JOURNAL OF SPACECRAFT AND ROCKETS
Vol. 39, No. 4, July–August 2002

Field-Programmable Gate-Array-Based Graph Coloring Accelerator

L. M. Pochet,* M. L. Linderman,† S. L. Drager,‡ and R. L. Kohler§
U.S. Air Force Research Laboratory, Rome, New York 13441

A hardware methodology is described for implementing a graph coloring for the Latin squares problem that is compatible with more course grain approaches routinely implemented in software. The approach described maximizes the use of local communication and fine-grained parallelism while still ensuring a complete search of the solution domain. An implementation of a graph coloring architecture using field-programmable gate arrays and high-level programming tools is presented. An exploration of the tradeoff among nodes per processor, fill depth, and latency is presented. The use of this hardware-based graph coloring accelerator architecture to the more efficient implementation of routing for wave division multiplexing fiber optic communications systems and multihop radio communications is also discussed.

Introduction

THE graph coloring accelerator architecture was developed to improve the speed and efficiency with which Latin-square problems could be solved. Specifically, the targeted problem was wave division multiplexing routing in fiber optic communications systems. The graph coloring accelerator architecture was implemented on a field-programmable gate array (FPGA). FPGAs allow a user access to most of the benefits of application-specific integrated circuits (ASICs) without the costs and long turnaround time. Targeting an FPGA allows the designer to experiment with the accelerator architecture. The designer is able to vary architecture-dependent parameters and to test the impact of the changes on the actual hardware in a matter of hours instead of the weeks required to spin new ASIC designs. For example, the node processor cache size may be varied over some range to determine the optimum cache size based on results from the physical hardware. This mode of experimentation can also be used to evaluate different implication rules. A Latin square of order N comprises an $n \times n$ array of N symbols in which every symbol occurs exactly once in each row and column of the array.[1] The Latin square involves the use of two sets of blocks, one of which is organized by rows and the other by columns. Latin-square designs have the character of a double randomized block design, where the experimental variable is confounded with the row and column interactions. Figure 1 is an example of a Latin square.

Problem Description

Completing a Latin square is a nonpolynomial (NP) complete task that has been shown to take considerable compute time on large ($N > 30$) problems.[2] The time required to complete a Latin square is dependent on N and on the number of preset nodes. Preset nodes in a Latin square correspond to preexisting requirements on a scheduling system, such as wave division multiplexing. A relatively sparse graph with few presets is likely to have many solutions and will be solved relatively quickly. A graph with a large number of presets will likewise be solved or proved inconsistent quickly because there are few degrees of freedom remaining for the solution. The solution of a partially colored graph will result in a valid assignment of symbols to nodes or a determination that no such assignment exists.

Empirical evidence suggests that the number of backtracks, and thus the number of incorrect guesses, reaches a maximum when approximately 40% of the nodes are preassigned.[2] A backtrack occurs when a contradiction is arrived at; after the contradiction is discovered, the guess that caused the contradiction is undone.

The maximum number of nodes that may be preset in a Latin square is 50%. After the 50% preset point is reached, all other nodes may be implicated. Implication takes place when a node has only one valid color remaining available to it. The node then selects this color to be its selected color until a backtrack is required. Furthermore, as the number of preset nodes approaches 50%, the graph may also be solved relatively quickly. This is because there are fewer combinations to try before a solution is found, or it is discovered that no solution exists.

Local search methods are usually able to solve partially colored graphs faster due to the increased parallelism available. However, local search methods cannot assure that all possibilities have been tried. This presents a problem because local search methods are not able to tell when there is no possible solution to a graph. On the other hand, global search methods assure that all possibilities are tried. However, global search methods are generally slower because less parallelism can be exploited.

Latin squares can easily be made parallel in the course-grain sense. To make the system parallel over five processors, we may simply assign a different color for the first node in each instantiation in parallel, then solve the system assuming that as an initial condition. Our fine-grained implementation makes the computation parallel at a lower level, while allowing us to make the system parallel at the courser grain as well.

Related Work

Several architectures have been developed to apply FPGA technology to accelerate combinatorial search. Most attention has been paid to the problem of Boolean satisfiability (SAT), a classical NP-complete problem with many applications. Typical FPGA-based approaches start by assuming that SAT problem is expressed in a canonical form, for example, three-term clauses, to simplify the FPGA architecture and often to optimize the mapping onto the FPGA.

One approach compiles the FPGA to solve a specific problem instance.[3] This approach attempts to optimize execution speed, but it must first compile the instance before solving it. For larger problems,

Received 28 February 2001; revision received 15 January 2002; accepted for publication 12 April 2002.

*Electronic Engineer, Advanced Computing Architectures, Information Technology Division, Information Directorate, 26 Electronic Parkway; Louis.Pochet@rl.af.mil.

†Electronic Engineer, Systems Demonstration and Integration, Information Systems Division, Information Directorate, 525 Brooks Road; Mark.Linderman @rl.af.mil.

‡Technical Advisor, Advanced Computing Architectures, Information Technology Division, Information Directorate, 26 Electronic Parkway; Steven.Drager@rl.af.mil.

§Electronic Engineer, Radar Signal Processing Technology, Radio Frequency Sensor Technology Division, Information Directorate, 26 Electronic Parkway; Ralph.Kohler@rl.af.mil.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \\ 3 & 2 & 1 & 0 \\ 2 & 0 & 3 & 1 \end{bmatrix}$$

Fig. 1 Latin square of order four.

this would seem to be a good tradeoff because the compilation speed grows slowly with problem size compared to the execution time, which grows exponentially. A difficulty with this approach is that automated tools for placement and routing often generate poor results that lead to very slow clocking rates and circuit utilization. Others avoid most of the placement and routing efficiency and compilation time by tailoring computational blocks that have been already placed and routed.[4] The architecture presented in this paper requires neither compilation nor tailoring, allowing successive problems to be streamed through the hardware without any reconfiguration.

Fine-Grain Implementation Algorithm Architecture

There were two design goals for the graph coloring accelerator architecture. First, the architecture must be scalable for problem sizes of up to at least $N = 40$. Second, the algorithm must exploit the maximum degree of parallelism to take best advantage of the underlying hardware.

The principal contribution of this paper is the implementation of a parallel implication mechanism on an FPGA. Most approaches to combinatorial search differ in how values are assigned (guessed) to variables and in what order. Once this step is performed, the implications of the guess are computed. Several search heuristics for variable assignment for the Latin-square problem are discussed in Ref. 5. Because our architecture implements the guessing procedure in the master node, only this node needs be modified to implement another search heuristic.

FPGA Architecture

FPGA architecture comprises configurable logic blocks and configurable interconnect. Configurable interconnect is optimized for local routing and performs poorly when used to span long distances, as when used for global communication. Configurable logic blocks (also known as slices) can be used to implement small logic functions. When larger functions are implemented, multiple slices and configurable routing must be used, resulting in slower computation. The algorithm developed was tailored to match the underlying FPGA architecture.

Algorithm Overview

A high-level flow diagram of the graph coloring algorithm process is shown in Fig. 2. The algorithm works by exploring the search space starting from an initial partial coloring of the nodes. Through a recursive sequence of guesses, implications, and backtracking, either a solution is found or it is proved that no solution exists.

The state of the graph comprises the states of the nodes of the graph. The state of a node is exactly one of preassigned, guessed, and unassigned. Nodes in either of the first two states have been assigned exactly one color. The color was assigned to the node by the initial conditions, subsequent guesses of previously unimplicated nodes, or implication of previously unassigned or guessed nodes. Unassigned nodes may still be assigned one or more colors based on the implication rules implemented. The set of colors that may be assigned to an unassigned node comprises the set of all colors minus the colors that have been proved to contradict the color assignments to the preassigned and guessed nodes.

The implication rules are a specified function over the state of the graph that monotonically decreases the size of the color sets of the unassigned nodes. The function is applied iteratively until the function no longer changes the state of the graph or a contradiction is detected. An unassigned node indicates a contradiction when its set of possible colors is reduced to the empty set. In the absence of a contradiction, a stable state S^* is reached when further application of the implication function I does not change the state of the graph, that is, $I(S^*) = S^*$. The implemented algorithm waits until the graph reaches a steady state before another guess is made, although this is not generally a requirement.

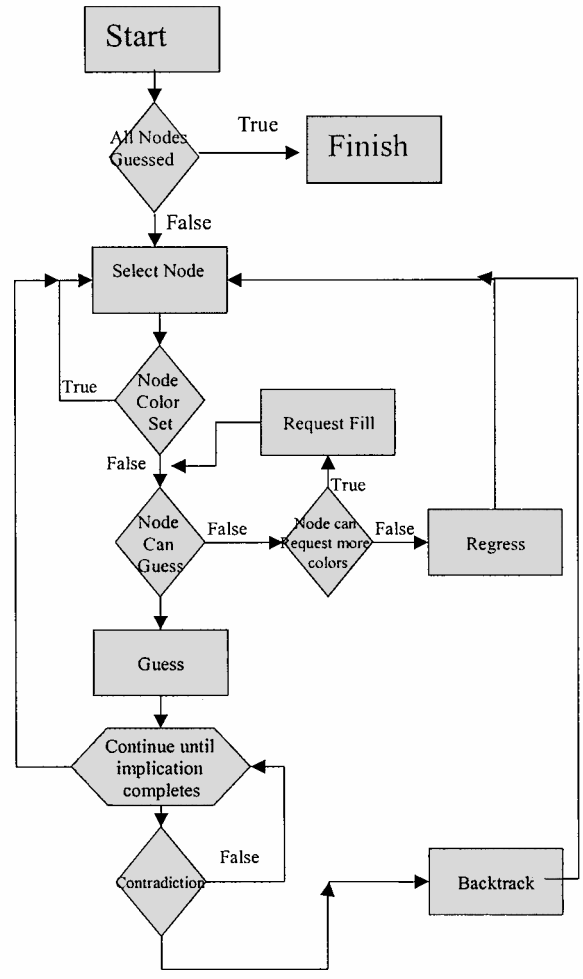

Fig. 2 Flow diagram of graph coloring algorithm.

Our implementation assigned colors to the nodes in a fixed order. Although fixing the order is not generally necessary, it simplifies the control logic in two ways. First, it makes recursion simple by predetermining the order in which we will check color assignments for each node. Second, it gives us a simple method of determining when we have completed a square by checking the status of the first square. If the first assigned node has no valid color assignment, then no solution exists. Other local searches do not make this assumption. Although this often results in a faster solution to the problem, it makes it more difficult to prove that no solution exists because it is difficult to prove that all possibilities have been tested.

If a contradiction is detected, the last guessed node is assigned a different color, and implication is performed. If a contradiction is again detected, the node is assigned yet another untried color, and the process is repeated. This process is referred to as backtracking. If a stable state is reached, then the search recurs one level deeper as an unassigned node is selected and assigned a color. If all possible colors have in turn have been assigned to a node, and each has resulted in a contradiction, then the algorithm backtracks up to the previous level of recursion. We call this extended backtrack a regression.

Our implication function is computed at each node by removing from its set of possible colors any color assigned to another node in the same row or column. Additional implication rules (currently unimplemented) include detection of colors excluded from all nodes within a row or column save one. There are several others implication rules that could be implemented to allow nodes to implicate sooner, but this would increase the complexity of the node processor. The architecture of the FPGA requires a tradeoff between computational complexity and number of nodes that can simultaneously be implemented because the computational resources are fixed for a given FPGA architecture.

Like the related problem of transitive closure, the order of implication among the nodes does not affect the final state, and, therefore,

the implication of one node can occur in parallel with implications of others. If a contradiction exists, only the location of the contradiction may change, not the occurrence of one. In the absence of a contradiction, the state of the nodes after implication is the same regardless of the order of implication.

Implementation Description

An FPGA architecture was developed comprising an array of $(N \times N)$ small node processors for each node of the Latin square, an edge controller for each row and column of the graph, and a graph master to manage host interaction and the guessing process. Each edge controller keeps a last in, first out (LIFO) list of guesses and the implications caused by the guesses. Use of the guess LIFO simplified backtracking by enabling the edge to step back in time to a state where no contradictions existed. Figure 3 shows a high-level schematic of the architecture, where circles represent edge control, rectangles represent node processors, and the graph master is represented with a triangle.

The communication mesh is a simple two-dimensional toroidal mesh of unidirectional busses. These busses comprise a tag identifying the data being transferred and a data bus $\log_2 N$ bits wide. There are two bus cycles. On the first cycle (referred to as the column focus), the nodes listen for communications from the column-edge processor. At the same time, node processors can send information to the row-edge processor. The second-cycle (row focus) node processors receive from the row-edge processor and transmit to the column-edge processor. The cycles are synchronized so that the node receives data from the row processor on one cycle and the column processor on the other. This systolic data flow gives the appearance of continuous communication with an edge processor. Edge to node communication consists of remove, backtrack, and guess broadcasts, as well as memory loads directed at individual nodes.

The choice of bus widths and communication methodology was critical to the performance and scalability of the system. All busses were chosen to be ring busses to minimize the communication latency and simplify the communications.

We chose to make the edge busses N bits wide, to move color information quickly, because the number of edge nodes scales linearly with N. In this representation, each wire indicates a separate color, so that we can send information about all colors in a single cycle with each message. Most of the edge communication consists of messages detailing message fill calculations and backtrack control. Memory fill is the term that describes the processor nodes need for information on what other colors are currently in use in its row and column. This bus is used to have each edge node communicate with every other edge node.

We chose to make the processor node busses $\log_2 N$ bits wide because the number of processor nodes scales with N^2. The resulting architecture balances the need for small processor nodes with the need for fast communication, especially among the edge nodes, to mitigate bottlenecks in communication. This bus is used for passing the results of implication requests and making and filling requests for information about the current row and column. This choice of bus architectures is physically well suited to our target architecture and works both to keep the physical size of the system reasonable and to maintain performance.

The bus that connects the edge processors is comprises a tag and an N bit wide data bus. Edge to edge communication consists of node processor memory fill calculations, as well as guess and backtrack control. Using an integer representation would have required one cycle for each color transmitted to the edge controller, creating the fill packet to be sent to the node processor. A representation where each color is represented by a single bit (one hot) allows all available colors to be transmitted in one cycle; it also simplifies the union operation required to create a fill packet. To create a fill packet, the edge controller must perform a union of colors available to the node from its row controller and its column processor.

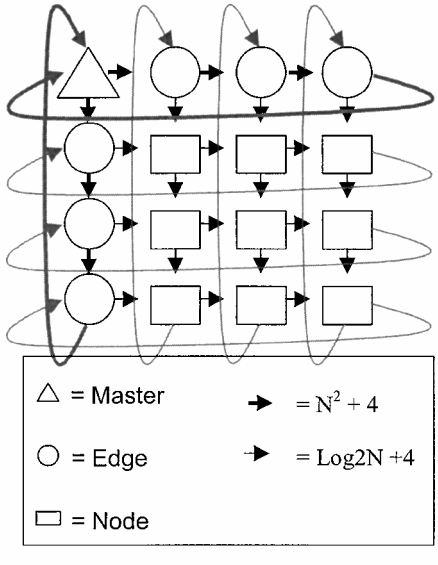

Fig. 3 Top view of architecture implemented.

Component Detail

This section describes in detail each component (node processor, edge controller, and graph master) of the Latin-square coloring architecture. It will be shown how design decisions support the overall direction and evolution of the design.

Node Processor

The node processor is a minimal processor able to request memory fills, implicate when only one color remains in its set of possible colors, backtrack when its set of possible colors is empty, guess a color from its memory, and remove colors from its local list. It is connected to adjacent nodes through a unidirectional node bus that starts and ends at the edge controller. A simple view of the node processor can be seen in Fig. 4.

Bins are used to hold the current subset of possible colors available to the node. Each bin holds a color and a bit indicating whether the bit is valid.

An implicate/backtrack controller counts the number of valid bits and requests an implication when the complete list flag is thrown and there is only one valid color remaining. It requests a backtrack when the complete list flag is thrown and there are no colors remaining in the bins; this indicates there are no colors available for the node.

As described earlier, there are two parts to a communication cycle, column focus and row focus. During the column focus stage, information sent from the column controller arrives at the node and is processed; the node also sends information toward the row processor. On the other half of the cycle, the opposite happens, data are retrieved from the row controller and sent to the column controller. When the node bus is idle, a node is free to send requests on the node bus; otherwise, the input is passed through to the next node on the node bus. Requests written to the node bus must first pass through all nodes below (or to the right of) the sending node before arriving at the edge controller.

The most important consideration during the node processor's design was scaling. Graphs too large to fit on a single FPGA will be time multiplexed with partial row column blocks completed in a windowing technique to be described later. Each node processor, and the required interconnect between node processors, grows as a function graph size grows. Effort was made to minimize the impact of graph size on node processor size.

The interconnect between each node processor and from each edge processor to the adjacent node processor grows $\mathcal{O}[\log_2(N)]$. Colors are sent one at a time across the node bus. The most demanding task for the node bus is the memory fill, which is a constant value regardless of the size of the graph and requires multiple colors to be sent serially. Node processors require memory fills to update their bins with colors available to the node when the current set stored locally has been exhausted through remove operations. For remove requests, only one color, the color to be removed, is sent.

Node processor memory (also known as a bin) also grows as the graph size increases. Node memory holds a subset of the complete list of available colors. This subset contains M colors, where M

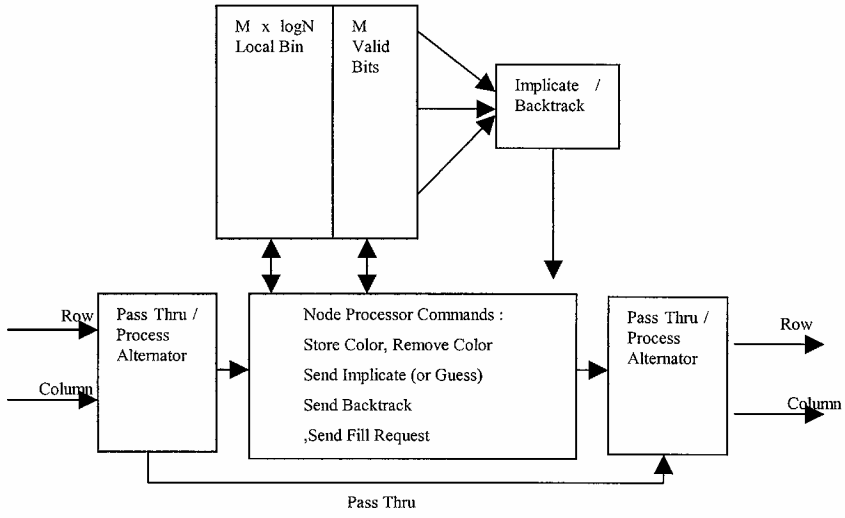

Fig. 4 Functional diagram of a node processor.

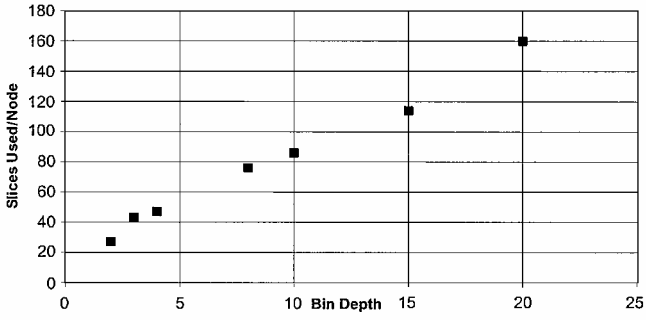

Fig. 5 Bin size scaling with *M*.

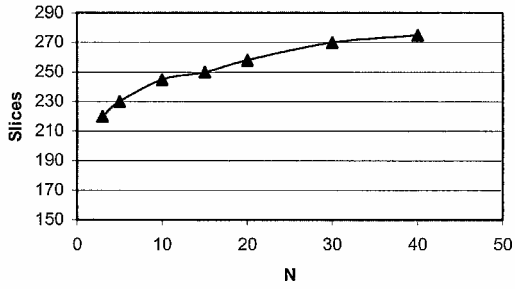

Fig. 6 Node size with respect to *N*.

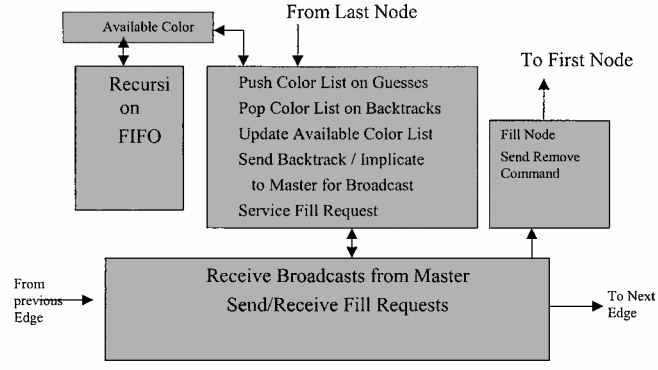

Fig. 7 Edge controller overview.

is less than or equal to N. Memory bins are analogous to local cache used in general purpose processors. Each node also holds a flag indicating whether the list of M colors is a complete list of all colors available for this node, given the constraints for that row and column, or a partial list of possible colors. As colors are removed from the list due to implication, more are requested from the edge controllers. This paging process involves both row and column controllers. The set of colors available to the node requesting data is the intersection of colors available to the row and column. Each edge controller contains a list of colors available to the row or column it controls. A row controller receives a fill request and sends a list of available colors to the column controller. The column controller performs a intersection of the two sets and sends the first M colors to the requesting node serially through the nodewise bus. Complete is asserted when are no colors are left to be checked.

As node size increases, there can be fewer nodes on a given sized FPGA. Eventually, node size increases to a point where there are not enough nodes on a given sized FPGA for useful computation. One solution to this problem is decreasing the amount of logic required for node memory bins by decreasing bin depth, M. For example, if M is set to 7 and node size becomes prohibitive because a large Latin square is being attempted, M can be decreased to 4 to decrease the size of each node. The penalty for this reduction is more numerous and frequent fill requests. Total memory requirements for each node are $M \times \log_2(N)$. The effect of M on bin size is shown in Fig. 5. As N increases, M can be decreased to maintain a constant node size. Decreasing M incurs an increased overhead due to more frequent paging requests from empty node processors but allows more control over the size of each node processor. Total node size as a function on N as M is kept constant is shown in Fig. 6.

The bins contain the only information associated with the state of the node processor. If filled with different data, the same processor could be used to represent a different node. When coloring a graph too large to fit on a single FPGA, a windowing scheme will be used to clean the memories of each node processor and refill them with colors specific to a different grouping of nodes. Nodes with their colors already set (implicated or guessed) must also be noted and restored when windowing.

Edge Controller

Two edge controllers supervise each node processor. Each edge controller supervises N node processors. One edge controller governs the row information, and one governs the column information. For a given problem, $2N$ edge controllers are required. Each edge controller keeps a list of colors available to the node processors in its row or column. This list is N bits long, and each bit represents a color. As a color is removed from the list, the corresponding bit is changed to 0. When the color is added back, it is returned to a 1. The edge controllers also keep a stack of previous lists; each time a guess is made, the previous list is pushed onto the stack. When a regression is requested, the old list is popped off the stack and returned to the edge controller list. A high-level schematic of the edge processor is shown in Fig. 7.

The edge controller's primary function is to handle fill requests from node processors. This action requires input from both edge

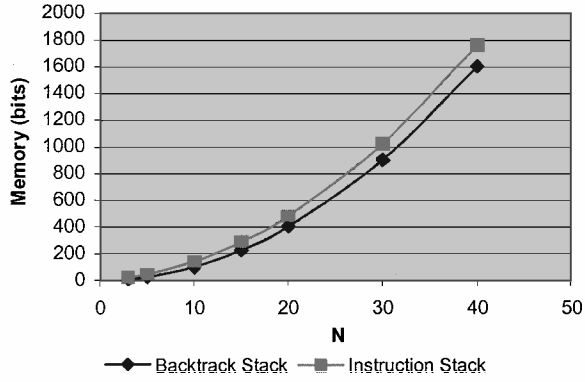

Fig. 8 Edge node memory requirements, scaling with N.

controllers that supervise the requesting node. The side edge sends the current list of available colors to the top edge through the edge bus that consists of an N-bit wide bus and a data tag. When the top edge receives the list, it obtains the intersection of its list of possible colors and transmits a fill packet to the requesting node containing up to M colors, as well as a flag indicating if the list transmitted is complete or partial. Implication can be performed on a complete list once only one color remains in memory. Implication cannot proceed on a partial listing of partial colors even if there is only one color remaining in memory.

When windowing to color large graphs, the color stack and the current color list must be stored for each edge controller windowed off chip. After the edge controller's data are stored, the color stack and current list for a new edge are passed in to the edge controller's stack and current free color list. This operation proceeds much as a context switch.

The top edge controllers play only a small part in the setting of initial conditions. The first fill of each node processor is started at the graph master, not the side edge. The graph master sends a color list consisting of one color to the top edge. The top edge treats this as a normal fill request and sends a list consisting of one color to the node. The node immediately implicates to the color received.

There are two memory stacks in each edge processor: the instruction stack, which is able to store instructions from node processors, and the backtrack stack, which stores all previous states. Node processor instructions are $N+4$ bits wide and each node processor can issue one instruction at a time. The size of the instruction stack is $N \times (N+4)$. Memory requirements for the edge are detailed in Fig. 8.

Graph Master

The graph master handles all host-FPGA communication. With migration to different FPGA platforms, only the graph master must be changed. Host communication consists of obtaining initial conditions and reporting results.

The graph master also handles all guessing and backtracking. Currently, when the graph master counts a set number of idle cycles on the edge bus, another guess command is sent. A guess command consists of a guess command tag and the row/column of the node to make a guess. Ultimately, it is preferable to detect when the implication process is complete rather then wait a fixed number of cycles. Note that it is not necessary that implication proceed to a stable state before assigning a value to a previously unassigned node.

The only backtrack control required of the graph master is to ensure that every guess causes a maximum of one backtrack. If multiple contradictions are caused by a single guess, only one backtrack is required to return to the previous valid state.

The graph master, having the ability to communicate to off-chip memory, also controls the windowing process to color large graphs. This control involves finding all nodes with set colors, storing all data from the edge controllers windowed out, loading new edge controller data, and filling all nodes with their new lists and preset colors.

Results

The described design has been targeted for an Annapolis Micro Systems WildStar board populated with Virtex 1000 FPGAs containing one million gate equivalents and 32, 4000-bit deep block ram units. A graph where $N=6$ requires 36 node processors, 12 edge controllers, and 1 graph master was implemented on one of the three available FPGAs. Almost 6000 slices and 12 block rams are required for this implementation. Xilinx backend tools report a maximum clock speed of 40 MHz. Because of the reliance on local communication, the maximum clock frequency is effected only slightly by changes in N and M. The critical path for clock frequency currently resides in the node processor.

One effective measure of Latin-square completing architectures is the number of implications that can be performed per second. Each node has the ability to implicate independently and simultaneously. Realistically, because remove and guess commands must move through the nodes systolicly, a guess causing complete implication of the square will take $2N$ cycles to propagate through N^2 node processors. In a graph of $N=6$, an n upper bound of 36 implication steps can be calculated in 24 clock cycles, resulting in 60 million implications per second.

As a measure of the relative performance of our implementation to that of a reference implementation running in straight JAVA, an engineer calculated that the implementation was running at 100 implications per second. Our implementation runs at 60 million implications per second on a 6×6 Latin square with a 40-MHz clock rate on a Virtex 1000 chip.

The graph coloring accelerator can be used to color a graph of any preset without resynthesizing the design circuit. This is achieved by allowing the graph master to preset each node immediately following a reset signal with presets found in off-chip memory. All nodes are preset with either one or N colors. Presetting to N colors equates to no constraint on the color of the node, whereas a one color preset equates to a hard requirement that must be met and cannot be backtracked or undone.

To allow for a windowing system to be implemented, accommodating problems too large to fit on a single FPGA, initial conditions of the Latin square, presets, are broadcast before calculation begins. Presetting a graph of $N=6$ consumes 140 clock cycles, due to a two clock cycle latency added at each node processor and edge controller for data passing through. There is also an $N+10$ cycle latency for preset information from an edge to the start of the node controller row or column, due to latency in the edge controller first in, first out (FIFO) stack catching information to send to node processors. This equates to a 0.0035-ms overhead for an $N=6$ graph.

A key component of this architecture is scalability. The node processor scales extremely well as N changes. Additional control over the scaling of N was added in the form of a variable sized local memory. As the size of the node processor increases due to increased N, local memory size can be decreased to decrease node processor size. Edge processors contain both current state information and all previous state information. The memory requirements of the edges scale with N^3. The large quantity of block ram on Virtex FPGAs is ample for even large graphs.

Currently, a naive guessing scheme is employed. When all implications possible have been completed, a guess is made. Another guess is made when all node busses have been idle for the longest possible propagation time from one node to another. This time is currently 140 clock cycles. The wait time is the worst-case cycle count for node bus activity to propagate back to the graph master. The graph master waits for all node busses to be idle for 140 clock cycles before ordering another node to guess. Therefore, approximately 4200 clock cycles are required to color an empty graph where there are no contradictions or backtracking. For each graph colored, there is an overhead of approximately 4200 clock cycles.

High-Level Software Tool Suite

The JAVA hardware description language (JHDL)[6] was used to design the graph coloring accelerator architecture for FPGA implementation. JHDL allows greater flexibility in the parameterization of models than conventional very high-speed integrated circuit (VHSIC) hardware description language (VHDL). This parameterization flexibility was necessary to allow constructs to change as the size of the problem changes. Architecture refinement is also simpler than when using the more behavioral VHDL. The structural basis

```
for(int i=0;i<N;i++){
  GoingRight[i]=wire(N+TagWidth,"RightWire");
}
for(int i=0;i<N;i++)
{//create top edge components
  GoingDown[i]=wire(NLog2+TAGWIDTH,"HangingWire");
 GraphTopEdge GTE = new GraphTopEdge(this,Reset_n,
 GoingRight[i],TopHanging[i],GoingRight[i+1],GoingDown[i],i,0,("TopEdge"+i));
}
```

Fig. 9 Code fragment showing parameterizability of JHDL.

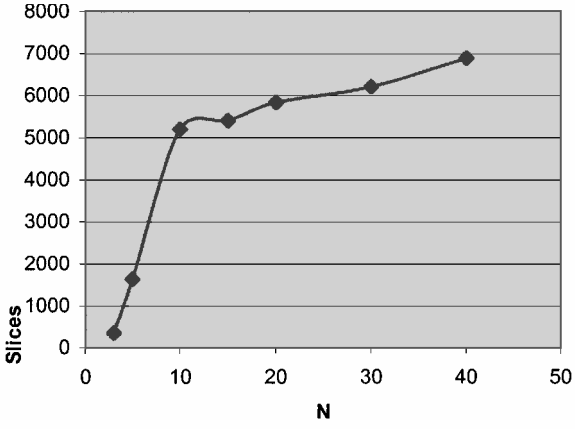

Fig. 10 Number of slices for implementation of architecture.

of JHDL allows a designer to know exactly what hardware will be instantiated from the code written. JHDL minimizes or avoids common problems, such as differences between simulation and synthesized design often experienced with other HDLs when migrating from simulation to an actual chip. Finally, JHDL allows the full readback of the FPGA to be annotated on the circuit schematic. This allows simulation like debugging to take place on both the target hardware and on the simulator, as needed.

The use of the high-level tool suite permits a software description that defines the application in a completely parameterizable manner. The same code that generates the hardware to solve an order 4 Latin square generates the hardware for an order 20 Latin square, with no changes to the software. This approach has three advantages:

1) It allows us to describe hardware in a high-level language and makes us much more productive than with traditional approaches.

2) It allows us to explore the design space much more fully than we could have with traditional approaches because changes are easy to make and compile.

3) It allows us to optimize the solution to a given problem very quickly.

The code fragment shown in Fig. 9 controls the instantiation of the edge controllers. The instantiation relies directly on N for wire widths and the number of controllers created. The ability to also parametrically hand place the edge controllers (not shown) is based on this same approach. Dramatic clock speed increases are achieved by explicitly placing controllers, nodes, and lower level blocks.

Figure 10 shows the total number of slices required for the implementation. Whereas any particular implementation will run only one size of Latin square, because the software is parameterized, we can run very large instantiations on a single FPGA. Alternately, as chip performance characteristics improve, we can replicate multiple Latin squares solvers on a single FPGA and use the same course-grained parallelization techniques that are available to software only solution approaches.

Applications of Latin Squares

The ability to complete Latin squares is useful in many scheduling and routing applications. One such useful application of the Latin square is the routing of wave division multiplexing (WDM) fiber optic communication systems. These passive, all-optical systems allow up to N wavelengths to be applied to each input without any

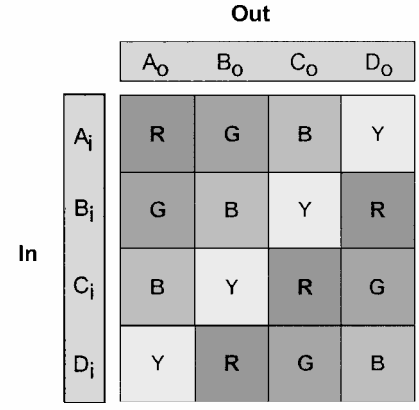

Fig. 11 Latin square used to route a WDM communications system.

output contention.[7] WDM routing systems also operate with a much lower latency added at each routing node. Optical systems that are routed with the use of Latin squares are referred to as Latin routers.

In a Latin router, each row represents a separate routing node. Each routing node is able to accept N different input wavelengths simultaneously. Each of these input wavelengths is represented as a color in the Latin router. Each column represents the destination of the color being routed.

Initial constraints used for a Latin square reflect preexisting requirements or limitations of the network being routed. Communications that must be funneled through specific nodes for security or topographical reasons appear as initial constraints. As stated earlier, Latin squares tend to increase in difficulty as the number of initial constraints increase.

In Fig. 11, there are four separate routing nodes shown, A_i, B_i, C_i, and D_i, each node able to route four input wavelengths simultaneously, R, G, B, and Y. Each node routes each distinct input to exactly one output A_0, B_0, C_0, and D_0 represented by the column that contains the input wavelength. For example, input wavelength G, when applied to A_i is routed to B_0. Solutions to the Latin-square problem have also been applied to time division multiple access scheduling in multihop packet radio networks[8] and high-performance asynchronous transfer mode switch architectures.[9]

WDM over optical intersatellite communication links have been studied for a variety of satellite constellations.[1] Optical satellite networks present several challenges:

1) Broadband traffic patterns exhibit peak intersatellite link (ISL) loads several times greater than average ISL loads.

2) Source-to-destination traffic patterns vary continuously as a function of time of day at both the sources and destinations.

3) Usage patterns are evolving over time, thus limiting our ability to predict future loading.

4) Connectivity of interorbit ISLs changes constantly, if deterministically.

Despite these challenges, suggested means of waveform allocation are static in nature. As an example, one suggested scheme called "matrix,"[2] adapted from terrestrial networking, assigns unique wavelengths to incoming and outgoing traffic depending on row and column (or orbit index and satellite index within an orbit). This allows reuse of wavelengths (unlike traditional single-hop architectures) and admits simple waveform assignments that guarantee that a wavelength is used at most once in a row and column, indeed, the principal requirement of a Latin router. Indeed, a suggested solution for allocation of wavelengths is identical to the intuitive solution to an unconstrained Latin square: for node (i, j) within an $(n \times m)$ the matrix is $[(i + j - 1) \bmod N]$, $i \in 1, \ldots, n$, $j \in 1, \ldots, m$.

Static allocation schemes, however, invariably lead to underutilized resources. Such allocation schemes do not spread network load dynamically as the underlying network topology evolves and the peak usage shifts from one minute to the next. In this situation, ideally allocations to existing ISL connections would remain fixed while old connections are dropped and new ones added. The existing ISL connection wavelength assignments comprise the Latin-square

preassignment, dropped or unallocated wavelengths are unassigned variables, and the new connections are assigned wavelengths when the constrained Latin square is solved.

Conclusions

This paper has presented a scalable hardware methodology suitable for implementing graph coloring of Latin squares in FPGAs. The approach maximizes the use of local communication and fine-grain parallelism, ensuring a complete and most efficient solution of the problem space. The graph coloring accelerator architecture has been demonstrated for Latin squares of $N = 6$, but requires the implementation of a windowing scheme to prove scalability to $N = 40$.

Acknowledgments

The Air Force Office of Scientific Research sponsored this research. The authors would like to thank Peter Athanas, Mark Jones, and the Virginia Polytechnic Institute and State University Configurable Computing Laboratory for their support and advice.

References

[1]Denes, J., and Keedwell, A. D., *Latin Squares and Their Applications*, Academic Press, New York, 1974.

[2]Gomes, C., Selman, B., and Crato N., "Heavy-Tailed Distributions in Combinatorial Search," *Principles and Practices of Constraint Programming*, Vol. 1330, Lecture Notes in Computer Science, Springer-Verlag, Linz, Austria, 1997, pp. 121–135.

[3]Platzner, M., "Reconfigurable Accelerators for Combinatorial Problems," *Computer*, Vol. 33, No. 4.

[4]Zhong, P., Martonosi, M., and Ashar, P., "FPGA-based SAT Solver Architecture with Near-zero Synthesis and Layout Overhead," *IEE Proceedings on Computers and Digital Techniques*, Vol. 147, No. 3, 2000, pp. 135–141.

[5]Kumar, S. R., Russell, A., and Sundaram, R., "Faster Algorithms for Optical Switch Configuration," IEEE International Conf. on Communications, Towards the Knowledge Millennium, 1997.

[6]Bellows, P., and Hutchings, B., "JHDL-an HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Inst. of Electrical and Electronics Engineers, New York, 1998, pp. 175–184.

[7]Barry, R., and Humblet, P. A., "Latin Routers, Design and Implementation," *Journal of Lightwave Technology*, Vol. 11, No. 5, 1993, pp. 891–899.

[8]Ji-Her Ju, and Li, V. O. K., "TDMA Scheduling Design of Multihop Packet Radio Networks Based on Latin Squares," *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 8, 1999, pp. 1345–1352.2000.

[9]Woo, T., "MPCBN: A High Performance ATM Switch," *IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications*, Inst. of Electrical and Electronics Engineers, New York, 1995, pp. 585–591.

T. Vladimirova
Guest Associate Editor