulator for downloading configurations and controlling clock speed. We are currently using a prototype VLE system consisting of a standalone board with a user array of 64 Xilinx 5210 FPGAs and sixteen 32Kx8 SRAMs, as well as an on-board microcontroller and glue logic. We are in the progress of upgrading to a larger production system, such as IKOS's six board system consisting of six arrays of 64 Xilinx 4013 FPGAs. The FPGAs on each board are directly connected in nearest-neighbor meshes augmented by longer connections to more distant FPGAs. Boards are coupled together with multiplexed I/Os. Additionally, each board has several hundred external I/Os, resulting in total external I/O connections of a few thousand.

When used as a logic emulator, the external I/O interface of the VLE is generally connected to the target system of the design under emulation. For reconfigurable computing, we have instead connected a portion of the external I/O to an Sbus interface card in the host SparcStation. This card provides an asynchronous bus with 24 bits of address and 32 bits of data which may be read and written directly via memory-mapped I/O to the Sparc Sbus. We are successfully operating this interface at nominal rates of 0.5MHz for reads and 1.0MHz for write operations, providing 2-4 Mbytes/sec rates for communication between the host CPU and the FPGAs of the emulator. This rate allows one 32 bit read/write every 100/50 cycles of the 50MHZ host CPU.

6 EXPERIMENTAL RESULTS

We have implemented a DCS generator and constructed an experimental software system out of this generator and other commercial tools. Figure 7 lists the software used for each tool step and rough running times on

Tool Function	Software Used	Rough running times
DCS Generator	New C Program	seconds
Behavioral Compiler	Synopsys	minutes
RTL Compiler	Synopsys	ten minutes
Virtual Wires Compiler	IKOS VirtuaLogic	two hours per board
FPGA Compiler	Xilinx	two hours per board (ten workstations)

Figure 7: Experimental Software System

semiring problem instance	nodes X edges	gate count	FPGA count (4013)	clock rate (MHz)	solution rate (KHz)	software rate special	(generic) (Hz)	speedup vs software	speedup per FPGA
Stc512	512x2051	187K	48	1.47	2.9	5.4	(7.2)	398X	8.29
Ssp16	16x64	44K	14	1.79	112	11K	(9.1K)	10X	0.71
Ssp64	64x256	181K	56	1.14	18	658	(578)	27X	0.48
Ssp64-mesh	64x224	159K	46	1.56	24	758	(641)	32X	0.70
Ssp128	128x515	366K	118	0.78	6.1	77.2	(149)	41X	0.35
Ssp256	256x1140	814K	261	0.34	1.3	12.9	(25)	52X	0.20
Sm16	16x64	156K	36	1.39	87	6.3K	(5.6K)	14X	0.39
Sm32	32x127	310K	90	1.19	37	1.5K	(1.2K)	25X	0.28

Figure 8: Semiring Results

SparcStation 10 class machines. For most experiments we did not execute the last FPGA compile step due to the excessive FPGA compile time requirements. However, FPGA compiles may generally be parallelized over a network of workstations to provide reasonably effective compile times. We have successfully compiled one of the smaller shortest path designs, Ssp16, all the way down to configuration bitstreams, and run the design on an emulation system. The execution of this design has been validated with I/O across the Sbus interface.

6.1 Performance comparison

We implemented both a DCS generator for an FPGA system and a dynamic code generator for a processor-based system. These compilers create specialized instructions for solving semirings with the classic Bellman-Ford shortest path algorithm. In addition to comparing to the specialized processor code produced by the dynamic code generator, we also compared our FPGA results to a generic version based on loop unrolling. Software results are based on execution on a 125 SpecInt SparcStation 10 processor. Figure 8 shows the results for a range of semirings and graph topology. The prefix of the problem specifies the semiring type as defined in Section 4. The following number represents the number of nodes. With the exception of Ssp64-mesh, each graph topology is randomly generated with a maximum node in-degree of eight, and an average in-degree of four. Ssp64-mesh is identical to Ssp64 with the exception that the nodes are arranges in a nearest-neighbor mesh topology. We list gate count and the total number of Xilinx 4013 FPGAs required. We also list the clock rate assuming a 25MHz internal virtual wires clock. To calculate the solution speed, we divide the clock rate by the number of iterations required to reach a solution, i.e., the number of graph nodes. Finally, we compute speedup by dividing the FPGA speed by the best software speed. The table also lists speedup per FPGA, a measure of the relative efficiency of solving the particular problem type on a reconfigurable architecture.

While software code specialization benefits significantly from unrolling the inner loop and de-referencing pointers, further architectural specialization allows parallelization of the entire inner loop, producing speedup proportional to the number of graph edges. In fact, software specialization only improves speed over a generic software solution for small graph sizes. For larger graph sizes, the specialized code grows larger than the instruction cache, resulting in a factor of two slowdown due to I-cache misses. Architectural specialization, on the other hand, provides a consistent speedup of between 10 and 52 for the shortest path problem, between 14 and 25 times speedup for multiplicative shortest path and 398 times speedup for transitive closure over our best software version. These speedups result from the increased parallelization of the inner loop as problem size increases. Figure 9 shows, however, that these speedups do not scale linearly with problem size due to the higher global wiring overhead for larger designs, resulting in slower clock rates and fewer gates per FPGA.

While the speedup of architectural specialization increases as the problem size increases, the efficiency of the

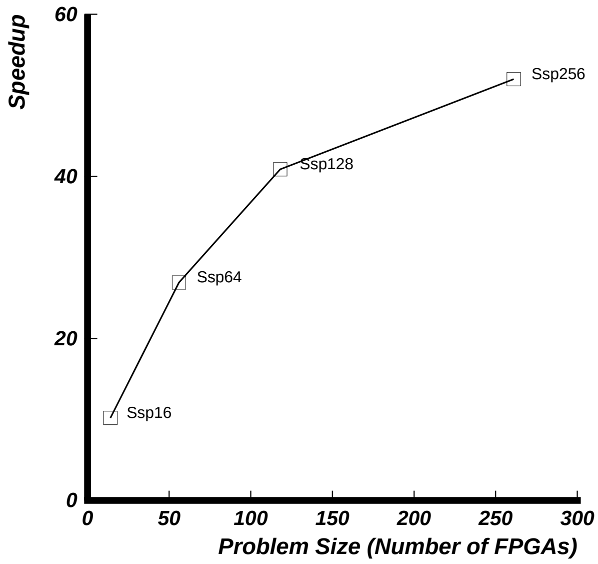

Figure 9: Scalability for Ssp16-Ssp256

hardware solution decreases as shown by the speedup per FPGA column of Figure 8. The reduction in efficiency is due to increasing inter-FPGA communication costs as problem size increases. Ssp64-mesh, a design with much greater locality than the randomly generated graphs in the other problems, has a speedup of 0.70 per FPGA as opposed to 0.48 for Ssp64, a design of similar size but poorer locality. The efficiency of the multiplicative problems is also lower than that of the additive shortest path problems because they require more work per graph edge, while the efficiency of the transitive closure problem is very high (8.29X speedup per FPGA) because the number of gates required per node and edge is very small.

6.2 Structure Area

Structure	Combinational	Sequential	Total
Stc512-Edge	51 gates	8 gates	59 gates
Stc512-Node-2	51 gates	41 gates	92 gates
Stc512-Node-4	65 gates	41 gates	106 gates
Stc512-Node-6	69 gates	41 gates	110 gates
Ssp64-Edge	223 gates	128 gates	351 gates
Ssp64-Node-2	309 gates	172 gates	481 gates
Ssp64-Node-4	543 gates	172 gates	715 gates
Ssp64-Node-6	776 gates	172 gates	948 gates
Sm32-Edge	1523 gates	128 gates	1651 gates
Sm32-Node-2	304 gates	161 gates	465 gates
Sm32-Node-4	537 gates	161 gates	698 gates
Sm32-Node-6	769 gates	161 gates	930 gates

Figure 10: Sample Structure Sizes

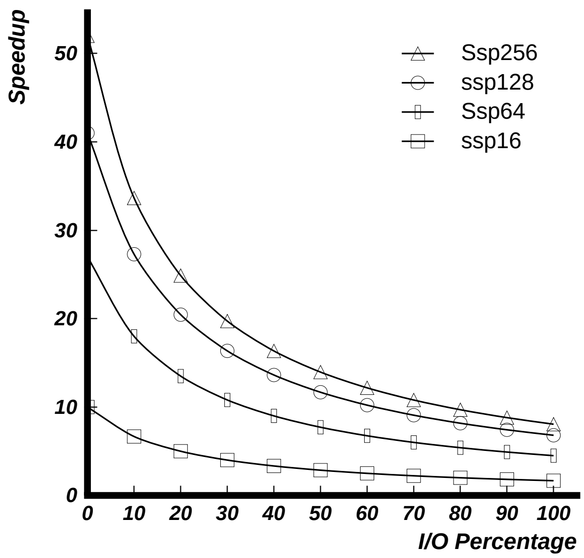

Figure 11: I/O Effects

Figure10 shows some resulting combinational and non-combinational gate count details for the generated structures. Each node is specialized to the exact degree of in-edges needed and nodes and edges are specialized to the semiring problem being solved. The Ssp and Sm nodes are about the same size, around 700 gates for nodes of degree 4, while the Stc nodes are much smaller at only 106 gates for degree 4. The edge sizes varied proportional to the complexity of the operators inside, 59, 351, and 1651 gates for the and, add and multiply operations used by Stc, Ssp, and Sm respectively.

6.3 I/O effects

This section analyzes the effect of I/O between our host workstation and FPGA array. Let us first consider the use of an algorithm such as shortest path within our application domain. Shortest path is typically the inner loop of a higher-level algorithm such as the Edmonds-Karp maximum flow algorithm. In this maximum flow algorithm, a shortest path is determined and then flow is scheduled along that path. Only one path is needed, so the best path for all nodes does not need to be read for each call to the shortest path algorithm. Furthermore, only the values of edge weights along that path will change, thus not all the edge weights need to be written. To study these effects for our system, we model *incremental* I/O as a fraction, I, and compute speedup across the range of I. I is the fraction of edges written and paths read in solving a single path instance. Given this model, the total solution time for a particular call to shortest path is computed as $(I \times edges + nodes + I \times nodes)$ cycles. I.e a percentage of the edge weights are written, the algorithm is executed in node cycles, and then a percentage of the node weights are read. Figure 11 shows the relative speedup for the shortest path semirings as I is varied. We assume single emulation cycle reads and writes, although this may vary as the clock rate of the emulator fluctuates above and below the interface transfer rates. While speedup is still achieved at high values of I, the I/O overhead is high, resulting in roughly a factor of 5X slowdown when $I = 100\%$ for the graphs we study. These results show that future research in I/O will have a significant impact on performance.

7 CONCLUSIONS AND FUTURE WORK

We have introduced dynamic computation structures, a compilation technique to automatically map specific problem instances into FPGA hardware. We implemented a DCS generator for closed semiring applications, including shortest path and transitive closure, with resulting speedups of one to two orders of magnitude versus a conventional microprocessor-based workstation. These speedups where obtained within a software framework which automatically maps the high-level behavioral verilog output by our generator onto hundreds of FPGAs without user intervention.

While our system is fully automated, it is still bottlenecked by FPGA compilation time (although parallel compilation already reduces compile time from days to hours). In circuit design compilation and synthesis time are secondary to optimization area/timing results. However, for reconfigurable computing compilation time is a primary concern. Dynamic computation structure are useless if they can not be compiled quickly. While improvements in FPGA compilation technology will directly affect compile time, we are also addressing this issue by developing a relocatable hardware library as a key input to the behavioral compiler. This library will consist of pre-compiled computation macro-blocks and will reduce synthesis time. Example macros include adders, multipliers, registers, and parameterized datapaths. These macros will be compiled down to the FPGA-bitstream level ahead of time, and prototype software to translate and stitch bitstreams together is under implementation.

In addition to speeding compile time, we will be addressing language issues for expressing computation structures as well as operating system approaches for securely multiplexing reconfigurable hardware among multiple user-level specializations. While emulation hardware is useful for targeting the application-specific portion of the user-level hardware, we will be implementing a prototype with both FPGAs and an on-board processor to further explore the operating system issues in the new design realm.

8 ACKNOWLEDGMENTS

The research report in this paper was funded in part by ARPA contract # DABT63-96-C-0036 and by an NSF Presidential Young Investigator Award to Prof. Agarwal. Matthew Frank is in part supported by an NSF Graduate Fellowship.

9 REFERENCES

[1] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142–151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, January 1993.

[2] T. Bauer. The Design of an Efficient Hardware Subroutine Protocol for FPGAs. Master's thesis, EECS Deptartment, MIT, Department of Electrical Engineering and Computer Science, May 1994.

[3] P. Betrin and H. Touati. Pam programming environments: Practice and experience. *Napa*, pages 133–138, April 1994.

[4] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, 1991.

[5] A. Dehon and S. Perentz. Transit Note No. 67: Transis Sbus Interface. Technical report, Artificial Intelligence Laboratory, MIT, June 1992.

[6] D. R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *PLDI '96*, 1996.

[7] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1), Jan. 1991.

[8] IKOS Systems, Inc. *VirtuaLogic Emulation System Documentation*, 1996. Version 1.2.

[9] R. R. T. Cormen, C. Leiserson. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1992.