Appearing in *IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM 2002), April 22–24, 2002

Hardware-Assisted Fast Routing

André DeHon
andre@cs.caltech.edu
Dept. of CS, 256-80
California Institute of Technology
Pasadena, CA 91125

Randy Huang
rhuang@cs.berkeley.edu
UC Berkeley
Soda Hall #1776
Berkeley, CA 94720

John Wawrzynek
johnw@cs.berkeley.edu
UC Berkeley
Soda Hall #1776
Berkeley, CA 94720

Abstract

To fully realize the benefits of partial and rapid reconfiguration of field-programmable devices, we often need to dynamically schedule computing tasks and generate instance-specific configurations—new graphs which must be routed during program execution. Consequently, route time can be a significant overhead cost reducing the achievable net benefits of dynamic configuration generation. By adding hardware to accelerate routing, we show that it is possible to compute routes in one thousandth the time of a traditional, software router and achieve routes that are within 5% of the state-of-the-art offline routing algorithms for a sample set of application netlists and within 25% for a set of difficult synthetic benchmarks. We further outline how strategic use of parallelism can allow the total route time to scale substantially less than linearly in graph size. We detail the source of the benefits in our approach and survey a range of options for hardware assistance that vary from a speedup of over 10× with modest hardware overhead to speedups in excess of 1000×.

1 Introduction

Several researchers have attempted to reduce the time taken to route a design by tuning software search algorithms or attacking the problem with multiple processors. They were, at best, able to achieve route times on the order of seconds (See Section 3). While these improvements are impressive and interesting for fast turn-around in the edit-compile-debug cycle and rapid prototyping, they still represent billions of cycles and are not sufficient to make runtime routing viable in typical circumstances where it might benefit dynamic spatial computations.

With a modest amount of support hardware added to the routing network, the network itself can assist in the search for free routing paths. At the core of our solution, we use this augmented network to find all available paths between a source-sink pair in time proportional to the distance of the route using a parallel, hardware search. Using this hardware assisted technique, the time for the search task can be reduced a thousand fold over the software version. Additionally, by using pipelining even the distance term need not limit the rate of route completion. This suggests a large family of possible hardware-software routing solutions and raises a host of questions, which we begin to address in this paper.

- How fast can we make routing?
- What area overhead must we pay to achieve various levels of hardware acceleration?
- What quality, if any, must we sacrifice to enable efficient hardware routing?
- How do these solutions scale with larger designs and systems?

In this paper, we outline the basic hardware solution (Section 5 and 6) and quantify where time goes in the router, how our solution reduces routing time, and what quality must we tradeoff to enable this solution (Section 7). Before expanding on these details, we review the relevance of this problem to this community (Section 2), the prior work on software solutions (Section 3), and relevant network and router background (Section 4).

2 Motivation

To take fullest advantage of reconfigurable, spatial computing platforms, we want to specialize the instantaneous computation to both the problem being solved and the available resources in the platform. Both of these may be *late bound* quantities. That is, for portability, the exact set of resources available on the platform may not be known until the program begins to run, and the characteristics of a particular problem are not known until the program sees the problem. In cases such as these the time required to partition, place, and route the design onto the platform is part of the critical runtime of the application, not part of a pre-runtime compilation process. As such, any runtime taken to solve these tasks diminishes the potential acceleration offered by the spatial computing platform and narrows the domain of application where the reconfigurable platform or the specialized solution is superior to the more conventional alternatives.

Our goal in this respect is to understand how far we can

compact the time required for routing of general-purpose computational graphs (*i.e.* without exploiting task-specific route structure). As noted, the minimum routing time will set a limit on how far we can reduce the runtime overhead we need to exploit specialized instances, dynamic resource requirements, and dynamic resource availability. Here, we focus entirely on routing and leave partitioning and placement as separate issues. Fast software partitioning is treated in part in our recent paper on quasistatic scheduling [1]. Callahan demonstrates one approach to fast, 1-D, datapath placement [2]. Sankar and Rose demonstrate a more general approach to fast placement [3].

SCORE and Runtime Mapping We have been developing SCORE, a stream-based compute model which virtualizes reconfigurable computing resources (compute, storage, and communication) by dividing a computation up into fixed-size "pages" and time-multiplexing the virtual pages on available physical hardware [4]. SCORE's goal is to serve as an abstract interface level, like an ABI or API in conventional programmable processors, that abstracts the detailed hardware implementation, including the number and kinds of resources, from the application and programmer. This allows SCORE designs to migrate automatically to newer, larger hardware platforms and exploit the additional resources. A key consequence of this abstraction is that the compiler does not know how big the SCORE platform will be. Consequently, the SCORE runtime must manage mapping the abstract SCORE graph onto the available physical hardware, time-multiplexing the large SCORE graph on smaller hardware as necessary.

Consequently, the SCORE runtime must perform routing no sooner than application load time, and potentially as frequently as every reconfiguration in the time-multiplexed execution. Therefore, route time will reduce the raw performance potential of a SCORE application. If routing takes more time than a typical time slice, a route will have to be amortized across multiple time-slices to be viable.

The use of fixed-size compute pages connected by multi-bit buses reduces the size of the runtime placement and routing problem by a constant factor. This has the effect of simplifying the routing task, but does not make it trivial or address long-term scaling. As we will see in Section 8, the multi-bit buses and larger grained pages allow us to amortize the hardware overheads associated with hardware-assisted routing, making their cost quite small for the typical SCORE case. Our current impression is that a compute page should be a few 100's (perhaps small 1000's) of bit-level operators (*e.g.* 4-LUTs or adder-bits) and buses will be 4–16 bits wide.

Nature of the Problem The work required to find routes is, *at least*, linear in the number of two-point net connections, and is likely to scale faster than linearly if we wish to achieve equivalent levels of route quality. In separate experiments, we found that if we limited the number of route searches (but not the work done per route search) to a constant multiple of the number of nets, the resulting, pathfinder route quality decreased with increasing network size, suggesting that the number of route trials required to find an equivalently good solution certainly increases faster than linearly in network size. Additionally, the work per route trial increases with larger networks, due both to the increased path lengths and the increased number of potential paths to search.

3 Prior Work

Chan and Schlag attacked the problem of FPGA routing times with both coarse-grained parallelism and FPGA-accelerator assistance. They were able to show a little over a $3\times$ speedup in route time using 4 uniprocessor workstations [5], and delay-driven routing acceleration of a little over 2–$4\times$ speedup using 5 processors [6].

Swartz, Betz, and Rose employ depth-first search and focussed target selection to tune a pathfinder-based router to decrease route time in low-stress routes (those where the router is allowed to use more channels than those required by the channel minimizing pathfinder). They show that this combination leads to a router which requires roughly 1.1 ms per LUT/FF pair using a 300 MHz Sparcstation [7]. Normalizing for hardware technology, this means roughly 300,000 cycles per LUT/FF pair, or, assuming an average of 4 input nets per LUT-FF, about 75,000 cycles to route each two-point net.

Tessier used domain negotiation and A* search to tune a pathfinder-based router for fast routing. He showed that domain negotiation and depth-first search allowed him to achieve similar reductions [8]. For the fastest, low-stress cases his router was able to route roughly one 4-LUT/FF every 1.4–4 ms running on a 140MHz Ultrasparc [9]. This achieves 200,000-500,000 cycles per 4-LUT placing it in the same ballpark as the Swartz depth first router.

Fast, greedy, maze routing in Lola achieves roughly 3–7 ms per net running on a 166MHz Pentium PC [10]. This corresponds to roughly 500,000 cycles per net.

Here we see the state-of-the-art in fast, software-based routers achieves roughly 75–100,000 cycles per net. Our own fast router achieves a similar result of roughly 95,000 cycles per net. Given the variations in machine architecture and increasing relative memory costs, cycles are a crude comparison metric for comparison. Nonetheless, this establishes a consistent base range for our detailed hardware-software comparison.

4 Background and Definitions

HSRA We build on the linear switch population HSRA [11] (See Figure 1). A key feature of this network is that the number of switches in each hierarchical switchbox is

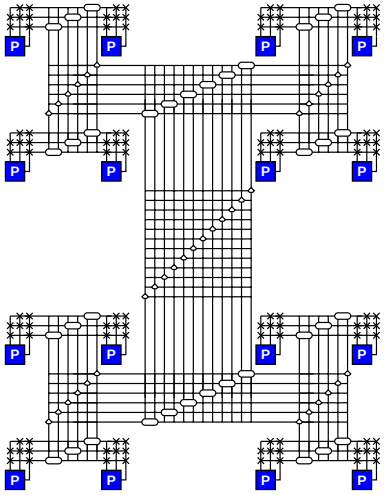

Figure 1: HSRA Network Topology
P's represent the network endpoints (*e.g.* LUTs or SCORE Compute Pages). The circles and ovals are switchpoints (See Figure 2 for details); X's show the connection-box switches. Network shown has 3 base channels.

linear in the number of wires in the switchbox and the total number of switches in the network is linear in the number of endpoints.

This network has an important property which is not shared by Manhattan arrays: There is a unique set of switchboxes between any source and sink. Consequently, global routing is trivial (there is only the one solution), making detail routing our only concern. For our hardware-assisted router, this also means there is a unique "least common ancestor" or "crossover" switchbox between any source and sink. We use this localization to detect route success, or failure, locally in the crossover switchbox. Further, once we select a particular wire (switch) in a crossover switchbox, the path from the crossover to the source and sink, including the set of switches and wires in the path, is completely unique; this property simplifies path identification and allocation.

Pathfinder Pathfinder is the dominant approach to FPGA-routing currently in use in the academic community and heavily used in industry as well. It forms the basis of the previous, software-based, attempts to accelerate routing (Section 3). Starting from the base Pathfinder algorithm [12], we implemented our own version for the HSRA [11]. We believe our implementation is very close in spirit to the original. The basic algorithm is as follows:

1. Create a fixed ordering of all nets in the design.
2. While there are unrouted nets and we have not exceeded the maximum number of route trials:
   a. for each net in the original fixed ordering
      - if net is unrouted (no path or shares paths)
      - Perform a *route trial* $\equiv$ rip up the congested net and reroute
   b. update history cost for each congested net

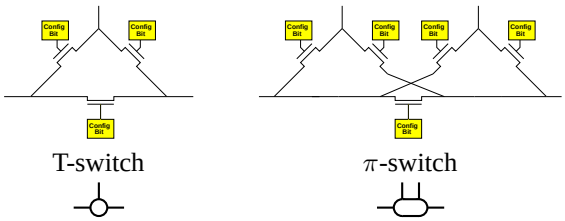

Figure 2: HSRA Switchpoints
(pass transistor switch implementation)

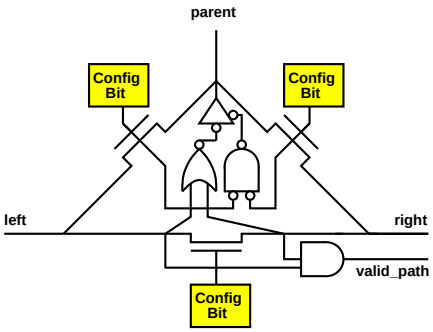

Figure 3: HSRA T-Switch with Path-Search OR
The π-switchpoint can be augmented in a similar manner. This logical augmentation can be easily adapted for rebuffered and clocked switchpoints as well.

5 Basic Solution

The key idea in our hardware-assisted router is to use the network structure itself to support the parallel route search and to keep track of the state of the network.

To find an available route in the HSRA network, we typically start at the source and the sink node and trace free (least cost) paths from the source and sink to the crossover switchbox. If the search from the source and the search from the sink meet on one (or more) wires at the crossover switchbox, we have found a viable route path. We can then allocate the path (one of the paths) to this source-sink pair.

A pair of typical HSRA switchpoints is shown in Figure 2. The switches allow us to make connections as appropriate, connecting the children for crossover connections, or connecting the appropriate child to the parent for up and down connections.

Now, consider adding a logical OR between the two children channels and placing the result on the associated parent channel (See Figure 3). With this addition, we can perform a *route trial* roughly as follows:

1. Set all endpoints (*e.g.* LUTs or SCORE compute pages) to drive zeros into all unused input and output connec-

tion to the network and all allocated source lines (leave allocated sink lines undriven as they will be driven by their associated sources).
2. For the designated source-sink pair which we are currently trying to route, drive a one into each unused (available) network connection.
3. Wait for the driven ones to propagate through the network to the unique crossover switchbox.
4. At the crossover switchbox, scan for a switchpoint which receives a one on both of its sibling sides; only this source-sink pair is driving ones, so a matched pair of ones indicates a complete path from both the source and the sink.
5. Allocate the unique path associated with one such matched pair; this means we go ahead and set the switches accordingly to connect this path. Note that this means this path will have zeros driven into it in the future and will not be considered in subsequent route searches.

Figure 4 shows an example of this route search. We now perform this search and allocation route trial successively for every network connection in the design.

The prospect for acceleration here is simple. In the traditional, software route search, each route trial takes several tens of thousands of cycles (*e.g.* see Table 5) to walk a network data structure and to explore all the possible paths between source and sink until a free (or inexpensive) path is found. In this hardware case, we use the network itself to explore all paths simultaneously. It does so quickly because all the switched paths are instantiated in hardware and directly connected by wires. It takes only the signal propagation delay across the wires and switches to trace back all possible paths. If the subsequent allocation can be performed cheaply in place, this turns the whole task from several tens of thousands of cycles into a just a few cycles.

6 Details

To obtain a complete scheme we will need to fill in a few of the details left open in the sketch above. In this section we will address these details and offer some sufficient solutions. Here, we need to answer:

- how to select among available paths?
- what to do when no routes are found?
- how to perform allocation and victimization?

A key issue with respect to the traditional, software pathfinder is history and costs. In the simple scheme above, we only have binary costs—either a path is free or it is not. Pathfinder allows nets to share paths and uses congestion and historical congestion to bias the cost of paths. We will either have to use a simpler scheme, perhaps at a cost in route quality, or we will have to complicate our hardware scheme further to approximate the history and congestion information used by pathfinder.

Which Path? In the simple form above, we found a set of paths and needed to select one of them. If they are all truly free, we can certainly allocate any one of them. Even in a software pathfinder, we often have this situation where there are a number of equal cost paths to select among. We can select deterministicly from them with some fixed priority scheme, or we can select randomly among them.

For the cases where we have less than complete history information, at least, we use random selection to increase path exploration. This way, multiple route attempts will tend to place a net on different, available paths, allowing us to stochastically explore the alternatives. A bad or limiting path selection on one route attempt will be unlikely to be repeated on a subsequent trial.

No Path? The bigger question is what do we do when there is no path available. Pathfinder allows the new path to share resources with the least congested existing path. There are some ways to begin to approximate this, but it certainly increases the state and complexity of mechanism required to support it.

A simpler case, more amenable to tight-hardware implementation, is to rip-up conflicting routes in order to expose an available path. This raises the question of which routes to victimize. Here, with a slight amount of additional complexity, we could identify the path that would disturb the least existing switched connections. Such a selection would be roughly equivalent to selecting the path with least congestion, ignoring any history information.

An even simpler case is to, again, select randomly among all possible paths. If the conflicting paths have alternatives, then they can be rerouted. Here, the fact that path selection is stochastic is especially important. As routes accumulate, we will bias the probability distribution function of routes with choice away from the paths which are needed most heavily for difficult connections.

Random Path Selection We can perform random path selection economically in the switchbox by using a pseudo-random number generator (PRNG) and a cyclic segmented parallel prefix (CSPP) circuit [13]. The PRNG indicates which crossover switch is preferred for allocation. We mask out those which are not selected, and use the CSPP circuit to identify the first circuit candidate switchpoint identified by the path search. The CSPP circuit allows us to identify the path in $O(\log(W))$ time, where W is the number of switchpoints in the switchbox. W grows as $O(N^p)$ ($1.0 > p > 0.5$; p is the exponent in Rent's Rule which can be used to characterize the growth rate of bisection bandwidth in the HSRA [11]), so the depth of the CSPP circuit grows only as $O(\log(N))$. Our experimental results using this PRNG-CSPP random number generation scheme are statistically indistinguishable from results generated using a pure random number generation.

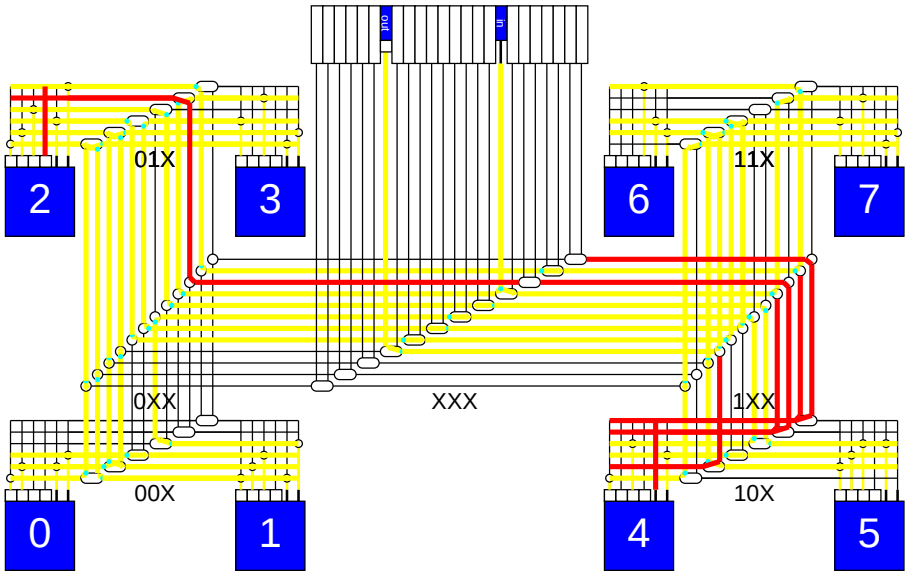

Figure 4: Route Search

Shown here is the result of a path search for a route from node 4 to node 2. The light (yellow), thick lines show pre-existing routes. The dark (red), thick lines show the paths driven to ones by the source and sink and propagated via the up OR logic. At the crossover switchbox (labelled XXX), there is only a single switch which has a one arriving from both sides. We allocate the path that is joined by this switch. Note that there is a single, unique path from the source (node 4) to the sink (node 2) through this switch.

Hardware Allocation We can build a route allocation mechanism into the network with only a single extra AND gate for each switch, an "allocate" pull up at the crossover, and a binary tree to identify the global route path. The global route path binary tree looks just like the OR-up logic in the T-switches, except that it has no configuration bits; since this is a binary tree, the global-route tree has only a single T-switch in each switchbox and is, therefore, only a small additional cost on top of the large number of switchpoints and wires already in each switchbox. At the leaves, this tree, which is separate from the normal routing paths, is also driven by the source and sink participating in the route search. Once the path search has found a possible path, we stop driving the normal network paths and drive an "allocate" request, a one, back down the selected path to perform the allocation. Each switch which receives this one performs the actual allocation on the appropriate parent-child link, propagating the allocation, in turn, down to that child; the global-route tree shows the switchpoint which child connection to allocate (See Figure 5).

Note that if we simply tried to allocate from the top without the global-route tree, the switchpoint would not know which child connection to make; the global-route tree provides this information. A similar problem occurs if we try to allocate from the bottom; without additional information, it is not clear which of the two up connections in a π-switchpoint the route should allocate. This binary tree is only used during routing, so could be shared with other control functions that are only needed during operation.

Hardware Victimization When there are no free routes found, we need to deallocate (victimize) existing routes in order to make a new route. Complete logic to support this in hardware is shown in Figure 6. The logic needs to identify the intersecting paths and propagate the fact that the path is a victim to all switchpoints along the path before actually clearing the switchpoints. All together, this takes 3 crossover-to-leaf trips in network to clear routes plus a 4th trip to perform the new allocation.

We also need to know which routes were victimized. At the end of the victim propagation, the sink will know, by the position of the input, which source it lost. If the sink knows which source is associated with this input, that is enough information for it to inform the route controller which source-sink pair(s) has been ripped up and needs to be re-routed. It is possible that many paths are victimized during a single deallocation. A binary collection tree would allow us to identify all victim paths in at most a number of cycles: $\log(N)$+number of victims.

Software Allocation and Victimization A more modest solution, both in hardware cost and performance potential, is to perform only the path search in hardware (Figure 3) and perform all record keeping in software. With no allocation or deallocation logic, we will need configuration bit

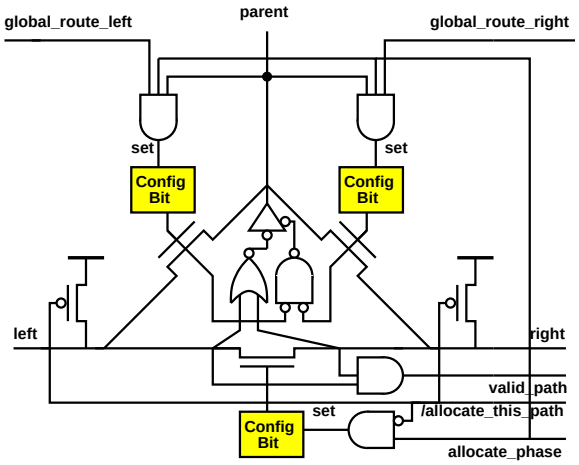

Figure 5: HSRA T-Switch with Allocation Logic

Allocate logic added to pass-transistor T-Switchpoint: At the termination of path search, we do 3 things in sequence: (1) deassert the source and sink drive, but leave the global route tree driven, (2) assert allocate_phase to enable allocation, and (3) assert allocate_this_path to drive a one into the selected path at the crossover switchbox. Primed in this manner, the single path selected at the crossover is allocated along successively stages all the way down to the source and sink.

addressability to set and clear configuration bits as routes are allocated and victimized.

The software bookkeeping needs a table to track switch usage. This table is indexed by switchpoint identification and contains:

1. switchpoint identification for the left child switch of this switchpoint
2. switchpoint identification for the right child switch of this switchpoint
3. the net that is using this switchpoint (if used)
4. switchpoint identification for the crossover switch for the route through this switchpoint when in use

This tables is $O(N)$ in size and is potentially accessed quite irregularly. It is unlikely this table will be cacheable, making each table reference moderately expensive.

Route search is initiated as before using the OR-up logic. When we find an available route, the controlling processor queries the crossover switchbox to discover which path was selected. The processor then walks the switchpoint table, starting at the crossover switchpoint, and stores the net identification and crossover switchpoint in each switchpoint along the path. As each switchpoint is visited, the processor also issue a command to the network to allocate the appropriate switch bit.

When it is necessary to victimize paths, we perform the route victimization in software. Once we pick a target path, we:

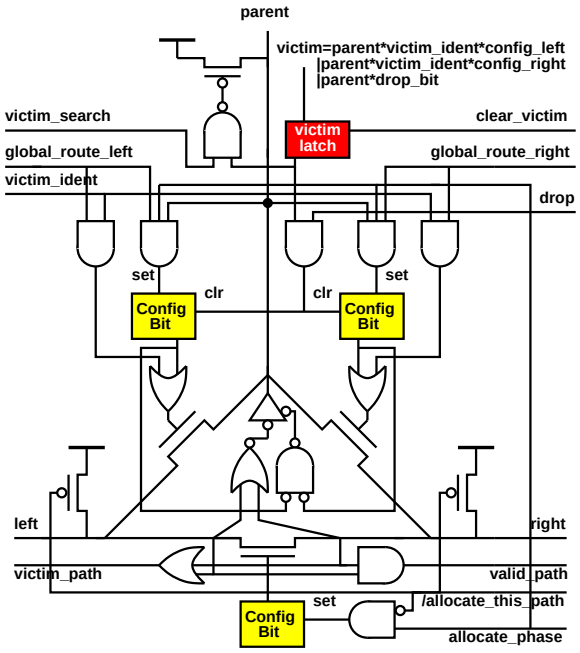

Figure 6: HSRA T-Switch with Victimiztion Logic

Victimization logic added to pass-transistor T-Switchpoint: When we must steal a path in use, we do the following: (1) assert victim_ident and drive a one into the selected path at the crossover switchbox; this sets the victim latch everywhere the new path intersects an old path. (2) propagate the vicitm information up to the top of the victimized routes by asserting victim_search; (3) drive a one onto the victim paths from the crossover and assert drop_bit to mark all affected paths; (4) assert drop to clear the old paths; (5) poll the leaves to discover victims; (6) perform a normal allocation of the now cleared path.

1. walk the switchpoint table starting at the victim crossover switchpoint
2. for each switch visited we need to:
   - find out if a net already occupies that segment
   - if a net occupies the segment,
     1. add that net to the list of unrouted nets
     2. lookup the switchpoint which is the root of the existing net connection
     3. for each switchpoint belonging to this net
        - issue a command to the network to clear the switch bit
        - clear the net from the wire segment's entry in the segment table

After deallocating the victim paths, we can proceed with allocation.

In the allocate phase the processor reads $2\log_2(N)$ switchpoint entries from the switchpoint table. While each entry in the table will likely fit nicely in a cache line, ac-

cess to the switchpoint table will almost certainly generate a cache miss and have to be satisfied from main memory. Therefore, we expect to pay a single main-memory reference time (T_m) for each of the $2\log_2(N)$ lookups, followed by 1–3 cache reference times (T_c) when we read more than one word from a switchpoint entry. There are $2\log_2(N)$ writes to set bits in the array (T_a) and $2\log_2(N)$ writes to the wire segment table. These writes are to items which are likely to be in the same cache line and are potentially pipelineable using a write buffer (T_{wb}). Therefore, allocate is likely to take:

$$T_{allocate} = T_m + 2\log_2(N)\,(T_m + 2T_{wb} + T_a) \qquad (1)$$

An allocated route could, worst-case, theoretically conflict with $2\log_2(N)$ different routes. Unrouting each of these requires $2\log_2(N)$ memory reads and configuration writes. In the worst-case, then, victimization requires $O(\log^2(N))$ reads to the wire segment table, most of which will probably incur main-memory reference latency. More typically, we expect a small number of associated victims. Assuming an average number of victims V, and noting that deallocating a victim requires essentially the same operations as allocating a path, the deallocation takes roughly:

$$T_{victim} = 2\log_2(N)T_m + V \cdot T_{allocate} \qquad (2)$$

Let us assume $T_c = T_{wb} = T_a = 1$ cycle. If we assume $T_m = 50$ cycles, a 16K-node network takes: $T_{allocate} = 50 + 2 \cdot 14\,(53) \approx 1500$ cycles. From our experiments, typical values of V are between 3 and 4 for large networks, so we will assume $V = 4$ here, giving: $T_{victim} = 2 \cdot 14 \cdot 50 + 4 \cdot 1500 \approx 7500$ cycles. Maintaining these data structures in memory clearly becomes the dominant time cost if we go with this hybrid hardware-software scheme. For a modern, large-scale, FPGA, we would likely use on-chip memory such as the embedded DRAM block designed for SCORE/HSRA [14]. Random access in this memory takes 14 logic cycles ($T_m = 14$) making $T_{allocate} \approx 500$ cycles and $T_{victim} \approx 2500$ cycles.

Parallelism Only the searches to the same top, crossover switchboxes need to be sequentialized. A path search in the left half of a network can proceed completely in parallel with a path in the right half of a network. In general, if the least common ancestors of the nets' source-sink pairs are in different subtrees, the search can proceed in parallel. This means, we can search sequentially for paths which crossover in the topmost switchbox then search in parallel for the paths which crossover in its immediate left and right switchbox. This parallel decomposition continues in turn. As a result, the ultimate sequentiallization in this scheme is the sum of the maximum number of paths crossing over at each switchbox level rather than the total number of nets. For a typical network or design with $1.0 > p > 0.5$, this means the sequentiallization goes as N^p rather than N. Ultimately, of course, we will saturate the processor's time to give attention to starting and completing routes. For large designs, we might consider allocating a control processor to subtrees at some level(s). For large designs, there are many other good reasons to consider this in the SCORE case, at least. Consequently, it is possible for the entire route time to scale only as $O(N^p)$.

Applications	Pages	Array size	Page size	IOs
MPEG Enc.	92	128	512	18
JPEG Enc.	13	16	512	16
JPEG Dec.	12	16	512	16
Wavelet Enc.	30	32	64	6

Table 1: SCORE Benchmark Implementation Details

7 Results and Quantification

7.1 Benchmarks

To quantify the routing quality we tradeoff for simplicity and speed, we compare the random path selection algorithm with the traditional pathfinder algorithm using two distinct benchmark sets: a collection of multimedia SCORE benchmarks and a set of difficult synthetic benchmark.

SCORE Benchmark This benchmark set includes four applications: an MPEG encoder, a JPEG encoder, a JPEG decoder, and a wavelet-based image encoder. Design information is summarized in Table 1. Placement is performed by the SCORE scheduler [1], and the same placement is used for all routing experiments.

Synthetic Benchmark Although it is important to have netlists from real applications, the SCORE netlists are moderately small and are likely to be moderately easy to route. To make sure our solution performs reasonably with larger and harder designs, we augment the SCORE benchmarks with a set of difficult synthetic benchmarks; these benchmarks make sure to maximally fill many switchboxes according to the switchbox population, assuring that we can differentiate the effects of route quality and limited population switchboxes. We scale the synthetic network size from 8 to 16,384 and, for each size, we generate 100 netlists which stay unchanged throughout our experiments.

7.2 Testing Environment

Our software router is written in C and compiled with the GNU C Compiler (version 2.95.2) using `-O3` option. Benchmarks are run on 500MHz Pentium3-based system running Linux 2.2.12 with 100MHz system bus and variable amount of main memory (from 256MB to 1.5GB). We use the 64b TSC (Time Stamp Counter) timer on the processor to measure running time of the program in cycles. In general, since we are performing graph traversals on a

large data structure, most memory accesses will be cache misses. However, we make sure our entire data structure will fit in the main memory, so that we are not measuring performance derated by virtual memory thrashing.

7.3 Algorithm Comparison

For the SCORE benchmark, we route each netlist with the base channel capacity set to the number of IOs per page and increment the base channel capacity until a valid route is found. Table 4 summarizes the results and they are fairly encouraging. Compared to the pathfinder results, the random algorithm is at most 5% worse in quality. In fact, for the wavelet application, we achieve identical route quality in a similar amount of time. The random software algorithm may take more route trials than the Pathfinder algorithm to achieve comparable quality since it does not have the benefit of costs and history to speed convergence.

For the synthetic benchmark, we perform similar experiments. We route the 100 netlists for each array size; for each netlist, we increment the base channel capacity until a valid route is found. Average results are summarized in Table 5.

- The random algorithm is at most 25% worse in route quality than the pathfinder algorithm for the largest networks shown here.
- The two algorithms implemented in software use approximately the same amount of time. This is important because it shows any improvement in speedup comes strictly from hardware assistance and not from the simplifications to data structure and decision making inherent in the random path selection.
- For our implementation of the pathfinder algorithm, the number of cycles/net is comparable with previous work.

7.4 Is This the Best We Can Do?

In this section, we evaluate two potential strategies to improve the quality of the random algorithm.

Route Trial Multiplier In the previous experiment, we stop the router after the number of route trial has reached $RT_{max} = |\text{nets}| \times RT_{mpy}$ and re-start the router with an increment in the base channel capacity. One way to improve route quality is to raise the amount of route time allowed by increasing RT_{mpy}.

We perform the route trial multiplier experiments on two array sizes, 1024 and 2048, and summarize the result in Table 2. It is clear from this data that very few netlists converge after route trial multiplier of 2, perhaps because the random path selection has guided the router into a local minimum. This data suggests that increasing RT_{mpy} above 2 is largely futile.

Stop and Retry Because we are using a random algorithm, we get a different result every time. Another strategy to improve quality is to try multiple route starts so that we explore different parts of the route space. Based on the results from the previous experiment, we route each start for $RT_{mpy} = 2$, then clear the network and make a fresh routing start if we have not achieved satisfactory results.

trial mult.	1024			2048		
	channels	min	max	chan.	min	max
1	1190	11	14	1231	11	14
2–8	900	9	9	900	9	9
9	899	8	9	900	9	9
10	898	8	9	900	9	9

Table 2: Route Trial Multiplier Experiment

size	channels		
	8	9	10
512	72	28	0
1024, 2048, 4096	0	100	0
8192	0	65	35

Table 3: Sample Probability Density Function

We perform the stop-and-retry experiments by routing each netlist 100 times to obtain a probability distribution function (PDF) for each netlist from 512 to 8192 nodes. In Table 3, we show the PDF of a representative netlist for each array size. From the data presented, it is clear that we can use this strategy to get within one channel of the pathfinder solution. The data suggests, on average, it takes 1.6 starts to achieve 9 channels for the 8192 node network. With the hardware speedups we show in Tables 6 and 7, we see we can afford a few restarts and still achieve three orders of magnitude route time reduction.

7.5 Hardware Acceleration

The time to route a netlist with hardware assist is:

$$\begin{aligned} T_{netlist} &= N_{RT} \cdot (T_{ctrl} + T_{path} + T_{check} + T_{alloc}) \\ &\quad + N_{RO} \cdot T_{victim} \end{aligned}$$

Of a given netlist, N_{RT} is the total number of route trials and N_{RO} is the total number of ripouts. We measure these two numbers from our software implementation of the random algorithm and for each array size, we list the average (across the 100 nets) in Table 8. T_{ctrl} is the number of cycles to send a control signal. Control signals travel from the root to the leaf nodes in $O(\log(N))$ cycles. T_{path} is the number of cycles it takes to propagate a signal from the leaf nodes to the crossover switch box. T_{path} is net dependent and bounded by $O(\log(N))$. T_{check} is the time it takes to generate a random number and check for available routes at the crossover switch box and is $O(\log(N))$. T_{alloc} is the number of cycles it takes to allocate a route. During the allocation phase, we send the three control signals in sequence. These control signals can be pipelined, so T_{alloc}

Applications	Pathfinder				Software Random		
	channels	cycles/netlist	cycles/net	cycles/RT	channels	cycles/netlist	cycle/net
MPEG Enc.	24	148266090	176087	110567	25	280362511	237283
JPEG Enc.	18	3646453	58813	21917	18	9196693	148333
JPEG Dec.	18	2910831	47718	17186	18	3251943	53310
Wavelet Enc.	8	2881255	46471	16118	8	2938845	47400

Table 4: SCORE Benchmark Results

netlist size (nodes)	Pathfinder						Software Random				
	total chan.	channels min	channels max	average cycles/netlist	average cyc/net	average cyc/RT	total chan.	channels min	channels max	average cycles/netlist	average cyc/net
8	513	5	6	277401	18805	7209	502	5	6	129978	8779
16	585	5	7	688513	22958	10705	573	5	6	490791	16264
32	649	6	7	1767525	28755	12463	646	6	7	1446211	23552
64	691	6	7	2639439	21583	14252	700	7	7	4649098	37865
128	701	7	8	8707525	35618	19921	752	7	8	13831655	56837
256	712	7	8	35310749	72362	31206	800	8	8	31292686	64154
512	796	7	8	23592698	24127	17660	801	8	9	99440287	101422
1024	800	8	8	54085773	27681	20499	899	8	9	184577989	94488
2048	800	8	8	142989526	36544	24607	900	9	9	390927603	99908
4096	800	8	8	397137828	50741	32465	900	9	9	1349098041	172380
8192	800	8	8	1087326104	69457	41662	904	9	10	5035702952	121673
16384	800	8	8	2977991302	95185	56293	1000	10	10	5519778233	176429

Table 5: Software Comparison of the Pathfinder Algorithm vs. Random Route Selection

size	Pathfinder	Software Allocation		Base		Overlap		Parallel	
	average cycles/netlist	average cyc/netlist	average speedup	average cyc/netlist	average speedup	average cyc/netlist	average speedup	average cyc/netlist	average speedup
64	2639439	666338	4	7651	345	6115	432	3058	863
128	8707525	2011772	4	24737	352	19802	440	6601	1319
256	35310749	1454324	24	23415	1508	17871	1976	4468	7903
512	23592698	6988570	3	105959	223	82901	285	11843	1992
1024	54085773	3570559	15	79605	679	58235	929	7279	7430
2048	142989526	7888456	18	184251	776	134773	1061	9627	14854
4096	397137828	21491478	18	491797	808	363565	1092	21386	18570
8192	1087326104	96927040	11	2013247	540	1527775	712	52682	20639
16384	2977991302	50439784	59	1569221	1898	1116657	2667	31018	96008

Table 6: Hardware Acceleration for Synthetic Netlist

Application	Pathfinder	Software Allocation		Base		Overlap		Parallel	
	cyc/netlist	cyc/netlist	spdup	cyc/netlist	spdup	cyc/netlist	spdup	cyc/netlist	spdup
MPEG Enc.	148266090	2507114	59	35618	4163	28026	5290	16986	8729
JPEG Enc.	3646453	373182	10	3755	971	3080	1184	2934	1243
JPEG Dec.	2910831	493084	6	4909	593	4104	709	2952	986
Wavelet Enc.	2881255	157006	18	2129	1353	1655	1741	1199	2403

Table 7: Hardware Acceleration for SCORE Netlist

Nodes	Nets	N_{RT}	N_{RO}
64	122	256	57
128	244	705	187
256	487	693	75
512	979	2562	550
1024	1953	2137	62
2048	3912	4498	187
4096	7826	10686	835
8192	15654	37344	5982
16384	31301	32326	295

Table 8: Random Algorithm Statistics

equals one trip through the network plus a small constant number of cycles to drain the pipeline. T_{victim} is the number of cycles it takes to victimize a channel at the crossover switch. During the steps of the victimization process, we have to wait for one step to finish before we can start another. Therefore, T_{victim} includes four trips through the network plus a constant.

Using array size of 4096 as an example, $N_{RT} = 10686$ and $N_{RO} = 835$. For convenience, we assume that we pipeline every single level of the network hierarchy and $T_{ctrl} = T_{path} = \log_2(4096) = 12$. For p = 0.5 and a base channel capacity of 9, the top-level crossover switch box has 576 possible routes. We further assume it takes a single cycle to generate a random number and three cycles to calculate 576b CSPP ($T_{check} = 1 + 3 = 4$ cycles). $T_{alloc} = \log_2(4096) + 2 = 14$ cycles. $T_{victim} = 4 \cdot \log_2(4096) + 4 = 52$ cycles. Putting it all together, $T_{netlist} = 10686 \cdot (12+12+4+14) + 835 \cdot 52 = 492232$ cycles. Compared to the pathfinder result of 397,137,828 cycles to route netlists of size 4096, this is a speed up of over 800. Table 6 summarizes the average results for various array sizes under the "Base" column.

Optimization: Overlap Netlist Routing A small improvement to the base hardware case can be made by overlapping independent operations. In the final phase of routing, we perform allocation; at the same time, we can start issuing commands to route the next net. By the time the routing commands arrive at the leaf nodes, the previous net's allocation step has completed. This optimization eliminates the T_{ctrl} term from the equation. Using the previous example, $T_{netlist} = 10686 \cdot (12+4+14) + 835 \cdot 52 = 364000$ cycles and we achieve a speed of over 1000.

Optimization: Parallelize Netlist Routing As described in Section 6, the achieveable speedup depends on how much we have to sequentialize netlist routing; the ultimate sequentialization is the sum of the maximum number of paths crossing over at each switch box level. We summarize sequentialization and potential speedup in Table 9.

Software Allocation and Victimization Case As described in Section 6, if we assume $T_m = 50$ and use Equations 1 and 2 (using measured V's from the actual netlists), we see that we can achieve a speedup of 18 for our 4096 node netlists.

Nodes	average Nets	ultimate sequentialization	potential speedup
64	122	61	2
128	244	82	3
256	487	122	4
512	979	139	7
1024	1953	244	8
2048	3912	279	14
4096	7826	460	17
8192	15654	540	29
16384	31301	869	36

Table 9: Speedup from Routing Nets in Parallel

8 Hardware Costs

The software allocation and victimization scheme requires only 3–4 gates per switchpoint (Figure 3). This should be compared to the three configuration bits and pass transistors required for a minimum switchpoint implementation. The additional 3–4 gates are likely to be less than half the size of the base switchpoint, suggesting, at most, a 50% switch area penally for a bit-level network. These connections are completely local; when switchpoints are wire dominated the area for these additional gates may be free. As we go to registered and buffered switchpoints (*e.g.* [11]), the area for the base switch increases, making these additional gates an even smaller marginal cost addition.

The full hardware scheme (Figure 6) requires roughly 20 gates. Consequently, this additional logic is likely to be 2.5× the size of the minimal, pass-gate switchpoint. This size is probably untenable for bit-level networks. For multi-bit networks, this area can be amortized across the entire datapath. The inter-compute-page network in SCORE architectures [4], for example, uses 4–16b datapaths. Amortized across a 16b datapath, even this full hardware scheme adds about one gate per switchpoint, resulting in only 10–20% area overhead in the worst case.

9 Summary

By adding a few gates to switchpoints, we are able to use the network itself to perform the free path search in parallel, completing a search in just tens of cycles. This replaces a software operation which take tens of thousands of cycles even in the fastest software routers. Accelerating only this portion of the task is sufficient to get greater than an order of magnitude reduction in routing time. The remaining software bottleneck becomes tracking the network state. To address this problem, we can push the allocation and victimization support into hardware. This gives

us a solution which is greater than three orders of magnitude faster than the original, fast software router. The full hardware scheme we present requires roughly 20 gates per switch and is likely to only be viable when this cost can be amortized across a wider datapath, such as used in typical SCORE networks. Achieving these speeds, we appear to give up at most 5% in route quality on typical designs and at most 25% on designs which are intentionally difficult to route.

Future Work In this paper we only describe point-to-point network connections. We have identified some possible extensions to support fanout and will explore those further in future work.

Several convenient properties of the HSRA made it particularly easy to formulate this hardware-assisted search for the HSRA, however, in principle this scheme can be extended to any kind of network. Notably, hardware-assisted routing for mesh-based interconnection topologies appears feasible but will require a few additional mechanisms.

Acknowledgments

Primary support for this project has come from the Defense Advanced Research Projects Agency (DARPA) contract DABT63-C-0048, with additional support from the California MICRO Program, ST Microelectronics, and Xilnix.

References

[1] Yury Markovskiy, Eylon Caspi, Randy Huang, Joseph Yeh, Michael Chu, John Wawrzynek, and André DeHon, "Analysis of QuasiStatic Scheduling Techniques in a Virtualized Reconfigurable Machine," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, February 2002, pp. 196–205.

[2] Timothy Callahan, Philip Chong, André DeHon, and John Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, February 1998, pp. 123–132.

[3] Yaska Sankar and Jonathan Rose, "Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs," in *Proceedings of the 1999 International Symposium on Field-Programmable Gate Arrays (FPGA'99)*. ACM/SIGDA, February 1999, pp. 157–166.

[4] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial," <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html>, short version appears in FPL'2000 (LNCS 1896), 2000.

[5] Pak K. Chan and Martine D. F. Schlag, "Acceleration of an FPGA Router," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, April 1997, pp. 175–181.

[6] Pak K. Chan and Martine D. F. Schlag, "New Parallelization and Convergence Results for NC: A Negotiation-Based FPGA Router," in *Proceedings of the 2000 International Symposium on Field-Programmable Gate Arrays (FPGA'00)*. ACM/SIGDA, February 2000, pp. 165–174.

[7] Jordan S. Swarz, Vaughn Betz, and Jonathan Rose, "A Fast Routability-Driven Router for FPGAs," in *Proceedings of the 1998 International Symposium on Field-Programmable Gate Arrays (FPGA'98)*. ACM/SIGDA, February 1998, pp. 140–149.

[8] Russell Tessier, "Negotiated A* Routing for FPGAs," in *Proceedings of the 5th Canadian Workshop on Field Programmable Devices*, June 1998.

[9] Russell Tessier, *Fast Place and Route Approaches for FPGAs*, Ph.D. thesis, MIT Dept. of EECS, February 1999.

[10] Stephan W. Gehring and Stefan H.-M. Ludwig, "Fast Integrated Tools for Circuit Design with FPGAs," in *Proceedings of the 1998 International Symposium on Field-Programmable Gate Arrays (FPGA'98)*. ACM/SIGDA, February 1998, pp. 133–139.

[11] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon, "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, February 1999, pp. 125–134.

[12] Larry McMurchie and Carl Ebling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, February 1995, pp. 111–117.

[13] Bradley C. Kuszmaul and Dana S. Henry, "Cyclic Segmented Parallel Prefix," UltraScalar Memo 1, Yale, November 1998, <http://ee.yale.edu/papers/usmemo1.ps.gz>.

[14] Stylianos Perissakis, Yangsung Joo, Jinhong Ahn, André DeHon, and John Wawrzynek, "Embedded DRAM for a Reconfigurable Array," in *Proceedings of the 1999 Symposium on VLSI Circuits*, June 1999.