Development of a Place and Route Tool for the RaPiD Architecture

Master's Project
Autumn Quarter, 2001

Akshay Sharma
University of Washington

As submitted to
Department of Electrical Engineering
University of Washington
Seattle, WA-98195

Advisor
Scott A. Hauck
Department of Electrical Engineering
University of Washington

1. Introduction

In recent years, designers have begun to migrate to the System-on-a-Chip (SOC) paradigm from traditional board-level design methods. New fabrication technologies have enabled the production of integrated circuits that have hundreds of millions of transistors. This capability has motivated the VLSI design community to investigate the possibility of integrating the sub-systems that comprise a traditional board-level design on a single piece of silicon.

There are several important reasons for moving to SOC. Significant factors include increased inter-device communication bandwidth, reduced power consumption, and overall area improvement. However, there are challenges that need to be addressed. SOCs typically have a very large design space, thus complicating existing tool-flows. Interfacing the individual sub-systems on the chip is another major issue. There is also a considerable increase in prototyping costs due to the size of the final design. Finally, there is a lack of post-fabrication flexibility. This is because designers can easily remove and replace components in board designs, a feature that is noticeably absent in present day SOC.

Designing an SOC targeted to a single application could prove extremely expensive if the hardware requirements of the application vary with time. It is therefore clear that a reconfigurable sub-system would be useful in SOC. This reconfigurable sub-system would provide the desired hardware flexibility to SOC designs.

The immediate solution to lack of post-fabrication flexibility of SOC designs is to include a Field Programmable Gate Array (FPGA) core designed by a major commercial vendor. However, current FPGA architectures fall considerably short of the levels of performance that can be expected from Application Specific Integrated Circuit (ASIC) implementations. This is because FPGAs aim to provide a great deal of flexibility in hardware, resulting in a performance trade-off.

Domain-specific reconfigurable architectures attempt to bridge the wide gap in performance that exists between ASICs and FPGAs. Such architectures are targeted to specific application domains (Digital Signal Processing (DSP), for example), and contain customized computational elements that are optimized to perform functions that fall within that domain. RaPiD[4,6] and PipeRench[7] are examples of domain-specific reconfigurable architectures. The computational elements in these architectures can be configured to execute a variety of DSP applications. Thus, if an SOC were to be designed for a pre-determined domain of applications, the corresponding domain-specific architecture would form the reconfigurable core. In this way, near-ASIC performance levels could be achieved, while providing hardware flexibility attributed to commercial FPGAs.

While domain-specific architectures are an attractive compromise between the flexibility of FPGAs and the performance of ASICs, it should also be noted that the creation of custom reconfigurable architectures for each separate domain is a labor-intensive process. Redesigning the reconfigurable core for a new domain would result in increased time-to-market, and significant design costs. In view of these considerations, we are actively investigating the automatic generation of custom reconfigurable architectures. At the highest level, the domain of applications and a set of constraints specified by the user will be used to generate a suitable architecture. This architecture could lie anywhere in the space between ASICs and FPGAs, depending upon the competing demands for performance and programmability required by the domain.

We will now describe the organization of the rest of this paper. Section 2 provides a brief background on the reconfigurable computing paradigm. Section 3 describes the approach that was adopted to place and route to an existing domain-specific architecture. In Section 4, the experimental setup and procedure have been described in detail. The results from the experiments are presented in Section 5, and our conclusions in Section 6. Section 7 discusses potential future work in mapping designs to custom reconfigurable architectures, and Section 8 lists acknowledgments.

2. Background

General-purpose microprocessors are the most common form of conventional computing today. This is because they are extremely flexible, and can be used to execute any application that can be converted to a sequence of software instructions. It is important to observe here that the flexibility of microprocessors lies in software. The microprocessor hardware simply executes instructions that are determined by overlying software like a compiler. Hence microprocessors can suffer a performance penalty, since they do not provide hardware resources that are optimized for a specific application.

Applications that require the highest performance are implemented as ASICs. An ASIC is specifically designed to perform a given computation, and is thus optimized for speed, power and/or area. However, in their quest for better performance, ASICs typically completely sacrifice flexibility, and cannot be used to execute any computation other than the one they were designed for. Thus, if the ASIC needs to be modified after fabrication, re-design and re-fabrication are necessary. This could prove extremely expensive, especially in an environment where the application changes with time.

Reconfigurable computing attempts to provide hardware resources that achieve better performance than general-purpose microprocessors, while providing greater flexibility than ASIC implementations. Reconfigurable devices contain arrays of logic and arithmetic units that can be programmed electrically to perform different functions. Programmable routing resources are also provided, and these can be combined with the logic resources to execute a variety of applications. A common example of reconfigurable computing is an FPGA. FPGAs have been shown to accelerate applications in domains like DSP and Cryptography, among others.

2.1 FPGA

An FPGA is a programmable logic device that can be configured to execute applications in hardware. Unlike one-time programmable devices, SRAM based FPGAs can be reconfigured at run-time to execute different applications. Every application is associated with a unique configuration, which conceptually represents the current state of the hardware and routing resources. Whenever a new application is to be executed, a new configuration replaces the previous one, thereby changing the functionality of the logic resources and the routing topology.

An island-style FPGA is shown in Fig 2-1. This kind of FPGA architecture was first proposed by Xilinx in 1984, and is widely used.

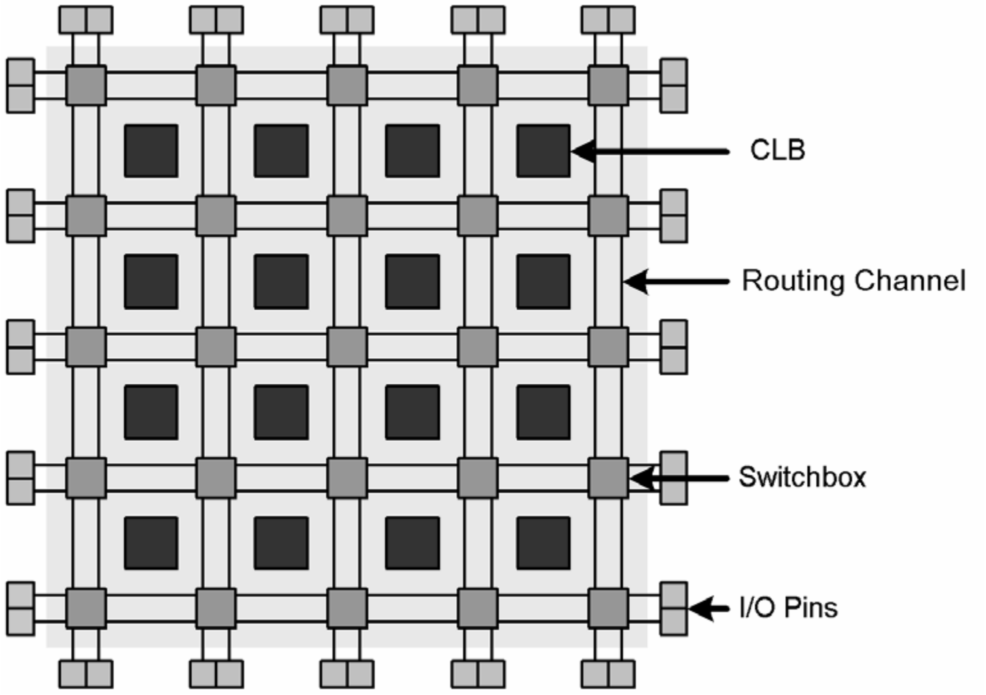


Fig 2-1: Xilinx style FPGA architecture contains an array of CLBs, switchboxes, and vertical and horizontal routing channels [11]

The computational element in the FPGA shown in Fig 2-1 is called a Configurable Logic Block (CLB). Fixed routing resources that run both vertically and horizontally surround the CLBs, and this sea of routing resources enables efficient communication among CLBs. The architecture also provides switchboxes, sites where routing resources can change direction.

The configuration of an FPGA is held in Static Random Access Memory (SRAM) cells. These cells are distributed throughout the chip, and connect to the configuration points in the FPGA. Turning the configuration points on or off can modify CLB functionality and the routing topology. An example of how routing can be configured is shown in Fig 2-2.

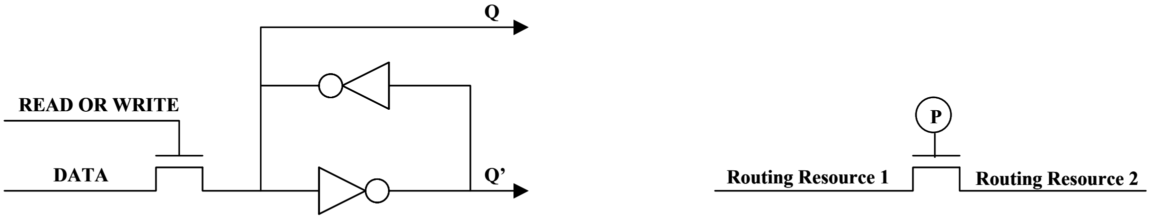


Fig 2-2: A programming bit for SRAM based FPGAs, and how it can be used to configure routing resources [3]

Here the programming bit P can be used to turn a pass-gate on/off, thus connecting/disconnecting Routing Resource #1 and Routing Resource #2. An FPGA can have several million programming sites like these, and a rich routing fabric can be constructed.

2.2 RaPiD

Reconfigurable hardware is generally constructed as an array of computational units. These computational units vary in complexity from a simple three input function to possibly a 4-bit Arithmetic Logic Unit (ALU). The size and complexity of the computational unit in a reconfigurable device is termed its granularity. Commercial FPGA architectures are generally fine- to medium-grained in nature. Fine-grained architectures are suited to bit level calculations, and can be used to implement computation structures of arbitrary bit-widths. However, such architectures perform poorly when they are used to implement datapath circuits that operate upon multiple-bit words. Logic blocks in medium-grained FPGAs like the one proposed in [8] operate on 4-bit words, thus improving upon the performance obtainable from fine-grained FPGA architectures. Even so, a considerable performance gap exists between fine-/medium-grained FPGAs and equivalent ASIC implementations.

The RaPiD [4,6] architecture was proposed a few years ago in attempt to narrow the performance gap between ASICs and FPGAs for certain application domains. RaPiD is a coarse-grained, field-programmable architecture that can be used to construct deep computational pipelines. The primary objective of RaPiD is to provide a flexible, yet efficient means of running regular, computationally intensive tasks such as those found in DSP, scientific computing and graphics.

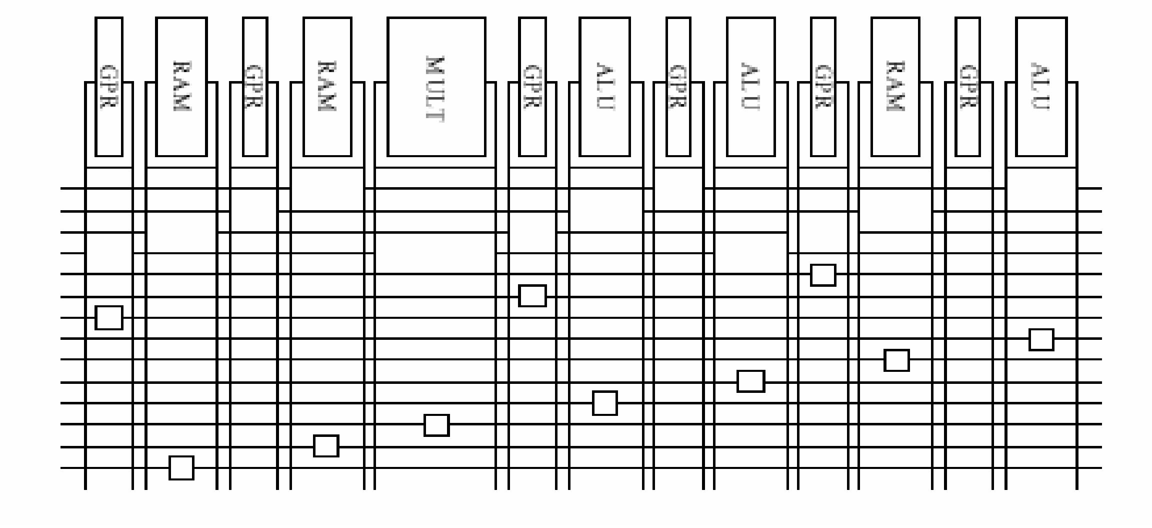


Fig 2-3: A block diagram of a RaPiD cell. Multiple cells can be tiled horizontally to create a RaPiD architecture [2]

Typically, a RaPiD datapath consists of hundreds of Functional Units (FUs) arranged in a linear, 1-D fashion (Fig 2-3). A number of FUs are presently supported by the RaPiD architecture. These include ALUs, multipliers, Random Access Memory (RAM) blocks and registers. It should be emphasized here that FUs are word-based, i.e. they operate on 16-bit wide words. This aspect of RaPiD is fundamentally different from most FPGA architectures, which are targeted at bit-oriented computations. The throughput of word-based compute-intensive applications that are mapped to the RaPiD datapath can be significantly higher than corresponding FPGA implementations. The reason for this acceleration is coarse-grained custom-built datapath FUs that are targeted to computationally intensive tasks.

The interconnect topology of RaPiD is 1-D in nature (Fig 2-3), and consists of segmented tracks. Each track is 16-bit wide because of the word-based FUs that comprise the datapath. A data-input of an FU can be driven by any one of the segmented tracks that constitute the interconnect. Similarly, the data-output of an FU can drive an arbitrary number of tracks. The majority of these tracks are connected by Bus Connectors (BCs), which are buffered, bi-directional switches. A BC can be used to drive left, drive right, or disconnect the two segments that connect to it. Further, a BC can be used to pipeline a signal, since it provides variable delay between 0-3.

2.3 Totem

As mentioned in Section 1, designing an architecture for every new domain could prove expensive in terms of design cost and time to market. In view of this observation, we are developing the Totem project. The primary objective of the Totem project is to automatically generate domain-specific reconfigurable architectures. In doing so, we will provide designers a tool-flow that would minimize human design effort for new domain-specific reconfigurable architectures.

The high-level tool-flow of the Totem project is shown in Fig 2-4. There are four main components of this tool-flow, which will eventually work in tandem to produce an optimized layout of the automatically generated reconfigurable architecture.

The Architecture Generator [2] inputs a description of the domain, and a set of constraints. Based on these inputs, it automatically generates a computational structure and an interconnect topology that it considers most suitable for the domain. The computational structure consists primarily of coarse-grained datapath elements like ALUs, RAMs, multipliers and registers, whereas the interconnect topology is a 1-D set of segmented buses. Note that the generated architecture is similar to the RaPiD architecture, and is described in structural Verilog.

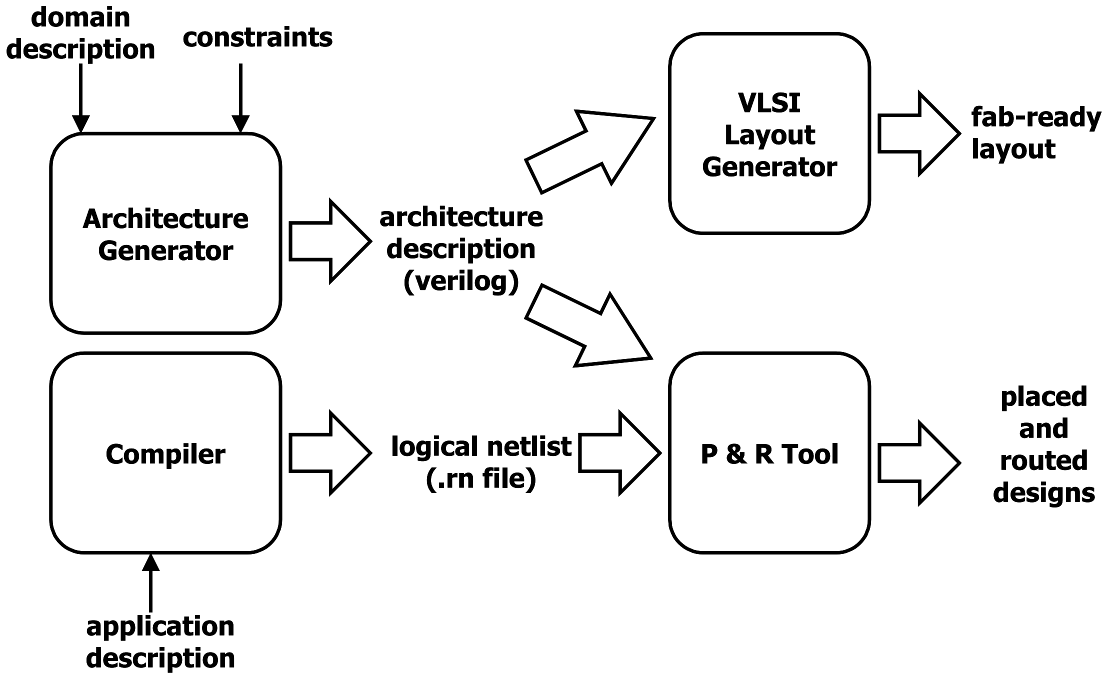


Fig 2-4: The Totem Project tool-flow. Together, the tools will generate the domain-specific architecture and its layout, and map applications to the architecture

We are using the latest release of the RaPiD compiler to synthesize logical netlists from a high-level description of an application. The Hardware Design Language (HDL) that is used to describe an application is a C-like language called RaPiD-C. The RaPiD compiler synthesizes a netlist file (.rn file) from a RaPiD-C program. This netlist file is a re-timed description of the application in terms of instances, and signals that interconnect these instances.

The main task of the Layout Generator [11] is to produce efficient layouts of the generated architecture that do not sacrifice any area or performance gains that the Architecture Generator is able to achieve over standard FPGAs. The Layout Generator also needs to be flexible enough to adapt to smaller device sizes as process technology scales down. Finally, the Layout Generator is responsible for evolving a bit-stream format that will be used by the place & route tool to generate a bitstream. This bitstream will be used to configure the architecture layout to execute a particular application.

The subject of this paper is the design and implementation of the Place & Route tool for mapping netlists to the RaPiD architecture. We have chosen the RaPiD architecture because it is very similar to an architecture that would be produced by Totem's Architecture Generator for signal processing applications. In addition, a mature compiler, and a benchmark set have already been developed for RaPiD. Thus, the testing environment for the Place & Route tool is already in place.

The Place & Route tool maps user applications to the RaPiD architecture. Specifically, the placement program attempts to determine the position that each datapath element will occupy in the architecture (hereafter, "datapath element" will be collectively referred to as an "instance"). The goal of the placement program is to ensure that the router is able to find enough routing resources to establish all necessary interconnections with minimum effort. Once the placement program creates a final placement of instances, the router uses an iterative, shortest-path algorithm to assign signals to the limited routing resources provided in the architecture. At the time of this writing, we have been able to accomplish connectivity routing successfully. There are several issues that still need to be addressed, and will be discussed in detail in later sections of the paper.

3. Approach

An important component of the design-flow for reconfigurable architectures is automated mapping software. This software converts a high-level description of the application to a bit-stream that configures the device to execute the user application. The first component of mapping software is a technology-mapping tool. This tool maps the user application to the logic elements that are provided in the reconfigurable architecture. The next component is the placement tool, which determines a physical assignment of logic elements to hardware resources. Finally, the routing tool is responsible for assigning signals to routing resources in order to successfully route all signals while meeting performance requirements.

Place & Route for FPGAs (and reconfigurable devices in general) is a challenging problem. This is because the resources that are available for doing logic and routing are fixed at the time of fabrication. Unlike standard-cell designs in which a successful routing can always be found by increasing channel widths, FPGA routers often fail to route all the signals in a design due to scarcity of routing resources. The role of the placement phase is thus extremely important, because a poorly placed design can result in routing failure. In addition, a poor placement can result in long, circuitous routes that adversely affect the delay characteristics of the design.

Several research-groups have studied the problems associated with place & route for FPGAs. [1,10] describe placement strategies for FPGAs, while [9] proposes a routing approach. [1,4] discuss complete tool-flows that have been developed for mapping applications to FPGAs. In the two following sub-sections, we will describe the approach we used to place and route netlists on the RaPiD architecture.

3.1 Placement

The two inputs to the placement program are descriptions of the RaPiD netlist (.rn file) and the architecture. The placement of the instances is determined using a Simulated Annealing [12] algorithm. This algorithm operates by taking a random initial placement of physical elements, and repeatedly moving the location of a randomly selected element. The move is accepted if it improves the overall cost of the placement. In order to avoid getting trapped in local minima, non-improving moves are also sometimes accepted. The temperature of the annealing algorithm governs the probability of accepting a "bad" move at that point. The temperature is initially high, causing a large number of bad moves to be accepted, and is gradually decreased until no bad moves will be accepted. A large number of moves are attempted at each temperature.

A good cooling schedule is essential to obtain high-quality solutions in a reasonable computation time with simulated annealing. For our placement program, we used the cooling schedule developed for the VPR tool-suite [1] at the University of Toronto. The various parameters that define the placement algorithm are determined as follows:

Initial Temperature: Let N be the number of instances in the application. A random initial placement is first created, following which N random moves are made. The standard deviation of the cost of the placement after each move is calculated, and the initial temperature T is set to twenty times the standard deviation.

Cooling Schedule: The temperature after each iteration is computed as $T_{new} = \alpha * T_{old}$. The value of α depends on the fraction of attempted moves that are accepted (R) at T_{old}. The variation of α with respect to R is as under:

Fraction of moves accepted R	α
$R > 0.96$	0.5
$0.8 < R \leq 0.96$	0.9
$0.15 < R \leq 0.8$	0.95
$R \leq 0.15$	0.8

It has been shown that it is desirable to keep R near 0.44 for as long as possible [1]. This is achieved by using a range limiter that imposes an upper bound D on the number of positions that an instance can be moved across at a given temperature. D is updated at every temperature based on the following relationship:

$$D_{new} = D_{old} * (1 - 0.44 + R)$$

A prolonged quenching step that greedily seeks lower-cost placements is undertaken at the end of the anneal.

Number of moves at each temperature: This quantity directly affects the quality and run-time of the placement program. As advocated by VPR, we make $10 * N^{1.33}$ moves at every temperature. This number results in good placements in a reasonable run-time.

The development of a representative cost-function for the placement program is an interesting problem. Since the number of routing tracks in the interconnect fabric of the RaPiD architecture is fixed, we capture the quality of the placement by means of a *cutsize* metric. The *cutsize* at a vertical partition of the datapath is defined as the number of signals that need to be routed across that partition for a given placement of datapath elements. The *max_cutsize* is defined as the maximum *cutsize* that occurs at any vertical partition of the datapath. The *total_cutsize* is defined as

$$total\_cutsize = \sum_{j=1}^{j=Y} (cut\_size)_j$$

where Y is the number of datapath elements that are placed on the architecture. The *avg_cutsize* is then defined as

$$avg\_cutsize = total\_cutsize/Y$$

Both *max_cutsize* and *avg_cutsize* are important estimates of the routability of a circuit. Since the RaPiD architecture provides a fixed number of tracks for routing signals, it is necessary to formulate a placement cost function that favorably recognizes a move that decreases *max_cutsize*. At the same time, it is clear that a simple cost function that attempts to reduce only *max_cutsize* will be inadequate. A cost function that is determined only by *max_cutsize* will not be able recognize changes in *avg_cutsize*. This means that the annealer will accept moves that increase *avg_cutsize*, but do not change *max_cutsize*. Such zero-cost moves may cumulatively increase the overall congestion in the datapath considerably, thus making it harder for the annealer to find the sequence of moves that will bring down *max_cutsize*. It can thus be concluded that *avg-cutsize* should also contribute to the cost of a placement. Reducing *avg_cutsize* not only reduces overall congestion in the datapath, but also brings down the total wire-length. The cost function can be mathematically formulated as follows

$$cost = w*max\_cutsize + (1-w)*avg\_cutsize$$

where $0 \leq w \leq 1$. The choice for the exact value of the weighting parameter *w* is presented in Section 4 of this paper.

3.2 Routing

We use the Pathfinder [9] algorithm to route signals in RaPiD netlists after completion of the placement phase. Our adaptation is an iterative scheme, and consists of two parts. The signal router routes individual signals based on Prim's algorithm, which is used to find a minimum spanning tree (MST) over an undirected graph. The global router adjusts the cost of each resource at the end of a routing iteration depending on the demands placed by signals on that routing resource. During the first routing iteration, signals are free to share as many routing resources as they need. However, the cost of using a shared routing resource is gradually increased during later iterations, and this increase in cost is proportional to the number of signals that share that resource. Thus, this scheme forces signals to negotiate for routing resources. A signal can use a high-cost resource if all remaining resource options are in even higher demand. On the other hand, a signal that can take an alternative, lower-cost route is forced to do so because of competitive negotiation for shared resources.

The routing resources in the RaPiD architecture are represented as a graph, with the track-segments constituting the nodes of the graph. The cost of using a node *n* in a route during a given iteration is determined using the following relationship:

$$Cn = (Bn + Hn) * Pn$$

where *Bn* is the base cost of using the node *n*, *Hn* is a cost term related to the number of times that *n* was shared in earlier iterations, and *Pn* represents the number of other signals that are sharing *n* during that iteration. During the first iteration of the global router, the *Pn* value for every node is assigned a value of one, whereas *Hn* is set to zero. This allows multiple signals to share routing nodes without incurring any penalty. However, the *Pn* value for a node is slowly increased in later iterations of the global router depending on the extent to which that node is shared. At the same time, the *Hn* factor for a node is increased slightly at the end of every iteration during which that node is shared. A detailed explanation of how *Pn* and *Hn* resolve congested nodes appears in the paper [9] that proposed the Pathfinder algorithm.

Fig 3-1 presents the step-by-step details of the Negotiated Congestion (NC) scheme of the Pathfinder algorithm. The signal router commences at step 2. The existing routing tree RTi of signal i is ripped up and initialized to the signal's source. A loop over all sinks of the signal is started at step 5. Steps 7 – 12 outline a breadth-first search for the sink tij closest to the source si. After a sink is found, a backtrace from the sink to the source is initiated, and the Cn value of every node in the backtraced path is updated.

Also, all nodes on the backtraced path are added to the current routing tree. Thus, every time a sink is found, all nodes in the newly updated partial routing tree of the signal are used as potential sources for routes to the remaining sinks.

```
While shared resources exist (global router)
   Loop over all signals i (signal router)
      Rip up routing tree RTi
      RTi <- Si
      Loop until all sinks tij have been found
         Initialize priority queue PQ to RTi to cost 0
         Loop until new tij is found
            Remove lowest cost node m from PQ
            Loop over fanouts n of node m
               Add n to PQ at cost Cn + Pim
         End
      End
      Loop over nodes n in path tij to Si (backtrace)
         Update Cn
         Add n to RTi
      End
   End
End
```

Fig 3-1: Negotiated Congestion (NC) algorithm used to eliminate congestion [9]

4. Experimental Setup

There were two inputs to the place and route tool (Fig 2-4). The first was a netlist file (in the .rn format) generated by the RaPiD compiler. The second input was a description of the RaPiD architecture expressed in structural Verilog. The two input files were read in by the place and route tool using lexical-analyzer (lex) & yet-another-compiler-compiler (yacc). The remainder of the tool was implemented in C++.

We determined that the most appropriate metric for testing the functionality of our tool would be the minimum number of routing tracks that were required to successfully route each netlist in a set of benchmark applications. The set of benchmark netlists was obtained from the RaPiD group, and consisted of DSP applications like FIR filters, FFTs, sorting, matrix multipliers and image filters. Table 4-1 lists statistics of the benchmarks that we used in our experiments.

NETLIST	DESCRIPTION	NUM INSTANCES	NUM SIGNALS
decNsR.rn	decimation stage for radar app	14	10
psd.rn	PSD stage for radar app	26	14
limited.rn	limited matrix multiply	42	57
matmult_bit.rn	generic matrix multiply	37	60
firTM_2nd.rn	time-muxed 4-stage FIR	35	50
firsm.rn	16-stage FIR	64	63
firsymeven.rn	symmetric 16-stage FIR	96	95
sort_G.rn	1D sorting	118	119
sort_2D_RB.rn	2D sorting	84	76
fft16.rn	16-point FFT	64	74
fft64.rn	64-point FFT	240	246
img_filt.rn	image filter	142	148
med_filt.rn	5 x 5 median filter	101	57
sync.rn	synchronization stage for OFDM	318	306

Table 4-1: statistics of benchmark netlists

For purposes of nomenclature, we defined two types of segmented routing tracks. In Fig 2-3, a sequence of tracks that lay in the same horizontal line and did not connect to a BC was collectively referred to as a "short" track. On the other hand, a sequence of tracks that lay in the same horizontal line and DID connect to a BC was collectively termed a "long" track. The only exception was the short track (top-most track in Fig 2-3) that consisted of segments that feed back the output of an instance to its input. This track was not considered either a short track or a long track. Thus, there were 4 short tracks, 10 long tracks, and 14 (10 + 4) tracks in total in Fig 2-3.

Test architectures for our tool were generated using software provided by Northwestern University graduate student Katherine Compton. This software was capable of generating RaPiD architectures that had a user-specified number of short tracks per cell, long tracks per cell, and bus-connectors per long track per cell. We used the architecture generation software to create RaPiD architectures that had between 5 – 35 tracks. Approximately 2/7th of the tracks in every architecture were short tracks, and 5/7th of the tracks were long tracks. Each short track consisted of 4 segments per cell, and each long track had 1 BC per cell. We picked these ratios to be consistent with the ratios proposed for the RaPiD cell in previous work [4].

5. Results

5.1 Effect of weighting parameter *w*

The cost function that we used to determine the cost of a placement is reproduced from Section 3 as under:

$$cost = w*max\_cutsize + (1-w)*avg\_cutsize$$

As a first step, we empirically determined the value of the weighting parameter *w*. We used only a sub-set of the available benchmarks for our experiments, so that we could detect trends in performance in a reasonable run-time. Table 5-1 shows the minimum number of tracks required to successfully route each netlist across a range of values for the weighting parameter *w*. There were two important observations that we made from this data. First, the minimum number of tracks required to route a benchmark did not vary appreciably for $w \leq 0.9$. The other observation was that there was a substantial increase in the minimum number of tracks required when $w = 1.0$. A value of $w = 1.0$ represented the case in which our cost function was determined purely by *max_cutsize*.

NETLIST	*W* = 0.0	*W* = 0.1	*W* = 0.3	*W* = 0.5	*W* = 0.7	*W* = 0.9	*W* = 1.0
limited.rn	7	7	7	7	7	7	9
psd.rn	7	7	7	7	7	7	7
firTM_2nd.rn	9	9	8	9	9	9	11
matmult_bit.rn	9	9	8	8	9	9	11
firsymeven.rn	9	9	10	10	10	9	21
cascade.rn	10	10	10	10	10	10	12
fft16.rn	12	12	12	12	12	12	16
img_filt.rn	15	15	15	15	15	15	32
Geometric Mean	**9.46**	**9.46**	**9.31**	**9.44**	**9.58**	**9.46**	**13.30**

Table 5-1: minimum number of tracks required to route a netlist for different values of *w*

The trends in Table 5-1 were consistent with our claim in Section 3.1 that the cost of a placement should be determined by *max_cutsize* and *avg_cutsize* together, rather than by *max_cutsize* alone. This is because a cost function that was solely determined by the maximum cutsize would not be able to detect changes in the average cutsize across the datapath. Consequently, the annealer could allow the average congestion in the datapath to get dangerously high, thus preventing it from finding the sequence of moves that would have decreased the maximum cutsize.

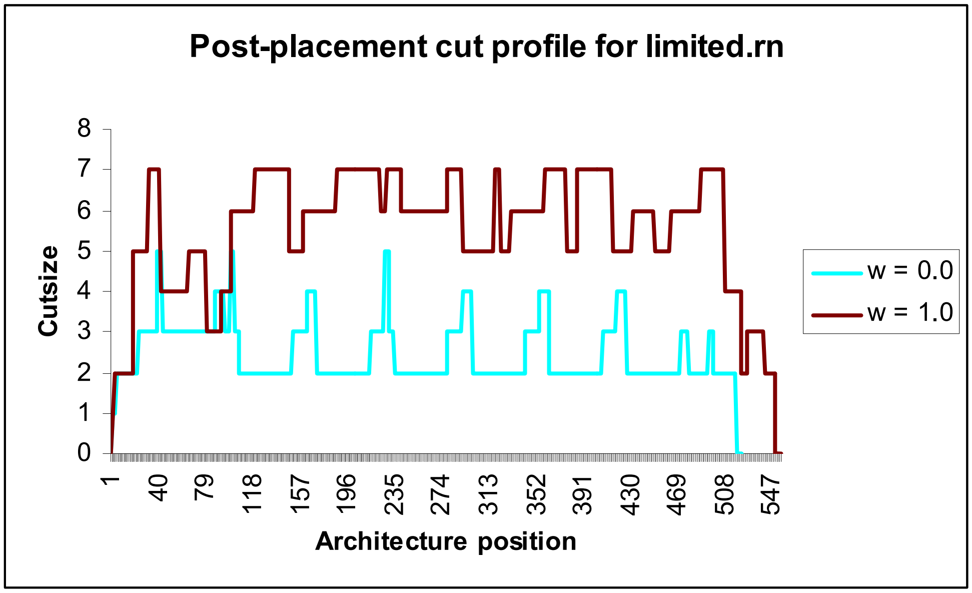


Fig 5-1: Post-placement cut profile for the benchmark limited.rn for *w*=0.0 and *w*=1.0

This effect was particularly pronounced in netlists in which the maximum cutsize occurred in several regions of the datapath simultaneously, and can be seen in Fig 5-1 which shows the post-placement cut profile of the netlist limited.rn for values of *w*=0.0 (blue profile) and *w*=1.0 (brown profile). The average congestion when *w*=1.0 was extremely high, and there were several regions in the datapath that were at maximum cutsize.

5.2 General Performance

We present the performance of our place & route tool in this section. We used a value of *w*=0.3 for our experiments, since this value of *w* produced the geometrically minimum number of routing tracks required to route a sub-set of the benchmark-suite (Table 5-1). Table 5-2 shows the comparison between the maximum cutsize of the final placement and the minimum number of routing tracks that were required to successfully route a netlist.

NETLIST	MAX CUTSIZE AFTER PLACEMENT *(MAX_CUT)*	MIN NUMBER ROUTING TRACKS REQD (*MIN_TRACKS)*	*MIN_TRACKS/MAX_CUT*
psd.rn	4	7	1.75
firsm.rn	5	7	1.4
decNsR.rn	5	7	1.4
limited.rn	5	7	1.4
matmult_bit.rn	6	8	1.33
firTM_2nd.rn	6	8	1.33
firsymeven.rn	6	10	1.66
sort_2D_RB.rn	7	12	1.72
sort_G.rn	8	12	1.5
fft16.rn	8	12	1.5
med_filt.rn	8	13	1.62
img_filt.rn	10	15	1.5
sync.rn	14	19	1.36
fft64.rn	15	24	1.6

Table 5-2: Minimum number of routing tracks required to route each netlist in the benchmark-set

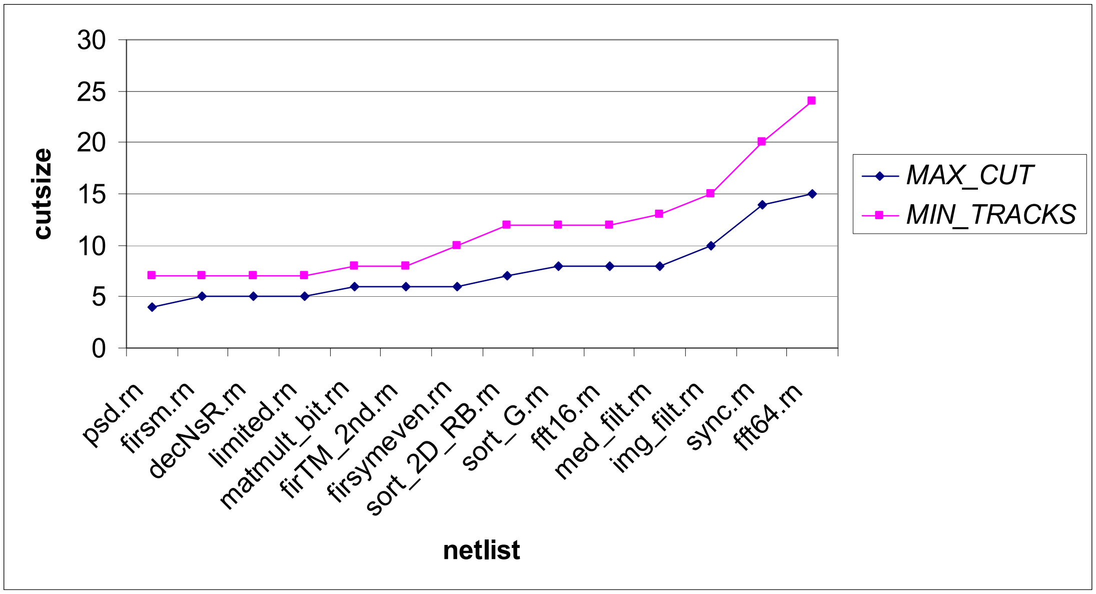


Fig 5-2: Variations in *MAX_CUT* and *MIN_TRACKS* across the benchmark set

The trend in the *MIN_TRACKS/MAX_CUT* ratio in Table 5-2 suggested that our placement tool modeled routing constraints reasonably well. This can be seen from the fact that *MIN_TRACKS* was approximately a fixed multiplicative factor off from *MAX_CUT* for each netlist. Further, Fig 5-2 shows that the variations in *MAX_CUT* and *MIN_TRACKS* across the benchmark set were smooth and consistent.

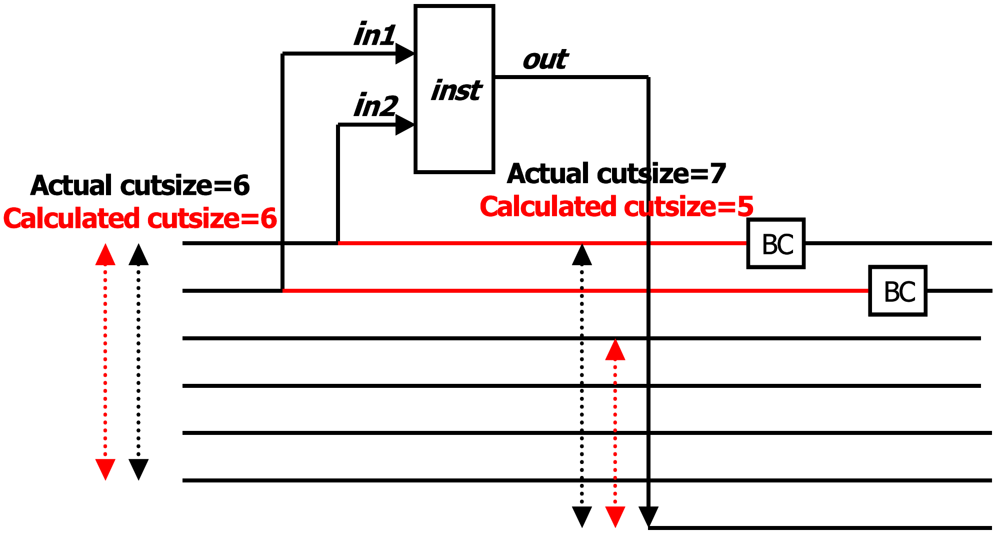


Fig 5-3: An example of Signal Spillover

5.3 Signal Spillover

As a next step, we tried to determine why *MIN_TRACKS* differed from *MAX_CUT* by a multiplicative factor at all. Fig 5-3 illustrates an explanation for part of this difference, and shows an instance mapped to RaPiD. For the sake of simplicity, only long tracks have been shown. Consider a case in which 6 signals occupied long tracks to the left of the instance, and that 2 of the 6 signals terminated at *inst*. Our placement tool calculated a cutsize of 6 on the left of *inst*, and this value matched the actual number of occupied routing tracks. However, the placement tool calculated a cutsize of 5 on the right of *inst*, when the number of occupied tracks was actually 7. This was because the placement tool did not recognize that the 2 routing tracks that were occupied by the terminating signals *in1* and *in2* were actually not available for routing signal *out*, and that this would force the router to seek a 7th track for routing *out.* Thus, we concluded that we needed to alter our placement model to account for this phenomenon that we called “Signal Spillover”.

Since there was no way of knowing the actual assignment of signals to specific routing tracks during the placement phase, we attempted to evolve a probabilistic model for Signal Spillover. Fig 5-4 shows an instance mapped to a RaPiD architecture that has 5 long tracks.

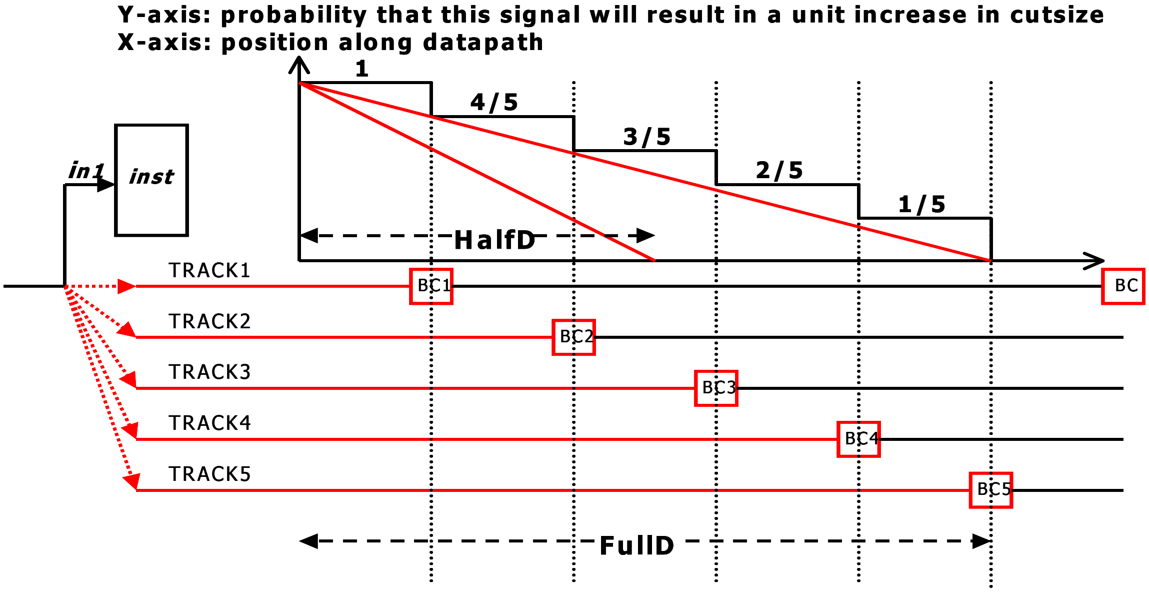

Y-axis: probability that this signal will result in a unit increase in cutsize
X-axis: position along datapath
1
4/5
3/5
2/5
1/5
HalfD
in1
inst
TRACK1
BC1
BC
TRACK2
BC2
TRACK3
BC3
TRACK4
BC4
TRACK5
BC5
FullD

Fig 5-4: Probabilistic model for Signal Spillover

We assumed that *in1*could eventually occupy any of the 5 long tracks shown in Fig 5-4 with equal probability. Under this assumption, the likelihood of incrementing the cutsize between *inst* and BC1 was 1, between *inst* and BC2 was 4/5, between *inst* and BC3 was 3/5, and so on. Note that this analysis held true irrespective of track permutation.

For experimental purposes, we modeled Signal Spillover by means of a linear decay shown in red in the graph of Fig 5-4. We thought that a linear decay would appropriately approximate the multiple step-profiles that would result because of the variation in the actual distance between *inst* and BC1. Table 5-3 shows the minimum number of tracks required to successfully route a sub-set of the benchmark suite with the new cost model in place. The FullD column shows the minimum number of required tracks when cutsizes were incremented based on a linear decay that spanned a length FullD as shown in Fig 5-4. The HalfD column shows the minimum number of required tracks when cutsizes were incremented based on a linear decay that spanned a length FullD/2. Finally, the Basic column shows the minimum number of required tracks when we used our original cost function.

Netlist	FullD	HalfD	Basic
psd.rn	7	7	7
limited.rn	7	7	7
firsymeven.rn	9	9	10
matmult_bit.rn	8	8	8
firTM_2nd.rn	9	8	8
sort_2D_RB.rn	11	11	12
fft16.rn	12	11	12
med_filt.rn	12	12	13
img_filt.rn	16	15	15
Normalized Geometric Mean	**0.99**	**0.96**	**1.0**

Table 5-3: Performance changes with Signal Spillover placement cost model

It can be seen from Table 5-3 that a cost model that took Signal Spillover in to account did in fact result in an improvement over the original cost model for firsymeven.rn, sort_2D_RB.rn and med_filt.rn. Table 5-2 shows that these 3 netlists exhibited a significant difference between *MAX_CUT* and *MIN_TRACKS*, indicating pronounced Signal Spillover effects in these netlists. At the same time, the FullD model produced worse results for firTM_2nd.rn and img_filt.rn. Recall here that the FullD model was based on the assumption that a long signal could be routed on any long track with equal probability. This was probably not an accurate model of the router, because if given the choice, the router would negotiate a signal like *in1*(Fig 5-4) on to a track that terminated at an earlier BC. The HalfD model may have reflected this bias better than the FullD model, and thus produced better overall results.

5.4 Short Track utilization

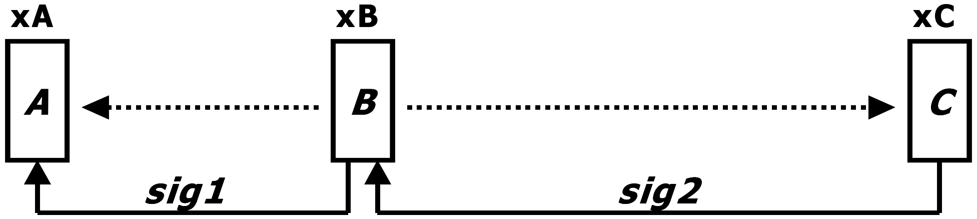


Fig 5-5: Instance B can go anywhere between A and C.

Fig 5-5 shows three instances A, B and C at positions xA, xB and xC respectively. The 3 instances are interconnected by signals *sig1* and *sig2*, and xA < xB < xC. Further, define *short_length* to be the length of a short segment. In such a case, our placement tool would increment the cutsize between positions xA and xC by 1, regardless of xB (subject to the condition that xA < xB < xC). However, note that the value of xB would in fact influence the router in assigning signals *sig1* and *sig2* to routing tracks. Fig 5-6 shows a placement of instance B right next to instance A such that xB – xC > *short_length.*

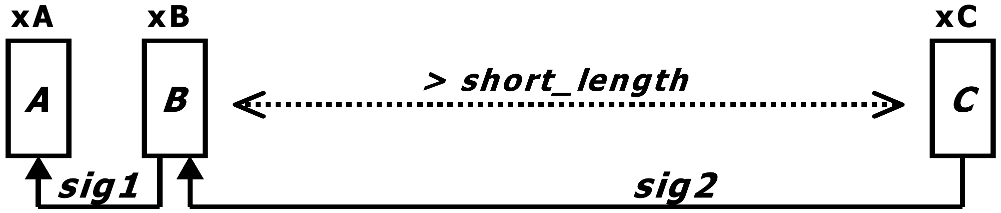


Fig 5-6: Instance B is adjacent to A. *sig1* is routed on a short track, and *sig2* on a long track

In this case, the router would assign *sig1* to a short track, and *sig2* to a long track. Fig 5-7 shows the situation in which xA-xB < *short_length* and xB-xC < *short_length*. In such a situation, the router would assign both *sig1* and *sig2* to short tracks, thus freeing up the long track that it would have used for the placement shown in Fig 5-6. In its original form, our placement tool was incapable of recognizing that the placement in Fig 5-7 was potentially better than the placement in Fig 5-6.

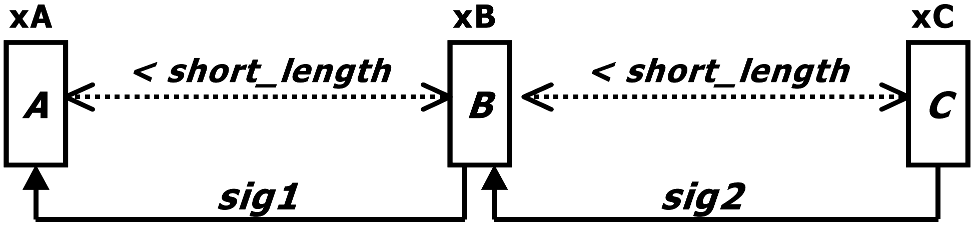


Fig 5-7: A placement for which both *sig1* and *sig2* are routed on short tracks

The approach that we adopted to improve short track utilization was to make short signals (signals that spanned < *short_length* positions) partially "free" during the placement cost computation. For the purposes of calculating *max_cutsize,* we considered the contribution of all long signals and those short signals that lay in regions of the datapath in which the number of short signals *exceeded* the number of short tracks available. Thus, for every vertical partition of the datapath, if the number of short signals spanning the partition exceeded the number of available short tracks, we incremented the cutsize by the difference between the number of short signals and the number of short tracks. On the other hand, if the number of short signals spanning the partition was no greater than the number of available short tracks, we left the cutsize unchanged. Essentially, this meant that we attempted to model only the demands that were placed on the long tracks provided in the architecture. In the absence of short signal congestion, this scheme would increment the cutsizes between xB and xC in Fig 5-6, and leave the cutsizes between xB and xC unchanged in Fig 5-7.

The *avg_cutsize* computation remained the same as that for our original cost function. Contributions due to all long and short signals were considered while calculating *avg_cutsize.* We observed that calculating *avg_cutsize* from contributions due to long signals and excess short signals was not an accurate way of representing the overall congestion in the datapath. This was because such a cost model would fail to recognize moves that did not affect cutsize, but did in fact change the span of short signals. Thus, moves that would have increased short signal wirelength without affecting the cutsize would be accepted, and this would adversely impact the routability of short signals.

Table 5-4 shows the minimum number of routing tracks required to successfully route a set of benchmarks. The FreeShort column shows the minimum number of required routing tracks when short signals were considered free, whereas the Basic column shows the minimum number of required routing tracks when we used our basic placement cost function.

Netlist	FreeShort	Basic
limited.rn	7	7
firTM_2nd.rn	9	8
matmult_bit.rn	8	8
firsymeven.rn	9	10
fft16.rn	12	12
sort_2D_RB.rn	11	12
img_filt.rn	15	15
Normalized Geometric Mean	**0.99**	**1.00**

Table 5-4: Minimum number of routing tracks required when short signals were considered free

Table 5-4 shows that making short signals free resulted in improvements in firsymeven.rn and sort_2D_RB.rn. However, results for firTM_2nd.rn deteriorated. Consider the case shown in Fig 5-8, in which two instances GPR1 and ALU1 were placed at the extremities of the short track shown in blue. We observed that only 1 two-terminal short signal that had its terminals at GPR1 and ALU1 could actually be routed on a short track. All other two-terminal short signals that had terminals at GPR1 and ALU1 would have to necessarily be routed on long tracks. In such a case, our placement tool may have underestimated the cutsize between GPR1 and ALU1, because it considered multiple short signals between GPR1 and ALU1 to be free during cutsize calculations.

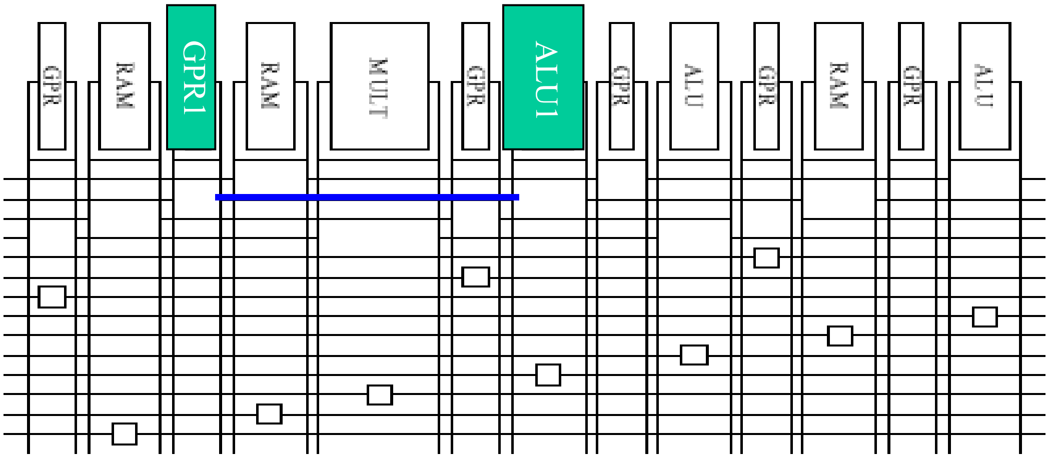


Fig 5-8: The RaPiD architecture has staggered short segments

6. Conclusions

In conclusion, we believe that we successfully evolved an approach for the placement of benchmark netlists on the RaPiD architecture. Further, we used the Pathfinder [9] algorithm to route each netlist. Table 5-2 and Fig 5-2 show that the minimum number of required routing tracks (*MIN_TRACKS*) scaled well with the maximum cutsize (*MAX_CUT*) after placement, and that *MIN_TRACKS* differed from *MAX_CUT* by a multiplicative factor. We proceeded to explain this difference in terms of Signal Spillover, and evolved a probabilistic placement cost model to take this phenomenon in to account. We also developed a placement cost model that attempted to reduce the number of short signals that could have potentially occupied long tracks.

7. Future Work

7.1 Pipelined Signals

The RaPiD architecture has a pipelined interconnect topology, and most target applications are deeply pipelined. Fig 7-1 illustrates an example of a pipelined signal *sig*.

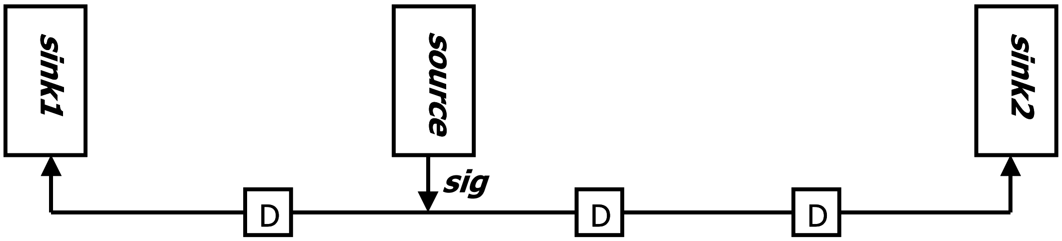


Fig 7-1: An example of a pipelined signal

The objective of the router when routing pipelined signals is to find a minimal route for a signal while meeting all pipelining constraints. For example, in Fig 7-1 the signal *sig* has to be delayed by 1 clock cycle before reaching *sink1,* and by 2 clock cycles before reaching *sink2*. Thus, for every pipelined signal, the router needs to find the shortest route that goes through a necessary number of delay elements. Note that it may be possible to find a shorter route that establishes the necessary connectivity between the source and sinks of a signal, but does not provide sufficient delay resources between terminals. However, such a route would result in a timing violation, and consequent functionality failure.

The present version of our place & route tool does not take pipelining considerations in to account, and only achieves connectivity routing. At the same time, Table 7-1 shows that the present interconnect topology of the RaPiD architecture does not provide enough delay resources to be able to satisfy the pipelining constraints imposed by signals in img_filt.rn and med_filt.rn. Consequently, we may need to change our placement cost model, and develop a new routing algorithm so that we can successfully route pipelined signals.

NETLIST	NUM SIGNALS	NUM PIPELINED SIGNALS	NUM VIOLATIONS
decNsR.rn	10	0	0
psd.rn	14	0	0
limited.rn	57	7	0
matmult_bit.rn	60	3	0
firTM_2nd.rn	50	6	0
firsm.rn	63	15	0
firsymeven.rn	95	16	0
sort_G.rn	119	0	0
sort_2D_RB.rn	76	0	0
fft16.rn	74	0	0
fft64.rn	246	0	0
img_filt.rn	148	13	3
med_filt.rn	57	5	5

Table 7-1: Pipelined signal violations

7.2 Architecture Exploration

Another area that we will actively investigate in the future is the interconnect structure provided in the RaPiD architecture (Fig 2-3). Specifically, we are going to study how the ratio between short tracks and long tracks affects the minimum number of routing tracks required to route each benchmark netlist. We also intend to determine whether the length of short segments affects the minimum number of routing tracks required.

The number of BCs per long track per cell could play a major role in determining the overall performance of placed & routed netlists. Firstly, increasing the number of BCs per track per cell could reduce the effect of Signal Spillover. This could result in an improvement in the minimum number of tracks required to route a netlist. Another reason for increasing the number of BCs per track per cell could be pipelining constraints. It would be interesting to study the extent to which the number of BCs per track per cell needs to be increased for a simple routing algorithm to find shortest routes while meeting pipelining constraints.

8. Acknowledgments

We would first like to thank Carl Ebeling and the RaPiD group at the University of Washington (UW) Computer Science and Engineering (CSE) department for providing the set of benchmark applications, and Katherine Compton at Northwestern University for giving us a copy of the architecture generation software that we used to generate different RaPiD architectures. We would also like to thank Carl Ebeling (UW CSE) again, and Mike Scott (Quicksilver Technologies) for answering queries about the RaPiD architecture. Further, we thank UW graduate students Chandra Mulpuri and Ken Eguro for their advice on programming CAD applications in general. Lastly, the author would personally like to thank his parents and Mary Ann Krug for the firm pillars of support that they have always been!

This research was funded by a grant from the National Science Foundation (NSF).

References:

[1] V. Betz and J. Rose, ``VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Seventh International Workshop on Field-Programmable Logic and Applications,* pp 213-222, 1997

[2] K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001

[3] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software" (PDF), submitted to *ACM Computing Surveys,* 2000

[4] Darren C. Cronquist, Paul Franklin, Chris Fisher, Miguel Figueroa, and Carl Ebeling. "Architecture Design of Reconfigurable Pipelined Datapaths," *Twentieth Anniversary Conference on Advanced Research in VLSI,* pp 23-40, 1999

[5] Darren C. Cronquist and Larry McMurchie. "Emerald - An Architecture-Driven Tool Compiler for FPGAs", *ACM/SIGDA Fourth International Symposium on Field-Programmable Gate Arrays*, pp 144-150, 1996

[6] Carl Ebeling, Darren C. Cronquist, Paul Franklin. "RaPiD - Reconfigurable Pipelined Datapath", *6th International Workshop on Field-Programmable Logic and Applications,* pp 126-135, 1996

[7] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and Reed Taylor "PipeRench: A Reconfigurable Architecture and Compiler" in *IEEE Computer*, Vol.33, No. 4, pp 70-77, April 2000.

[8] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, B. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Application*s*", *ACM/SIGDA Seventh International Symposium on FPGA*s, pp 135-143, 1999.

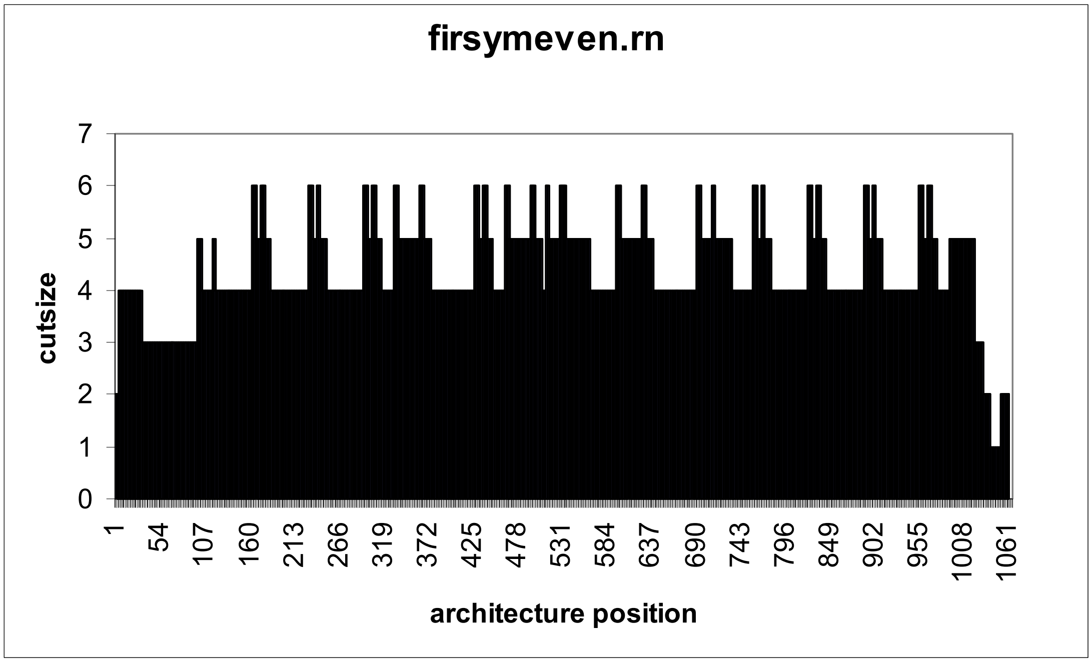
[9] Larry McMurchie and Carl Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM Third International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995

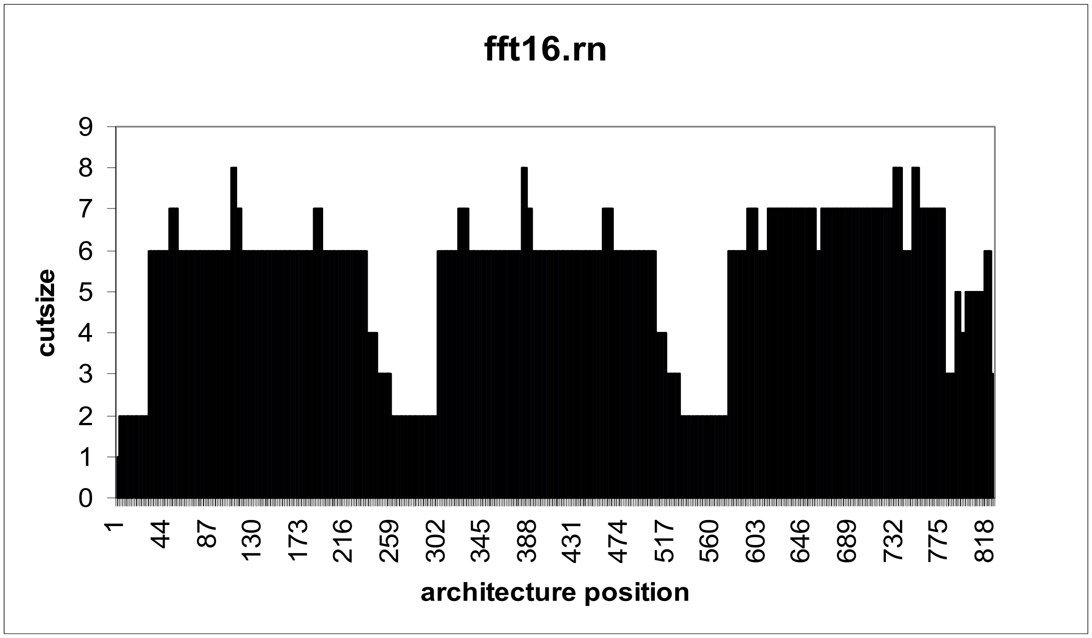
[10] C. Mulpuri and S. Hauck, "Runtime and Quality Tradeoffs in FPGA Placement and Routing", *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays,* pp 29-36, 2001

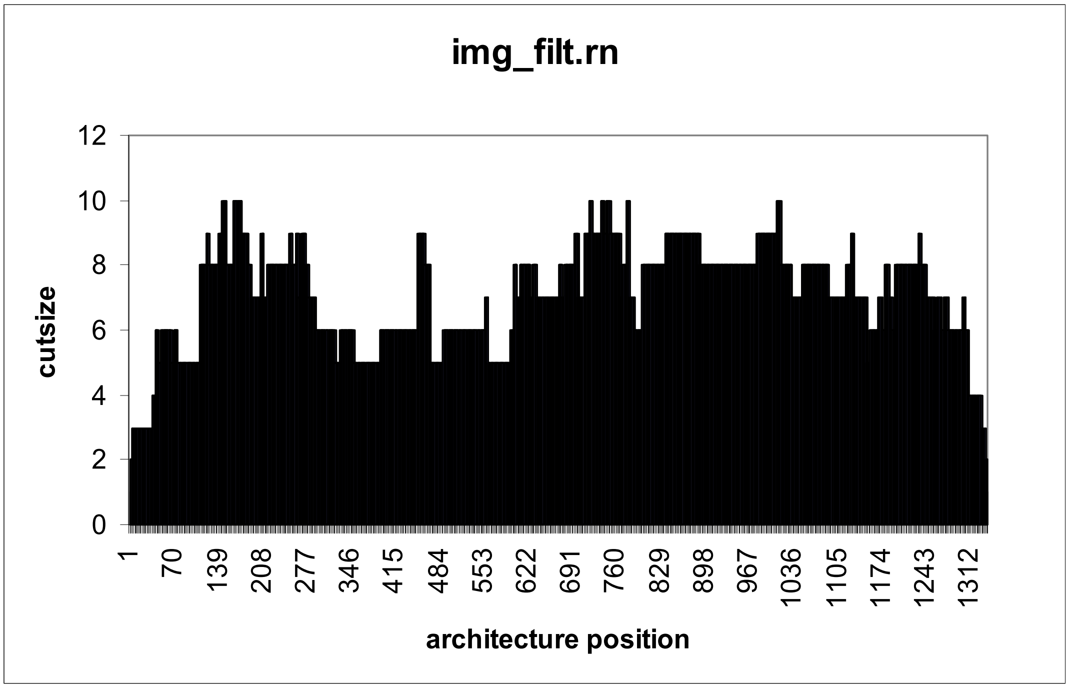
[11] Shawn Phillips, "Automatic Layout of Domain Specific Reconfigurable Subsystems for System-on-a-Chip (SOC)", *Master's Thesis, Northwestern University,* July 2001

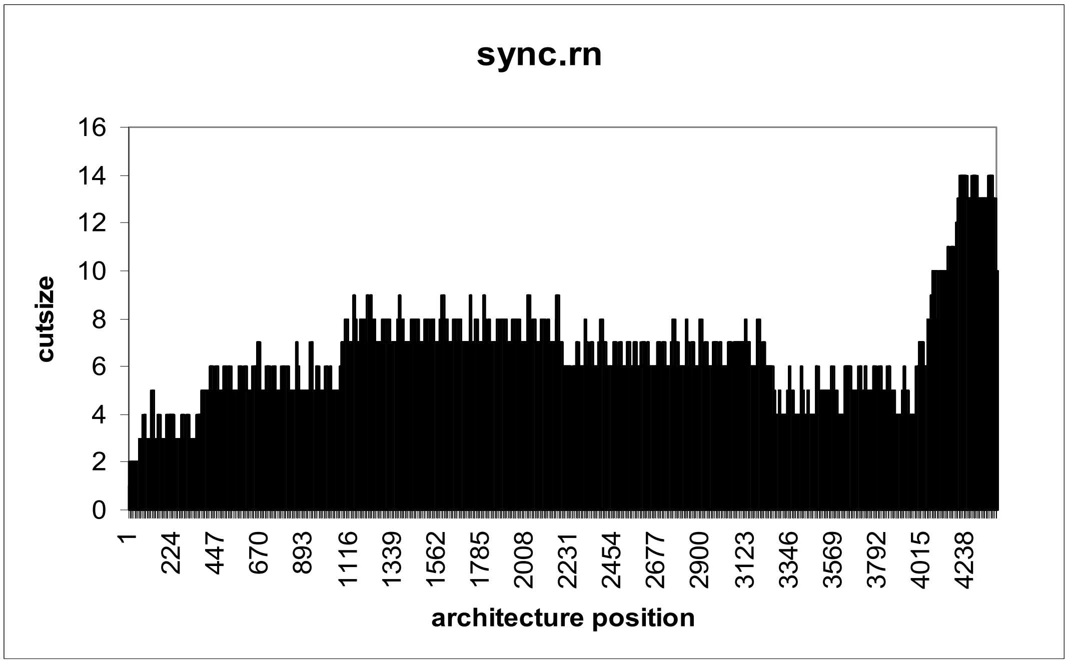
[12] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston, MA: 1988

Appendix A: Cut Profiles






img_filt.rn
cutsize
0
2
4
6
8
10
12
1
70
139
208
277
346
415
484
553
622
691
760
829
898
967
1036
1105
1174
1243
1312
architecture position


sync.rn
cutsize
0
2
4
6
8
10
12
14
16
1
224
447
670
893
1116
1339
1562
1785
2008
2231
2454
2677
2900
3123
3346
3569
3792
4015
4238
architecture position