Fast Resource Sharing in VLSI Routing

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Dirk Müller

aus Aachen

im September 2009

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Erstgutachter: Professor Dr. Jens Vygen
Zweitgutachter: Professor Dr. Bernhard Korte

Tag der Promotion: 18. Dezember 2009

Erscheinungsjahr: 2010

Acknowledgments

I like to express my gratitude to my supervisors, Professor Dr. Bernhard Korte and Professor Dr. Jens Vygen. This work would not have been possible without their guidance and valuable ideas, and the optimal working conditions that the Institute for Discrete Mathematics at the University of Bonn offers under their leading.

I am very thankful to my past and present colleagues at the institute, especially Dr. Sven Peyer, Christian Panten, Christian Schulte and Jun.-Prof. Dr. Tim Nieberg. The friendly and efficient working atmosphere in our team contributed much to the success of BonnRoute® in the last years.

I am thankful to all people at IBM who shared their knowledge with me and provided me with a lot of practical background information, in particular Karsten Muuss, Dr. Jürgen Köhl, Gustavo Tellez and Dr. Matthias Ringe, and again Dr. Sven Peyer, of course. Their expertise and continuous suggestions for improving BonnRoute® have been very valuable.

Sincere thanks go to Dr. Asmus Hetzel. I had the exciting experience to work together with him on the project of writing a completely new global router. His work was essential for putting this new global router into operation at Magma Design Automation, Inc.

Special thanks go to my colleagues for the support they gave me in the recent months in finalizing this thesis, especially for proof-reading important parts and making valuable comments. I explicitely want to mention Christian and Christian, Ulrich, Sven, Tim, Nico, Daniel and Thomas.

I thank my parents who made a care-free student life possible, and my whole family for being a source of advice and recreation.

My biggest thanks however go to Cathrin for her loving encouragements and tolerance of the many evenings I came home late during the recent weeks and months working towards completion of this thesis.

Contents

Chapter 1

Introduction

The design of VLSI logic circuits is fascinating in itself because of the enormous complexity and multitude of structures packed on an area of five square centimeters or smaller, and it is a fascinating field for applying mathematics. Millions of elementary devices, called *standard cells*, and a few hundred or thousand more complex building blocks (*macros*) have to be placed on a limited area, connected with each other, with power supply, and with the "outside world" via *primary inputs and outputs* of the chip. They all have to be synchronized with each other by feeding *clock signals* to each of them, and optimized with respect to timing, power consumption and other metrics such as manufacturing yield which can be significantly affected by how the devices and wires connecting them are arranged in *physical design*.

Figure 1.1 shows an example of a standard cell which computes a four-way NAND function $f(a,b,c,d) := \overline{a \wedge b \wedge c \wedge d}$ for Boolean input variables a, b, c, and d. Although it is well known that the two-way NAND function is universal, i.e. all complex Boolean functions can be expressed using only two-way NAND operators, *standard cell libraries* used in practice comprise also the other common elementary functions such as AND, OR, exclusive OR (evaluating to true iff exactly one input is true), NOR, NOT, etc. There are also a number of other standard cells, including low-complexity composite functions of these elementary Boolean functions, and *latches* for storing the result of a computation performed in one *clock cycle*, allowing to use it as input for computations in the next cycle. Each device comes in different discrete sizes, larger versions offering a higher *driving strength*, and thus the ability to send the output signal across a longer wire to its destination, at the cost of higher power consumption. Altogether, a *standard cell library* in current technologies typically contains a few thousand different of such device templates. A state-of-the-art chip can contain ten million and more instantiations of them. If the design of such a chip was printed at the same scale as figure 1.1, it would cover an area of 700×700 square meters, i.e. roughly the area of 70 soccer fields.

While the designs of the earliest chips in the 1960's were composed by hand, subsequently complemented by simple automated solutions as the number of elements to be arranged grew larger, mathematical methods began to become increasingly important to find "good" solutions. Today, given the huge instance sizes, even the attempt to find a

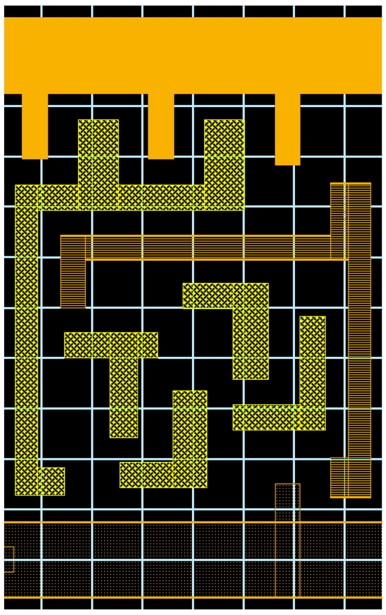


Figure 1.1: A 4-way NAND circuit with four *input pins* a, b, c, and d (small yellow areas) computing the function $\overline{a \wedge b \wedge c \wedge d}$ (identifying variables and pins for simplicity), whose resulting value is output at the large yellow pin. The orange solid and dotted structures at the top and bottom constitute *power supply*, and the lined rectangles are other *blockages*.

feasible solution often is hopeless without mathematics. Also with advanced combinatorial optimization algorithms, the VLSI design problem is too complex to be solved in one single step. It is therefore split up into *design stages:*

- *Logic design* translates the specification of the desired logic function to be performed by a chip into a list of standard cells (and some complex macros) and for each *source pin*, i.e. output pin of a device or primary input of the entire chip, a list of *sink pins*, i.e. input pins of other devices or primary output pins of the chip, to which its signal is to be communicated. This set of sink pins plus the source pin is called a *net* and is a statement that *electrical equivalence* of these pins is required in the final solution. Logic design usually is not considered as part of physical design, but there is no strict borderline as logic design *does* affect physical design, and logic design optimizations can help to reach design closure, see Werber [2007].

- *Placement* arranges the devices on the chip area. This step significantly affects the timing of the chip, as placement imposes lower bounds on the lengths of the wires by which the pins of each net can be connected. Also with respect to routability, feasibility crucially depends on placement. Because placement is done early in the design process, little information on timing and routability is available, so estimations are often used to guide the placement process. See e.g. Brenner, Struzyna and Vygen [2008] for a comprehensive overview and advanced placement algorithms developed and implemented as part of BonnPlace®.

- *Timing optimization* is an important step to optimize the tradeoffs between power and area consumption, and the *cycle time* that can be achieved. There are numerous opportunities for optimization: The size of the standard cells used, and wire widths selected to connect the pins of each net with each other, directly affect timing. Simply put, larger versions of cells and wider wires yield a better timing, but besides the higher power consumption of larger circuits, there is contention on limited space which has to be shared between different nets. Further, *buffering* of long connections is an important step to bridge long distances with acceptable delay. See Held [2008] for a recent and comprehensive work on *timing closure* (i.e. the process to achieve required timing specifications) in VLSI design.

- *Clock tree synthesis* organizes the synchronization between the devices on a chip. As part of timing optimization, desired time windows for required arrival times at each sink pin are computed, and based on these windows and the choices made for device and wire sizes which determine the delay across a connection from source to sink pins, also time windows during which each output pin of a latch (or other storage element) must send its signal. The task of clock tree synthesis is to distribute clock signals to all latches, controlling the time at which they send their output signals and accept signals at their input pins, and to do this such that the time windows prescribed by timing optimization are respected. Additional complexity arises from the need of *variation tolerance*, i.e. as high fidelity to the time windows as possible under variation of signal delays due to uncontrollable imprecisions in the manufacturing process. See Maßberg [2009] for an introduction and algorithms applied in BonnClock®.

- *Routing* is the last major step in the VLSI design process. Although almost all important decisions (placement of devices, choice of device and wire sizes) have been made already in previous design stages, a good routing solution is needed in order to realize the connection of each net with electrical characteristics that do not deviate too much from those prospected during timing optimization. Particularly, an unnecessarily long detour in the wiring of a single out of ten million nets can render the design unusable. Apart from connecting the pins of each net with each other, pins and wires of different nets must be disjoint, and in fact even keep a certain minimum distance from each other, to avoid short-circuits. Also, the way in which nets are routed affects global objectives such as power consumption and manufacturing yield. As the focus of this thesis is routing, we go further into details below.

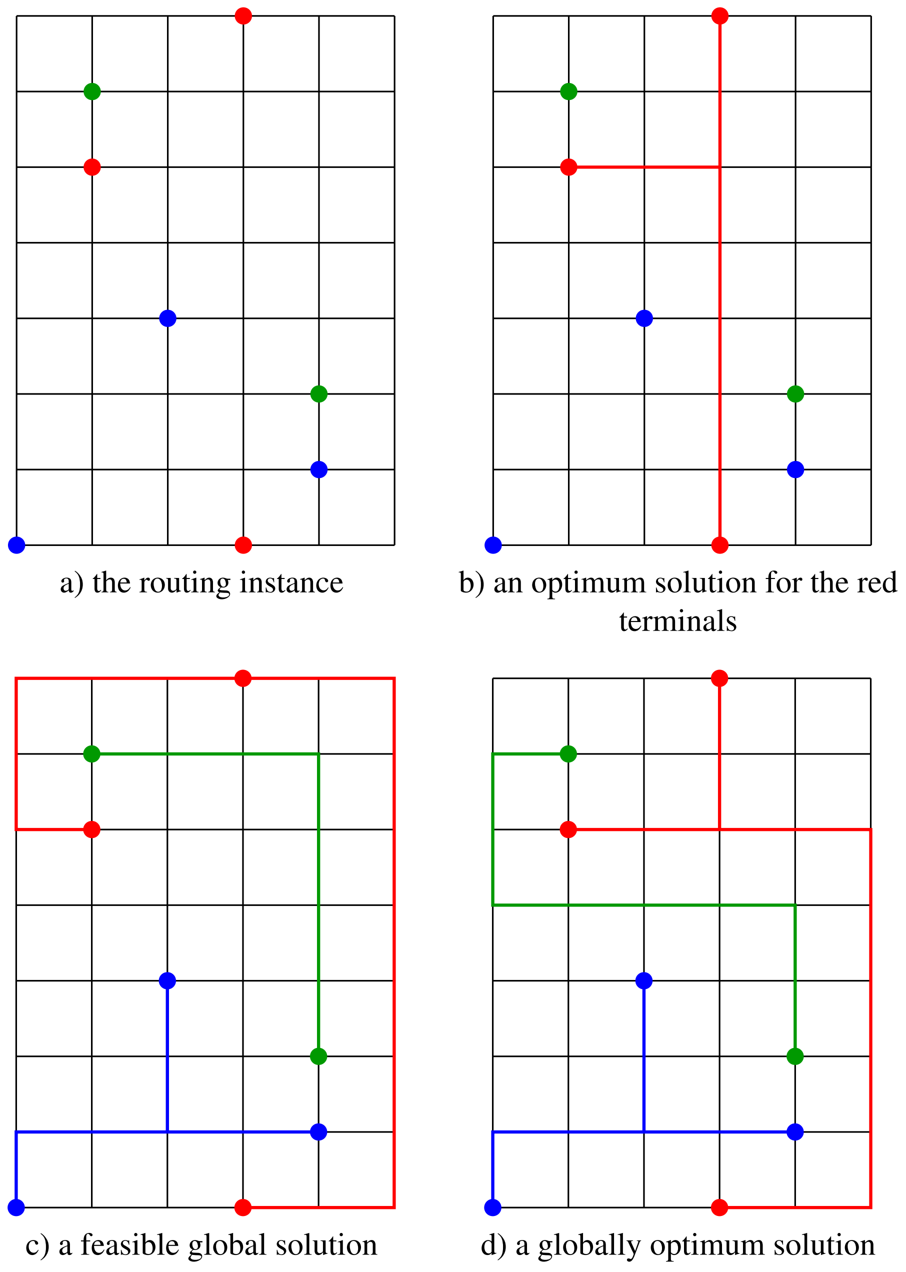
We also refer to Korte, Rautenbach and Vygen [2007] who show how combinatorial optimization algorithms are utilized in chip design by the BonnTools program suite which comprises programs for each of the design steps mentioned above. The book edited by Alpert, Mehta and Sapatnekar [2009] provides a general and up-to-date resource on state-of-the-art algorithms and methods applied to VLSI physical design. The short descriptions of each of the major design stages given above make clear that there are various interdependencies among them. In fact, they often cannot be done sequentially, but have to be executed in a feedback loop. E. g., a placement might turn out to be unroutable in the last step, meaning that it is impossible to find a connection for each net meeting the timing constraints, or finding a connection for each net at all. In such cases designers have to iterate, and much effort is spent in earlier steps on good estimations of achievable results in later steps to avoid this as much as possible. Targeting the interdependency between placement and routing, elaborate *congestion estimation* techniques are employed, see e.g. Shelar and Saxena [2009], Brenner and Rohe [2002], and Menge [2008].

To give an example of a routing problem, let us take one step back and abstract the geometric shapes of pins to points or vertices in a two-dimensional grid graph as shown in figure 1.2 a). Points of the same color (red, green or blue) define the pins of a net that have to be connected with each other using the edges of the grid graph. A connection for one net, such as the one shown in b) for the red net, is called a *Steiner tree*, and in this example it is a shortest one. However, choosing this solution for the red net makes the green and blue nets unroutable because they have pins on either side of the vertical red wire, and Steiner trees for different nets must be disjoint. A feasible solution to the routing problem is thus possible only with a detour in the routing of the red net, as shown in c). In this solution, the green and blue net are both routed shortest possible. This solution however is not globally optimum: Figure 1.2 d) shows a solution with shorter *total* wiring length, which indeed is an optimum solution and also has the advantage of a lower detour in the red net. The example demonstrates that some coordination has to take place between the nets not only to find a globally optimum solution, but even a feasible solution.

Except for very small instances, also the routing problem, even though only a part of the overall VLSI design problem, is too complex to be solved in a single step, for a number of reasons:

1. At the outset, even if extremely simplified as in the example given in figure 1.2, it contains many elementary combinatorial optimization problems such as finding a minimum length Steiner tree or a vertex-disjoint packing of paths, which are NP-hard already in their simplest forms, see e.g. Korte and Vygen [2008].

2. Increasingly intricate *design rules* which enforce or forbid certain geometric metal shape configurations to ensure manufacturability add substantially to the complexity which is inherent already in abstract formulations of the routing problem. This is especially true for those parts of the wiring that access pins because of the irregular geometric structures exhibited by standard cell libraries used in industry. Cho et al. [2009b] and Pan et al. [2008], respectively, provide a comprehensive overview of manufacturability-related constraints.


a) the routing instance

b) an optimum solution for the red terminals

c) a feasible global solution

d) a globally optimum solution

Figure 1.2: Example of a small routing instance: The task is to find three vertex disjoint Steiner trees in a two-dimensional grid graph connecting all red, green and blue vertices, respectively (see a)). If, as in b), the red Steiner tree is chosen shortest possible, a feasible solution cannot be found for the green and blue terminals. Figure c) shows that with a suboptimal red Steiner tree a feasible solution with total length (here: number of edges) 31 is possible. In this solution the green and blue Steiner trees are optimal. A globally optimum solution (shown in d)) however requires a detour also in the green Steiner tree. The total number of edges in this solution is 29.

3. At the same time it becomes more important to satisfy *timing constraints*, as electrical characteristics of wires scale significantly worse than transistors across successive process generations, and thus the relative contribution of wiring delay to overall latch-to-latch delay continuously grows with shrinking feature sizes.

4. Of course also some objective is to be optimized in routing, and the objectives that are of practical interest — manufacturing yield and power consumption — make the VLSI routing problem a non-linear optimization problem.

5. Finally, instance sizes are enormous: On the largest chips today ten million and more Steiner trees have to be packed within eleven wiring layers of about 22×22 mm^2 in size. The *track graph* which we define in chapter 2 as a discretized and thus simplified representation of routing space, and in which the Steiner trees are to be disjointly embedded, comprises more than 100 billion vertices on such designs.

Although VLSI routing is a most challenging problem, it can be tackled impressively well in practice, and mathematics contributes substantially to this. One factor that eases the routing problem is that design rules take effect only on a rather local scale, while timing constraints and the value of the objective to be optimized depend rather on global decisions. This suggests a decoupling into a *global* and a *detailed routing* step: In global routing, only an approximate wiring is generated, defining *corridors* for each net to which search space is restricted in detailed routing. The basic idea is that detailed routing performs a *legalization* of the global routing solution w.r.t. those constraints that have been neglected during global routing, and if doing this step carefully, does not significantly degrade timing or the overall objective that has been (nearly) optimized in global routing.

Remark. Detailed routing can indeed adversely affect timing even if all connections run inside the corridors prescribed by global routing, and all timing constraints have been met in global routing: One example are connections of chains of pins done entirely on one of the lowest wiring layers (saving vias and wiring length because the pins can be used as "bridges" in most cases), instead of connecting all of them to a backbone route running on a higher routing layer. This costs routing space, but can significantly reduce RC delay because all source-sink paths have high-resistive elements (vias and wire segments on lower layers) almost only close to the sink (see e.g. Hu, Robins and Sze [2009] for a survey on timing-driven interconnect synthesis).

In this partitioning of the routing task into global and detailed routing, global routing performs a coordination *among all nets* to find a feasible, and ideally, a globally optimum or close-to-optimum solution. Detailed routing performs only a coordination among nets sharing a corridor, which significantly reduces complexity.

While partitioning into global and detailed routing is very common, there are approaches which add further steps: First, global routing sometimes is done hierarchically on a series of decreasingly coarsened instances of the problem. The purpose of this approach, first proposed by Burstein and Pelavin [1983], is to control runtime: Wider corridors increase detailed routing search space and runtime (while potentially decreasing accuracy relative to the computed global routing solution), but narrower corridors increase global routing runtime. Performing a coarse global routing followed by a second global routing step restricted to corridors obtained from the coarser level can overcome these problems. Theoretically, such a series of routing steps could result in a detailed routing solution in the end, but in practice a — more or less sharp — line can be drawn between coarsening stages in which most design rules can be neglected, and where they become critical.

Second, so-called *track routers* solve the ordering problem of wires within a corridor obtained from global routing, which means that they have to deal with local scale effects induced by design rules, but only a limited subset of these because they do not handle pin access. Although a (very simple) track routing step sometimes is done mainly to save runtime, considering several nets at once in principle offers potential for optimization which a detailed router that processes nets sequentially does not have. As realizing the objective value achieved in global routing, and ensuring that timing constraints are still met after detailed routing legalization, depends on the arrangement of wires within corridors, such a step can help to narrow the gap between global and detailed routing solution quality. Cho et al. [2007], for example, present a track routing approach for optimizing manufacturing yield.

Apart from track routing, there are more methods to work on *groups* of nets in detailed routing: First, there are iterative ripup-and-reroute approaches (see e.g. Salowe [2009] for a survey) which remove already routed wires of one or more nets to free up space for routing another net, and try to reroute the connections which have been "ripped up" in a different way. To do so, in turn other connections might be ripped up. In practice, such heuristical methods work well for resolving most local conflicts. Grötschel, Martin and Weismantel [1996b], on the other hand, present a cutting plane algorithm for packing Steiner trees which might help to resolve very difficult conflicts. This approach, like ripup-and-reroute and track routing, is usable however only on rather small groups of nets.

1.1 BonnRoute®

BonnRoute® is the routing tool in the BonnTools program suite and contains a global routing and a detailed routing module. Earlier versions of BonnRoute® contained a global router based on a linear programming formulation of a Steiner tree packing problem with relaxed integrality constraints, developed by Albrecht [2001a,b] and later extended by Müller [2006a] based on the work of Vygen [2004] to take linearized spacing dependent costs into account. This version has been recently replaced by an implementation of a new convex min-max resource sharing algorithm which can (approximately) find a provably optimum solution with respect to the real non-linear objective functions defined by manufacturing yield and power consumption. Moreover, as running time is an important metric for a router, the new algorithm has been parallelized very efficiently. The new global router was developed in a joint project with IBM and Magma Design Automation, Inc. It is used today at IBM to route most current chip designs, and has become part of Magma's tool suite.

In detailed routing, BonnRoute® takes a sequential approach, i.e. one connection is routed after another. Routing space is represented by an efficient interval data structure that is decribed in chapter 2, and an extremely fast interval-based Dijkstra algorithm due to Hetzel [1998] and subsequently generalized by Peyer, Rautenbach and Vygen [2006] and Humpola [2009] is employed for sequentially routing all connections, using a ripup-and-reroute approach to resolve local conflicts. Also the detailed routing module of BonnRoute® looks back on a success story of hundreds of chip designs routed with it at IBM.

BonnRoute® does not contain a track routing module. Schulte [2006] showed that yield optimization by a post-routing wire spreading approach gives very good results in practice. As the prime objective of any router is *design closure*, i.e. getting all nets routed without violating design rule or timing constraints, targeting a major step in the routing flow towards a single objective such as manufacturing yield (which certainly is secondary relative to design closure) is not a viable approach in practice. Routing nets individually usually offers the advantage that some optimality guarantee on the solution quality for a single net can be given. The interval-based Dijkstra algorithm employed in BonnRoute® provably finds shortest paths and provides a significant advantage over other routers w.r.t. total wiring length and via count.

The detailed routing engine of BonnRoute® was parallelized by Rohe [2001], distributing routing tasks over different client machines and communicating over a network using a message passing protocol. A master process was used to coordinate the work and accept or reject solutions returned by clients depending on if they contained conflicts among each other.

A more efficient parallelization approach, both with respect to memory consumption and parallelization speedup, was implemented by Panten [2005] and later enhanced by Zühlke [2008]. This method is based on area partitioning and the shared-memory paradigm, i.e. the routing tasks are not distributed across different machines. It partitions the chip area into subareas, one for each processor that is used, and lets each processor sequentially route all connections that can be made without leaving the subarea that it has been assigned. Nets with pins in different subareas are not (completely) routed, so BonnRoute® iteratively chooses new partitionings to close connections that could not be done in the earlier partitionings. With a small number of different partitionings, almost all nets can be routed, and a final step using just one processor on the whole chip area is done to route a very small number of connections that could not be done within any of the subareas before. This approach scales reasonably well up to 8 processors on current designs.

1.2 Outline

The main focus of this thesis is on the resource sharing aspects inherent in the VLSI routing problem, and on fast algorithms and data structures that allow instances with millions of nets to be routed in acceptable time.

We begin with efficient data structures for modeling routing space in chapter 2. As geometries of shapes (see figure 1.1) have become more irregular compared to "gridded" technologies a few years ago, efficient representation of routing space, both with respect to memory consumption and query time, becomes more involved. We develop a space-efficient spatial data structure, called *fast-grid*, that stores precomputed and continuously updated information to enable very fast queries on legality of a wire of a certain type on an edge of the *track graph* which we define as a discretized representation of routing space. This data structure is queried by the interval-based path search algorithm used in BonnRoute®, reducing path search runtime by a factor of more than 5. The *routing tracks* defining the track graph need not be placed at uniform distances to each other, but should align well with blockage structures on the chip, such that total usable track length is maximized, making it easier to find feasible solutions to the Steiner tree packing problem restricted to the discrete track graph. We formulate this problem as an abstract interval covering problem and show that it can be solved efficiently.

In chapter 3 we discuss the convex MIN-MAX RESOURCE SHARING PROBLEM, which is a generalization of linear fractional packing problems, and present a simple fully polynomial approximation scheme to (approximately) minimize the maximum relative resource utilization, also called *congestion*. Our algorithm makes use of the *block-angular* structure of many resource sharing algorithms, using *block solver* subroutines to find $(1+\varepsilon_0)$-optimum solutions to subproblems defined by each *block*, for some $\varepsilon_0 \geq 0$, and provably finds an $(1+\varepsilon_0+\varepsilon)$-optimal solution to the overall resource sharing problem, for any value of an *approximation parameter* $\varepsilon > 0$. The algorithm allows to use *weak block solvers* which do not provide an approximation guarantee for arbitrarily small values of ε_0, and achieves the best currently known runtime in this case. Further, we present a shared-memory-parallel variant of our algorithm which achieves very good parallelization speedup because it does not use *mutex locking* for synchronization between concurrent threads. The MIN-MAX RESOURCE SHARING PROBLEM is formulated as an optimization problem over conex sets. In many applications, including global routing, one is interested in *integral solutions*, i.e. solutions from discrete subsets. At the end of chapter 3, we show how to obtain an integral solution from a fractional solution by *randomized rounding* without increasing maximum congestion too much.

The algorithm presented in chapter 3 is applied to the GLOBAL ROUTING PROBLEM in chapter 4. We show how the important objectives in VLSI routing, manufacturing yield and power consumption, can be optimized by formulating the GLOBAL ROUTING PROBLEM as a MIN-MAX RESOURCE SHARING PROBLEM, at the same time obeying timing constraints on *critical paths* of the design. These constraints and objectives depend nonlinearly on wire-to-wire spacing, and BonnRoute® is the first global router to directly optimize over this model without linearization. Moreover, we present a block solver which, based on dynamic programming, can be used to route nets with certain topology restrictions and can be used for *port assignment* in hierarchical design scenarios. Finally, we discuss implementation details and present experimental results, proving the efficiency of our algorithm in practice and showing yield-, power- and wiring length optimization results on state-of-the art industrial chips.

Chapter 2

Modeling Routing Space

In this chapter we introduce a simple model of the routing space and the design rules that the metal shapes implementing a chip — especially wires generated by a routing tool — have to respect. We focus on minimum distance rules that can be viewed as packing constraints. We do not cover rules here that govern shapes of a single net, such as short edge rules, via reliability rules, minimum area rules, or connectivity rules that state conditions under which a net is considered electrically connected. See e.g. Peyer [2007] for a survey of these and other so-called *same net rules*, and the LEF/DEF Language Reference [2007] for a formal standard widely used in industry for specifying design rules to routing tools. We just remark that achieving correctness of a routing solution w.r.t. same net rules, although non-trivial, is almost always possible by rather local modifications to the routing in practice if the underlying Steiner tree packing problem has been solved and some simple heuristics have been applied that help to avoid most same net rule violations.

Because of the complexity and enormous problem sizes faced in VLSI routing, BonnRoute® restricts wires to run on predefined *tracks* located at a certain minimum distance to each other. Only for connecting pins that cannot be accessed in this way short wire segments are used that do not run on a track. Although theoretically this restricts the solution space, it can be justified in practice for current technologies: With almost no routability degradation and considerably reduced complexity, many design rule violations are avoided. See chapter 2.5 for a discussion of key properties of current technologies supporting this approach, and of challenges that might be faced in future technologies if these properties do not continue to hold.

BonnRoute® uses a spatial data structure, called *shape grid*, to query objects like pins, wires or blockages in the proximity of a given location on a chip, and to relate them to the nets they belong to, if any. It is used to find out if a wire can be put somewhere without violating minimum distance rules to already existing objects, or if this is not the case, to determine if existing wires can be removed such that the answer becomes positive. The shape grid consumes a large part of the memory used by BonnRoute®, so memory efficiency plays a key role. We show how the current implementation of the shape grid can be generalized to support irregular track-to-track distances without sacrificing memory efficiency.

For querying legality of wires to be routed on the predefined tracks, we develop a second data structure, called *fast grid*, in section 2.3 that stores precomputed and continuously updated legality information for the most frequent wire types. We show experimental results that demonstrate considerable runtime reductions achieved when employing this data structure in BonnRoute®. Finally, in section 2.4 we propose a method to determine routing tracks such that the limited space available for routing is used efficiently.

2.1 Basic Definitions

Definition 2.1. *The* base coordinate system *used in BonnRoute® is a three-dimensional cartesian coordinate system. The* chip area *is an axis-parallel cuboid*

$$\mathcal{A} := \left\{(x,y,p) \in \mathbb{Z}^3 : x_{\min} \le x \le x_{\max}, y_{\min} \le y \le y_{\max}, 0 \le p \le p_{\max}\right\},$$

in the base coordinate system with $x_{\min}, x_{\max}, y_{\min}, y_{\max} \in \mathbb{Z}$ and $p_{\max} \in 2\mathbb{Z}$ such that $\mathcal{A} \neq \emptyset$. Let $P := \{0, \ldots, p_{\max}\}$ the set of plane numbers. *For each $p \in P$, we refer to the number p and to the set $\mathcal{A}_p := \{(x,y,z) \in \mathcal{A} : z = p\}$ as* plane p. *If p is even, we call $\mathcal{A}_p$ a* wiring plane *and synonymously refer to it as* layer $p/2$, *otherwise we call it a* via plane.

Two wiring planes $p, p' \in P$ are called neighbouring *if $|p - p'| = 2$.*

All objects on the chip, e.g. pins, wires or blockages, will have their borders on integral coordinates in the base coordinate system.

On each layer, with very few exceptions wires run into the same direction, called *preferred direction* of that layer. We restrict ourselves to *Manhattan routing*, meaning that in each layer wires run parallel to the x- or y-axis. This still is common design practice today, although there are design systems that allow wires running in directions that are integer multiples of 45 degrees. Proposed as early as 1968 by Heiss [1968] for printed circuit boards, this "diagonal routing" approach was later considered also for VLSI chips, e.g. by Lodi [1988], Chiang and Sarrafzadeh [1991], Natarajan et al. [1992] and Ho et al. [2005], and termed "X architecture" by Teig [2002]. In order to make full use of the potential for wire length, power and die size reduction offered by this approach, of course the placement has to be adapted, too. Teig et al. [2009] describe a design system, including placement and routing, for the X Architecture. Diagonal routing however must be supported not only by the placement and routing tools, but also by many other tools in the tool chain, also in manufacturing. The practical difficulties entailed in this are one of the main reasons why Manhattan routing is still prevalent today. Analogously to the X Architure, a "Y architecture" (using preferred directions that are integer multiples of 60 degrees) has been proposed (Chen et al. [2003a,b]), but was adopted in practice even less than the X architecture.

We assume that preferred directions alternate between neighbouring layers, which is rather a requirement to than a limitation of our router. The main practical rationale behind this is to avoid crosstalk problems between wires running close to each other in neighbouring layers for a long distance. Cross-talk sensitive wires can be shielded from neighbouring wires in the same layer by metal shapes connected to ground voltage, but this is not possible between different layers. We call a plane *horizontal* if the preferred direction is parallel to the x-axis, and *vertical* otherwise, and assume to have at least two wiring planes. We start with a few convenient definitions:

Definition 2.2. *For any $a := (a_1, a_2, a_3) \in \mathbb{R}^3$, let $x(a) := a_1$, $y(a) := a_2$ and $z(a) := a_3$. For any $b := (b_1, b_2) \in \mathbb{R}^2$, let $x(b) := b_1$ and $y(b) := b_2$.*

For $A \subseteq \mathbb{R}^2$ or $A \subseteq \mathbb{R}^3$, let $x(A) := \{x(a) : a \in A\}$, and $y(A) := \{y(a) : a \in A\}$.

Definition 2.3. *Let $A \subseteq \mathbb{R}^3$ and $B, C \subseteq \mathbb{R}^2$. We define*

i) $A + B := \{(x(a) + x(b), y(a) + y(b), z(a)) : a \in A, b \in B\}$

ii) $-B := \{-b : b \in B\}$

iii) $A - B := A + (-B)$

iv) $C + B := \{b + c : b \in B, c \in C\}$

v) $C - B := C + (-B)$

Definition 2.4. *For any $F \subseteq \mathbb{R}^3$ or $F \subseteq \mathbb{R}^2$, let $|F|_x := \sup\{|x(a) - x(b)| : a, b \in F\}$ and $|F|_y := \sup\{|y(a) - y(b)| : a, b \in F\}$. If F is a closed set, F° denotes the interior of F.*

Definition 2.5. *For any compact $A \subseteq \mathbb{R}^k$, $k \in \mathbb{N}$, let bbox(A) be the minimal axis-parallel cuboid containing A, called* bounding box of A.

Definition 2.6. *A* shape *on plane p, $p \in P$, is a pair (R, c), where $R := [x_1, x_2] \times [y_1, y_2] \times \{p\}$ with $x_1 \leq x_2$, $y_1 \leq y_2$ and $x_1, x_2, y_1, y_2 \in \mathbb{Z}$ is an axis-parallel rectangle intersecting $\mathcal{A}$, and $c \in \mathbb{N}$ is called a* shape class.

For a shape (R, c) and an axis-parallel rectangle R' we define the intersection $(R, c) \cap R' := (R \cap R', c)$, which is again a shape. For convenience, we define area$(S) := R$ for a shape $S = (R, c)$, area$(\mathcal{S}) := \cup_{S \in \mathcal{S}}$area$(S)$ for a set $\mathcal{S}$ of shapes, and area$(A) := A$ for any $A \subseteq \mathbb{R}^3$.

Naturally, we say that a shape, shape set or set of points A intersects or covers another shape, shape set or set of points B if area(A) intersects or covers area(B).

2.1.1 Wire and Via Representation

Definition 2.7. *A* routing shape model *is a pair (R,c), where $R = \emptyset$ or R is an axis-parallel rectangle in $\mathbb{R}^2$ with $(0,0) \in R$ and integral boundary coordinates, and $c \in \mathbb{N}$ is a shape class. If $R = \emptyset$, the routing shape model is called* empty*. For convenience, we define area$(S) := R$ for a routing shape model $S = (R,c)$.*

A wire model *is a routing shape model, and a* via model *is a triple (b,m,t) of routing shape models, where b is called* bottom pad model*, m* via middle model *and t* top pad model.

A wire type *t is a function that maps each even number in P to a wire model and each odd number in P to a via model. A wire type t is called* undefined on plane *p ($p \in P$) if $t(p)$ is or contains, respectively, an empty routing shape model.*

Let $\mathcal{T}$ be the set of all wire types. For a plane $p \in P$, the set $\mathcal{M}_p$ of routing shape models of plane *p is*

- *the set of non-empty wire models $t(p)$, bottom pad models in $t(p+1)$ (if $p < p_{\max}$), and top pad models in $t(p-1)$ (if $p > 0$) for $t \in \mathcal{T}$, if p is a wiring plane*
- *the set of non-empty via middle models in $t(p)$ for $t \in \mathcal{T}$, if p is a via plane*

Electrical connections in a net are encoded by one-dimensional line segments, also called *stick figures*, with an associated *wire type*:

Definition 2.8. *A* wire segment *is a pair (w,t), where w is a (possibly zero-dimensional) axis-parallel line segment contained in a plane $\mathcal{A}_p, p \in P$, and t is a wire type which is not undefined on plane p. A* via *is a pair (v,t), where v is a line segment parallel to the z axis with endpoints in $\mathcal{A}_p$ and $\mathcal{A}_{p+2}$ for some $0 \leq p <= p_{\max} - 2$, and t is a wire type which is not undefined on plane $p+1$.*

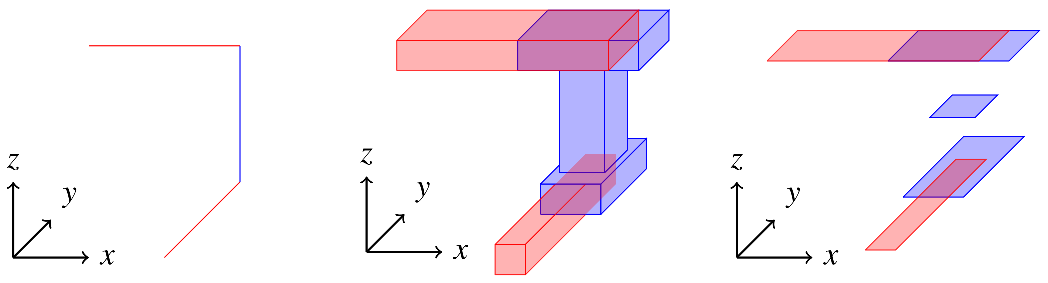
The wire shape *of, or induced by a wire segment (w,t) in plane $p \in P$ is the shape $(w+R,c)$ if $t(p) = (R,c)$.*

The via shapes *of, or induced by a via (v,t) connecting planes p and $p+2$, $0 \leq p <= p_{\max} - 2$, are the shapes*

$$\left\{\left((v \cap \mathcal{A}_p) + R_b, c_b\right), \left((v \cap \mathcal{A}_{p+1}) + R_m, c_m\right), \left((v \cap \mathcal{A}_{p+2}) + R_t, c_t\right)\right\},$$

called bottom, middle *and* top shape*, respectively, if $t(p+1) = ((R_b,c_b),(R_m,c_m),(R_t,c_t))$. See figure 2.1 for an illustration.*

Remark. These definitions require stick figures to run *inside* the wire or via shapes they define. We make use of this property later in Lemma 2.20 and section 2.3.1 for efficiency reasons, but the strict requirement that $(0,0) \in R$ for each non-empty routing shape model (R,c) can actually be relaxed to the requirement that R intersects the x- or y-axis, respectively, depending on the preferred direction of the plane on which a routing shape model (R,c) is used. If neither the x- nor the y-axis is intersected, this only affects efficiency, but not correctness of BonnRoute®.



a) Stick figure representation b) Physical shapes c) Shapes as modeled in BonnRoute®

Figure 2.1: Stick figure representation, physical shapes and shapes modeled as in BonnRoute® for two wire segments (red) and a via (blue)

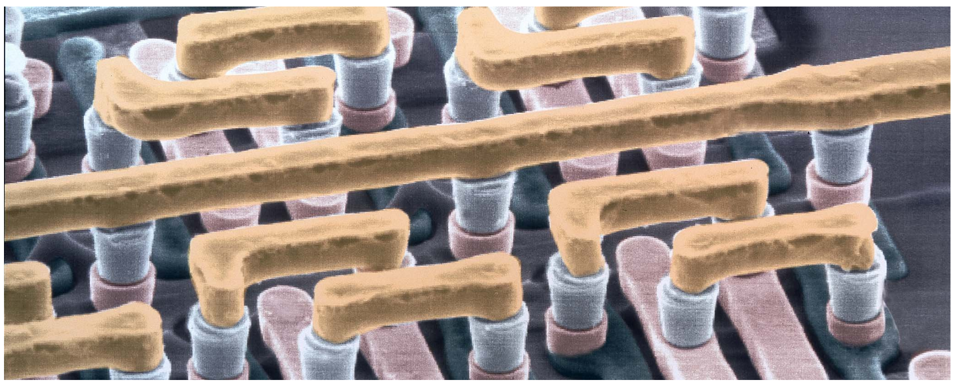


Figure 2.2: Small part of a real chip viewed by an electron microscope (artificially colored).

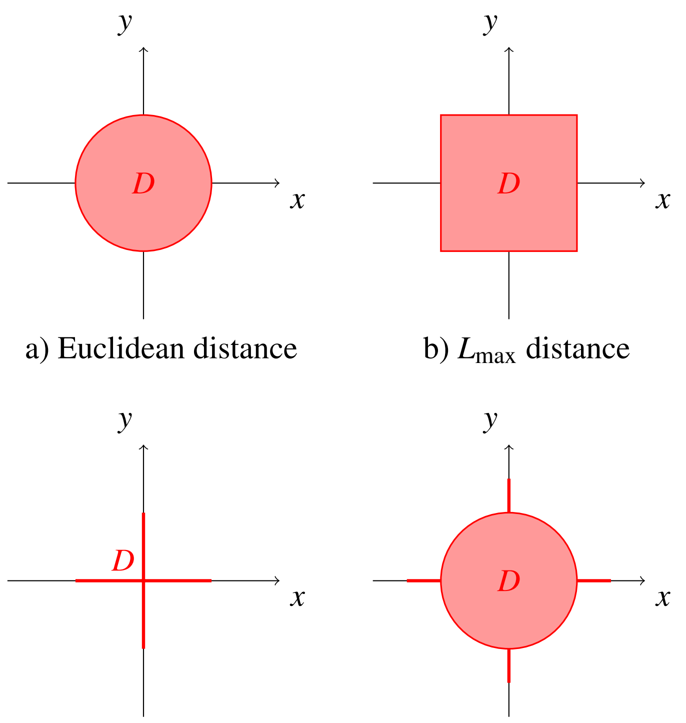
We remark that the three-dimensional shapes shown in figure 2.1 b) are still an idealization of the actually produced shapes. In particular, shapes cannot be manufactured with sharp corners as in the idealized picture. Figure 2.2 shows a photography of a small part of a real chip taken by an electron microscope.

The stick figure representation is convenient because wire and via models are defined such that sufficient electrical conductance among wire and via shapes is guaranteed if the corresponding wire segments and vias are connected. Wire shapes sometimes are abstractions in the sense that they are not fully metallized, but contain electrically disjoint parts like ground shielding. In this case ensuring electrical correctness by the connectivity of the stick figure representation keeps even more complexity out of the router.

2.1.2 Minimum Distance Rules

Definition 2.9. *A* minimum distance rule *is a triple (c_1, c_2, D), where $c_1, c_2 \in \mathbb{N}$ are shape classes, and D is a non-empty closed set in $\mathbb{R}^2$ with $(\lambda_1 x, \lambda_2 y) \in D$ for each $(x,y) \in D$ and $0 \le \lambda_1, \lambda_2 \le 1$.*

Two shapes (R_1, γ_1) and (R_2, γ_2) respect *a minimum distance rule $r := (c_1, c_2, D)$ iff either $(c_1, c_2) \neq (\gamma_1, \gamma_2)$ or $(R_1 + D) \cap R_2^\circ = \emptyset$. Otherwise we say that they* violate *rule r.*



a) Euclidean distance

b) $L_{\max}$ distance

c) hor./ver. distance

d) Typical minimum distance rule for via and wide metal shapes

Figure 2.3: Examples of minimum distance rules

Figure 2.3 shows examples of minimum distance rules. The ordering of the two shape classes in the definition of a minimum distance rule is important for encoding direction-dependent minimum distance requirements. However, one minimum distance rule per unordered pair of shape classes suffices:

Proposition 2.10. *Let $r_1 := (c_1, c_2, D)$ and $r_2 := (c_1, c_2, D')$ be minimum distance rules. Two shapes (R_1, c_1) and (R_2, c_2) respect r_1 and r_2 if and only if they respect the minimum distance rule $(c_1, c_2, D \cup D')$. Further, (R_1, c_1) and (R_2, c_2) respect r_1 if and only if they respect the minimum distance rule $(c_2, c_1, -D)$.* $\square$

We assume that for any pair $\{c_1, c_2\}$ of shape classes that can occur in shapes on plane p $(p \in P)$ there is a minimum distance rule (c_1, c_2, D) that prohibits intersection of the two shapes (which is not applied, however, if both shapes belong to the same net). By proposition 2.10 we can assume w.l.o.g. that there is exactly one such rule, but for convenience we assume that for each rule (c_1, c_2, D) we also have the rule $(c_2, c_1, -D)$.

Remark. Minimum distance rules can be considerably more complex than introduced in definition 2.9. Notably, the minimum distance between two shapes might not only depend on direction (which is covered by definition 2.9), but also on the presence of other objects of a certain type within the proximity of one or both shapes, see e.g. Cho et al. [2009b]. However, practically all of these complex rules so far can be successfully handled in BonnRoute® by a skillful combination of

1) allowing some rarely occuring types of errors which are corrected by a post-processing step, and

2) introducing (small) safety margins in the minimum distance rules.

This is possible because combinations of shapes that violate only complex rules, but none of the simple minimum distance rules as introduced in this section do not occur frequently. If this assumption should turn out to be unsustainable in future technologies, the fast grid data structure presented in section 2.3 offers the opportunity to handle more complex rules in BonnRoute® without too much additional runtime.

In the following, let $p \in P$. Let $\mathcal{B}_p$ be a set of shapes intersecting $\mathcal{A}_p$, called *blockages*. Blockages are given as input to the router, and the router cannot modify them. Besides structures for power supply, abstractions of complex circuits (called *macros*) used as building blocks of the chip, and areas reserved by designers for various reasons, the blockages also comprise the set $\mathcal{B}_p^{\text{pin}} \subseteq \mathcal{B}_p$ of all pin shapes.

At any time, $\mathcal{W}_p$ denotes the set of all wire and via shapes on plane p, i.e. shapes induced on plane p by all wire segments and vias, given as input or generated by the router. For convenience we assume that each point outside $\mathcal{A}$ is covered by a blockage shape, so all wire and via shapes are forced to be contained in $\mathcal{A}$. Each wire, via or pin shape belongs to exactly one net. As stated at the outset, we do not consider rules between shapes of the same net. Further, since blockages cannot be modified by the router, we check minimum distance rules between two shapes only if at least one of them is induced by some via or wire segment.

Because minimum distance requirements depend on metal width, rules are extended to sets of shapes as follows: Let $\{(w_1^p, c_1^p), \dots, (w_k^p, c_k^p)\} \subseteq \mathbb{N} \times \mathbb{N}$, $k \in \mathbb{N}$, $0 < w_1^p < \dots < w_k^p$, and $w_k^p \geq \max\{x_{\max} - x_{\min}, y_{\max} - y_{\min}\}$. Let $\mathcal{Q}$ be the set of shapes (S, c_i) with S being a maximal axis-parallel square covered by $\mathcal{W}_p \cup \mathcal{B}_p^{\text{pin}}$, and i the smallest number such that w_i is greater than or equal to the width of S. Analogously, let $\mathcal{Q}'$ be the set of shapes $(S', c_{i'})$ with S' being a maximal axis-parallel square covered by $\mathcal{B}_p$, and i' the smallest number such that $w_{i'}$ is greater than or equal to the width of S'. For simplicity, we assume here that all shapes are fully metallized.

Definition 2.11. *Let $\mathcal{R}$ be a set of minimum distance rules. Then with the above notation we say that $\mathcal{W}_p$ is* legal w.r.t. $(\mathcal{B}_p,\mathcal{R})$ *iff each $s_1 \in \mathcal{W}_p \cup \mathcal{Q}$ and $s_2 \in \mathcal{W}_p \cup \mathcal{Q} \cup \mathcal{B}_p \cup \mathcal{Q}'$ that are not intersected by shapes belonging to the same net respect all rules in $\mathcal{R}$.*

We can safely assume that $\mathcal{R}$ and $\mathcal{B}_p$ are given such that if $s_1 \in \mathcal{W}_p \cup \mathcal{Q}$ and $s_2 \in \mathcal{Q}' \setminus \mathcal{B}_p$ violate a rule $r \in \mathcal{R}$, there is also an $s_2' \in \mathcal{B}_p$ such that s_1, s_2' violate a rule $r' \in \mathcal{R}$, for otherwise one can efficiently construct $\mathcal{B}_p'$ and $\mathcal{R}'$ with this property and

$$\mathcal{W}_p \,\text{legal w.r.t.}\, (\mathcal{B}_p,\mathcal{R}) \Leftrightarrow \mathcal{W}_p \,\text{legal w.r.t.}\, (\mathcal{B}_p',\mathcal{R}').$$

This means that it suffices to consider single blockage shapes in $\mathcal{B}_p \setminus \mathcal{B}_p^{\text{pin}}$ when checking minimum distance rules. However, groups of pin shapes and/or shapes induced by vias or wire segments might be involved in a rule violation while this is not the case for any individual shape among them. In practice, this is mainly a concern in pin access, and can be avoided in most cases. In the following we therefore restrict ourselves to checking minimum distance rules only for pairs of shapes and, assuming a fixed set $\mathcal{R}$ of rules given as input, define legality of wire or via shapes accordingly:

Definition 2.12. *A wire or via shape s_1 on plane p ($p \in P$) is called* legal *(w.r.t. the already existing shapes $(\mathcal{W}_p,\mathcal{B}_p)$) if there exists no $s_2 \in \mathcal{W}_p \cup \mathcal{B}_p$ not belonging to the same net such that any minimum distance rule $r \in \mathcal{R}$ is violated by s_1 and s_2.*

2.1.3 Routing Tracks and Track Graph

To limit the number of routing tracks in a wiring plane, we will require them to run at a certain minimum distance from each other which we define as the *minimum pitch* of this plane:

Definition 2.13. *Let $p \in P$ be a wiring plane and $\mathcal{T}_p \subseteq \mathcal{T}$ those wire types $t \in \mathcal{T}$ with $t(p)$ non-empty. If p is horizontal, we define*

$$\Delta_p(t_1,t_2) := \lceil \max\{y : (x,y) \in R_1 - R_2 + D\} \rceil,$$

for $(t_1,t_2) \in \mathcal{T}_p \times \mathcal{T}_p$, where $t_1(p) = (R_1,c_1)$, $t_2(p) = (R_2,c_2)$ and $(c_1,c_2,D) \in \mathcal{R}$ is the minimum distance rule to be respected between these wire models. We define $\Delta_p(t_1,t_2)$ analogously if p is vertical.

The minimum pitch of plane *p is*

$$\Delta_p^{\min} := \min_{t \in \mathcal{T}_p} \Delta_p(t,t),$$

i.e. the minimum distance orthogonal to the preferred direction at which any two wire segments with the same wire type can run without violating minimum distance rules.

Remark. Note that wire segments with different wire types can be closer to each other than the minimum pitch.

Definition 2.14. *Let $p \in P$. We define*

$$w_p^x := \min\{|area(m)|_x : m \in \mathcal{M}_p \cup \mathcal{B}_p\}$$

as the minimum extension of any possible shape on plane p in x-direction, and

$$d_p^x := \min\left\{ \max_{(x,y)\in D} x : (R_1, c_1), (R_2, c_2) \in \mathcal{M}_p \cup \mathcal{B}_p, (c_1, c_2, D) \in \mathcal{R} \right\}$$

as the minimum distance in x-direction required between any two possible shapes on plane p. w_p^y and d_p^y are defined analogously.

Remark. In practice $\Delta_p^{\min} = w_p^y + d_p^y$ on horizontal and $\Delta_p^{\min} = w_p^x + d_p^x$ on vertical wiring planes $p \in P$, respectively. The wire type $t \in \mathcal{T}_p$ attaining the minimum pitch of plane p in practice is used for almost all wires in current technologies; see also the discussion in chapter 2.5.

Now we define a *track coordinate system* on each plane. Let $p \in P$. First assume that p is a horizontal wiring plane. Let $T_p = \{t_1, \dots, t_{|T_p|}\}$ be the coordinates of horizontal *tracks* on which wires shall run, with $y_{\min} \le t_1 < \dots < t_{|T_p|} \le y_{\max}$. For convenience we assume two artificial tracks $t_0 < y_{\min}$ and $t_{|T_p|+1} > y_{\max}$ that are not used for routing. We require $\Delta_p^{\min} \le t_i - t_{i-1} \le c\Delta_p^{\min}$ for $1 \le i \le |T_p| + 1$ and some constant $c \ge 1$. In most cases, neighbouring tracks will have distance $\Delta_p^{\min}$ to each other, but sometimes it may be desirable to align tracks with blockage structures extending for long distance in preferred direction, like power rails or large macros. Track coordinates are either specified as input to the router, or determined during initialization, e.g. as described in section 2.4.

For convenience we define $T_{p'} := \emptyset$ for $p' \notin P$. As p is a wiring plane and has at least one neighbouring wiring plane, we can assume that $Q_p := T_{p+2} \cup T_{p-2}$ is non-empty and call this set the set of *points of interest on plane p*. Let $Q_p = \{q_1, \dots, q_{|Q_p|}\}$ with $q_1 < \dots < q_{|Q_p|}$. We call pairs of indices (i, j) with $1 \le i \le |T_p|$ and $1 \le j \le |Q_p|$ *track coordinates* on plane p. We associate track coordinates with the base coordinates of the corresponding tracks by the mapping $b(i, j, p) := (q_j, t_i, p)$ for all $1 \le i \le |T_p|$ and $1 \le j \le |Q_p|$. We call base coordinates $(x', y', p) \in \mathcal{A}_p$ *on-grid* if they are contained in the image of this mapping. Trivially, $b(\cdot)$ can be computed in constant time. This is also true for the reverse mapping:

Lemma 2.15. *Let $(x, y, p) \in \mathcal{A}$ be a point on a wiring plane. The closest track to (x, y, p) and the closest point of interest on this track can be found in constant time and with $O(|T_p| + |Q_p|)$ bytes of precomputed information.*

Proof. W.l.o.g. assume that p is a horizontal plane. Let $k \in \mathbb{N}$ be the largest number such that $2^k \le \Delta_p^{\min}$. Compute $y' := \left\lfloor \frac{y}{2^k} \right\rfloor$ and look up the smallest $i \in \mathbb{N}$ with $t_i \ge 2^k y'$ in an array indexed by y'. Then t_i or $t_{\min\{i,|T_p|\}+1}$ is the next higher track from (x, y, p), and t_i or $t_{\max\{i-1,0\}}$ is the next lower track from (x,y,p). Analogously the next higher or lower point of interest on this track can be found. □

Note that the lookup arrays needed for this are relatively small, and divisions by powers of two are very fast.

If p is a vertical wiring plane, track coordinates are defined analogously, and the mapping to base coordinates is $b(i,j,p) := (t_i, q_j, p)$ for all $1 \le i \le |T_p|$ and $1 \le j \le |Q_p|$.

If p is a via plane, we define track coordinates as pairs of indices (i,j) with $1 \le i \le |T_{p-1}|$ and $1 \le j \le |T_{p+1}|$, defining intersections of tracks in the neighbouring wiring planes projected to plane p. Analogous to above, $b(\cdot)$ maps track coordinates on via planes to these projected intersection points.

Definition 2.16. *We call a via or a wire segment* on-track *if both endpoints are on-grid, and* off-track *otherwise.*

Note that this definition includes on-track wire segments orthogonal to preferred direction, called *jogs*, that of course are not contained in a track.

As mentioned at the beginning of this chapter, BonnRoute® uses on-track vias and wire segments for all connections of a net except for accessing pins which often requires off-track wiring locally around a pin. The core routing engine of BonnRoute® is a generalization of Dijkstra's algorithm (Dijkstra [1959]) that can make use of the structure of the given graph and edge costs to process whole sets of vertices at once. The graph used for on-track routing is a subgraph of the *track graph* which is defined as follows:

Definition 2.17. *Given a set of routing tracks on each wiring plane, the* track graph *is the graph $G = (V,E)$, where V is the set of all on-grid points on wiring planes, and E the set of edges $\{v_1,v_2\} \subseteq V \times V$ such that $(bbox(\{v_1,v_2\}),t)$ is a via or a minimal on-track wire segment with length greater than zero for some wire type $t \in \mathcal{T}$.*

The on-track path search used in BonnRoute® goes back to the work of Hetzel [1995, 1998] who proposed a shortest-paths algorithm for regular track graphs (i.e. restricted to uniform track-to-track distances over all layers) that labels *intervals* of vertices on the same track with distance functions. It was extended by Peyer, Rautenbach and Vygen [2006] from intervals to more general vertex sets. Humpola [2009] improved the runtime bound of Hetzel's algorithm and generalized it to irregular track graphs, allowing arbitrary distances between tracks, even within the same layer. This interval based shortest paths algorithm motivates the interval structure of the data structures presented in sections 2.2 and 2.3.

2.2 The Shape Grid

The *shape grid* is a spatial data structure that efficiently stores all relevant data about blockage, wire and via shapes. The information in the shape grid allows to decide if a wire (of any given wire type) can be placed somewhere without violating minimum distance rules. If not, it allows to find out if there is a set of nets which can be ripped up such that the answer becomes positive, and to explicitly determine such a set of minimum size.

Each plane $\mathcal{A}_p$, $p \in P$, is tiled by a grid of so-called *cells*. In the following, let $p \in P$ w.l.o.g. be a plane with horizontal preferred direction. Let $x_{\min} = x_0 < x_1 < \ldots < x_{n_x} = x_{\max}$ and $y_{\min} = y_0 < y_1 < \ldots < y_{n_y} = y_{\max}$, $n_x, n_y \in \mathbb{N}$, be integral coordinates of cell boundaries.

For $i, j \in \mathbb{Z}$, we define cell (i, j, p) by the area

$$C(i,j,p) := [x_{i-1}, x_i] \times [y_{j-1}, y_j] \times \{p\}$$

if $1 \leq i \leq n_x$ and $1 \leq j \leq n_y$, and $C(i,j,p) := \emptyset$ otherwise. We call the non-empty intersections of shapes in $\mathcal{B}_p \cup \mathcal{W}_p$ with $C(i,j,p)$ *cell shapes* of cell (i,j,p). We define the *cell configuration* of cell (i,j,p) as the set of its cell shapes, each translated by $-(a_x, a_y, 0)$, where $(a_x, a_y, p) \in C(i,j,p)$ is a well-defined anchor point within the cell area. The shapes in a cell configuration are called *cell configuration shapes*. If cell boundary coordinates and anchor points are chosen well, the number of different cell configurations is not very high in practice (see below for further discussion). Each of them is assigned a *cell configuration number* $k \in \mathbb{N}$.

Basically, the shape grid stores a set $\mathcal{I}_p$ of intervals $\{(i_1, j, p), \ldots, (i_2, j, p)\}$, $1 \leq i_1 \leq i_2 \leq n_x$ and $1 \leq j \leq n_y$, of cells in preferred direction in plane p having the same cell configuration number k. The intervals stored in this way are disjoint, and cells not contained in any of them have a default configuration number denoting an empty cell configuration. For an interval $I \in \mathcal{I}_p$, we write $k(I)$ to denote the cell configuration number of the cells contained in I. Storing intervals in a balanced search tree (e.g. an AVL tree, see Adel'son-Vel'skiĭ and Landis [1962]) allows to find the interval containing a cell in time logarithmic in the total number of intervals.

Remark. BonnRoute® uses one search tree per cell row (or cell column on vertical planes), which does not change the theoretical bound on interval lookup time, but makes a considerable difference in practice. Further, this saves space because it suffices to store only the three numbers i_1, i_2 and $k(I)$ for an interval $I = \{(i_1, j, p), \ldots, (i_2, j, p)\}$, i.e. not storing the cell row index (or cell column index on vertical planes, respectively).

Additionally a table is stored that maps each cell configuration number to the actual cell configuration. In order to decide if adding or removing a shape to or from a cell configuration results in an already existing or a new configuration, another balanced search tree is maintained.

The presented data structure can be updated efficiently if wire or via shapes are added to or removed from $\mathcal{W}_p$, and allows for efficient queries on legality of a given new wire or via shape that is not yet contained in $\mathcal{W}_p$. This relies on two important conditions that are fulfilled on all chips in practice:

1) Most wire segments run in preferred direction, and wire segments running orthogonal to preferred direction are very short (often only 1–2 tracks).

2) Wire (and via) shapes span only a small constant number of cells orthogonal to preferred direction. This is even true for the widest wires that occur, and most wire shapes completely fit into a row of cells.

We align cell rows with tracks by choosing cell boundaries in the middle of neighbouring tracks (see section 2.2.2 for details). In this way most wires split intervals only in one cell row, which is important to keep memory consumption low. The current version of BonnRoute® uses uniform cell sizes within each plane. This enables an alignment of cells and tracks only if distances between neighbouring tracks within a plane are regular. With the non-uniform cell-sizes that we propose in section 2.2.2 in order to align cells and tracks also if track-to-track distances are irregular, special attention has to be paid to storing net identifiers memory efficiently. We propose a method for this in the next section.

2.2.1 Storing Net Identifiers

If a shape is not legal, we often want to determine if there is a set of wire or via shapes of other nets that can be ripped up in order to make it legal and to find such a set, ideally of minimum cost w.r.t. some objective function. In order to support this task, for each removable cell shape the information on which net it belongs to must be available. To store this information, we introduce a function $\sigma_{i,j,p}$ for each cell (i,j,p) that maps a subset of its cell configuration shapes to a set of net identifiers. These functions are not stored explicitly for each cell. Instead, they are stored at intervals of cells, and intervals are chosen maximal such that mapping functions and cell configurations are equal for all cells in an interval. For simplicity we assume in the following that all non-blockage shapes are removable.

We now propose a method for storing these mapping functions memory efficiently. Let $1 \leq i \leq n_x$ and $1 \leq j \leq n_y$. For any shape F that intersects $C(i,j,p)$, let $\gamma_{i,j,p}(F)$ the cell configuration shape induced by F in the cell configuration of cell (i,j,p), that is $F \cap C(i,j,p)$ translated by $-(a_x,a_y,0)$, where (a_x,a_y,p) is the anchor point of cell (i,j,p). Now let $S \in \mathcal{W}_p$ be a removable shape intersecting $C(i,j,p)$. We store a net identifier for S in cell (i,j,p), i.e. add $\gamma_{i,j,p}(S)$ to the domain $\mathrm{dom}(\sigma_{i,j,p})$ of $\sigma_{i,j,p}$, if and only if the conditions

$$|\gamma_{i,j,p}(S)|_x = |C(i,j,p)|_x \text{ or } (|\gamma_{i,j,p}(S)|_x, i') \geq_{\mathrm{lx}} (\min\{|C(i,j,p)|_x, |\gamma_{i',j',p}(S)|_x\}, i) \quad (2.1)$$

and

$$|\gamma_{i,j,p}(S)|_y = |C(i,j,p)|_y \text{ or } (|\gamma_{i,j,p}(S)|_y, j') \geq_{\mathrm{lx}} (\min\{|C(i,j,p)|_y, |\gamma_{i',j',p}(S)|_y\}, j) \quad (2.2)$$

are satisfied for each $i' \in \{i-1,i,i+1\}$ and $j' \in \{j-1,j,j+1\}$, where $\geq_{\mathrm{lx}}$ denotes lexicographical ordering, i.e. $(a,b) \geq_{\mathrm{lx}} (a',b')$ iff $a > a'$ or ($a = a'$ and $b \geq b'$). For example, if $|\gamma_{i-1,j,p}(S)|_x = |\gamma_{i,j,p}(S)|_x < |C(i,j,p)|_x$, we do not add $\gamma_{i,j,p}(S)$ to $\mathrm{dom}(\sigma_{i,j,p})$. We make use of this in the proof of Lemma 2.20 below.

It is easy to see that if (2.1) and (2.2) are not met, the corresponding net identifier is in the image of the mapping function of one of the eight neighbouring cells in x-, y- or diagonal direction. Because we require minimum distance rules to prohibit intersection of any two shapes not both belonging to blockage or to the same net, two intersecting cell shapes $A \subseteq C(i,j,p)$ and $B \subseteq C(i',j',p)$ ($i' \in \{i-1,i+1\}$ and $j' \in \{j-1,j+1\}$) must both belong either to blockage or to the same net. Assuming that the functions that map cell configuration shapes to net identifiers can be evaluated in constant time (which will be the case with our choice of cell boundaries), we therefore have the following fact:

Proposition 2.18. *For a removable cell shape, the identifier of the net to which it belongs can be found in constant time.*

We refer to the image $\operatorname{img}(\sigma_{i,j,p})$ of $\sigma_{i,j,p}$ as the set of *net identifiers stored for cell* (i,j,p). Cell sizes will be chosen such that, given the minimum distance rules $\mathcal{R}$, with only very few exceptions at most one net identifier has to be stored for a cell. $\sigma_{i,j,p}$ is then encoded by a number $n \in \mathbb{Z}$ and a lookup table as follows: if n is non-negative, this means that at most one net identifier is stored for this cell. In this case each cell configuration shape of cell (i,j,p) that is in $\operatorname{dom}(\sigma_{i,j,p})$ is mapped to this identifier, which is directly encoded by n. If n is negative, cell configuration shapes belonging to different nets are in the domain of $\sigma_{i,j,p}$. In this case, n is used as negated index into a table which explicitly stores the mapping function for this cell. The size of this table is very small in practice.

If cell sizes are uniform in preferred direction, mapping function domains can be stored in the cell configurations without increasing the interval count, as the following Lemma shows.

Lemma 2.19. *Let $x_i = x_{i-1} + c$ for $1 \leq i \leq n_x$ and some $c \in \mathbb{N}$. Then if cell configurations are extended to store the domain of a cell's mapping function additionally to cell configuration shapes, the number of different cell configurations and the number of intervals do not change.*

Proof. Let $S \in \mathcal{W}_p$ intersecting $C(i_1,j,p)$ and $C(i_2,j,p)$, $1 \leq i_1,i_2 \leq n_x$, $1 \leq j \leq n_y$ and $i_1 \neq i_2$. Then, $\gamma_{i_1,j,p}(S) = \gamma_{i_2,j,p}(S)$ implies that either $\gamma_{i',j,p}(S) \in \operatorname{dom}(\sigma_{i',j,p})$ for both $i' \in \{i_1,i_2\}$ or none of them. □

Remark. Choosing uniform cell sizes in preferred direction is advisable also to avoid unnecessary changes of cell configuration numbers along a wire running in preferred direction, as this would mean an increased number of intervals that have to be stored.

If we store mapping function domains in the cell configurations, e.g. by marking those cell configuration shapes that belong to the domain, we do not have to inspect neighbouring cells to decide if the net identifier for a removable cell shape can be determined by the cell's mapping function. Neighbouring cells have to be inspected only if the answer is negative.

2.2.2 Choosing Cell Boundaries

With the above considerations, we suggest to choose cell boundaries as follows:

i) Choose an offset value $m \in \mathbb{Z}$ with $0 < m \leq \min\{x_{\max} - x_{\min}, d_p^x + w_p^x\}$ and set $n_x := \lceil (x_{\max} - x_{\min} - m)/(d_p^x + w_p^x) \rceil + 1$ and $x_i := m + (i-1)(d_p^x + w_p^x)$ for $1 \leq i < n_x$.

ii) $n_y := |T_p|$ and $y_j := \lfloor (t_j + t_{j+1})/2 \rfloor$ for $1 \leq j < n_y$.

We then define $(\min x(C(i,j,p)), t_j, p)$ as anchor point of cell (i,j) for $1 \leq i \leq n_x$ and $1 \leq j \leq n_y$. With these choices, only for cells that contain cell shapes not induced by on-track vias or wire segments it may become necessary to explicitly store mapping functions in the lookup table:

Lemma 2.20. *If there are no minimum distance rule violations and cell (i,j,p) ($1 \leq i \leq n_x$ and $1 \leq j \leq n_y$) contains only cell shapes induced by on-track vias or wire segments, we have $|img(\sigma_{i,j,p})| \leq 1$.*

Proof. Assume there are two cell configuration shapes S_1, $S_2 \in \mathrm{dom}(\sigma_{i,j,p})$ with $\sigma_{i,j,p}(S_1) \neq \sigma_{i,j,p}(S_2)$. As p is a horizontal plane without loss of generality, and by the definition of cell boundaries, exactly one horizontal track runs through each cell. First assume that both S_1 and S_2 intersect this track, and w.l.o.g. $\max x(S_1) < \min x(S_2)$. Then $|S_1|_x > w_p^x/2$ and $|S_2|_x \geq w_p^x/2$ by (2.1). Hence, by the choice of cell boundaries, S_1 and S_2 must have distance less than d_p^x to each other in x-direction, which is a contradiction as the definition of d_p^x implies that S_1 and S_2 violate a minimum distance rule.

Now assume that one of the shapes, w.l.o.g. S_1, does not intersect the track. S_1 is induced by an on-track via or wire segment, so the corresponding via or wire shape S must intersect the track running through a neighbouring cell (i,j',p) with $j' \in \{j-1, j+1\}$ because $(0,0) \in R$ for each routing shape model $(R,c) \in \mathbb{R}^2 \times \mathbb{N}$ by definition 2.7. By the choice of horizontal cell boundaries, $j' = j+1$ implies $|S_1|_y < |\gamma_{i,j',p}(S)|_y$, and $j' = j-1$ implies $|S_1|_y \leq |\gamma_{i,j',p}(S)|_y$, in both cases contradicting (2.2). $\square$

Remark. For the proof of Lemma 2.20, it actually suffices that R intersects the x-axis for each routing shape model $(R,c) \in \mathcal{M}_p$ (as plane p was assumed to have horizontal preferred direction). Analogously, R has to intersect the y-axis if plane p has vertical preferred direction.

Further, endpoints of vias and wire segments in most cases split intervals only in one row of cells, so the number of intervals and thus memory consumption is kept low. This is owed to the fact that most wire segments on plane p have a wire type $t \in \mathcal{T}_p$ with $\Delta_p(t,t) = \Delta_p^{\min}$ and stick figures centered within shapes. Besides that, also the expected number of cell configurations is less than if cell rows are not aligned with tracks.

Choosing m can considerably influence the number of intervals, see figure 2.4 for an example. We suggest choosing m such that the number of intersections of cell boundaries with the shapes induced on plane p by all possible on-track vias of some “standard”

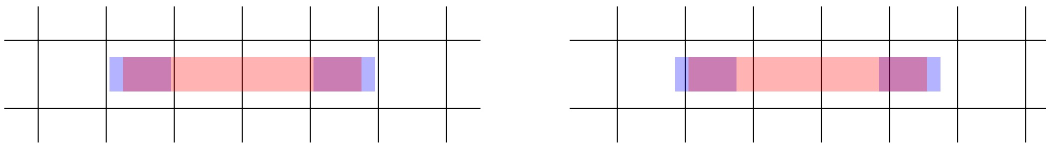


Figure 2.4: A wire shape and two via shapes represented by a different number of intervals depending on the choice of m

wire type is minimized. It is not hard to see that this can be done in $O(|Q_p|\log|Q_p|)$ or $O(d_p^x + w_p^x + |Q_p|)$ time.

Remark. With a copper atomic radius of around 10^{-10}m and one coordinate unit in x- or y-direction in the base coordinate system corresponding to a step length of at least this value, $d_p^x + w_p^x$ is dominated by the number of tracks on today's largest chip routing instances. Therefore the second (pseudopolynomial) bound above is actually better in practice.

Cell boundaries and anchor points are defined analogously for vertical wiring planes. Although all vias of course run in z direction, we define the "preferred direction" of a via plane to be equal to the preferred direction of the next lower plane. This definition is just used for grouping neighbouring cells in preferred direction to intervals. Let $p \in P$ be a via plane with horizontal preferred direction, $x_0, \dots, x_{n_x}$ the coordinates of vertical cell boundaries on plane $p+1$ and $y_0, \dots, y_{n_y}$ the coordinates of horizontal cell boundaries on plane $p-1$, $n_x, n_y \in \mathbb{N}$. Then it is natural to use these boundary coordinates for defining cell boundaries on plane p, and intersections of tracks in the neighbouring wiring planes projected to plane p as anchor points of cells.

We finally remark that if a design uses a standard cell library that is not designed in a "gridded fashion" (meaning that all pins can be accessed by on-track vias or wire segments if cells are placed on a certain grid and routing tracks are suitably defined), many off-track wire or via shapes can be necessary in the lowest wiring and via plane, respectively, in order to access pins. This is the standard case today because industry-standard cell libraries are not designed with routing tracks in mind. In such cases it might be better not to align cells with tracks on the lowest plane, but to choose cell sizes small enough such that the number of net identifiers to be stored for an interval cannot exceed one.

2.3 The Fast Grid

In this section we present a data structure, called *fast grid*, that, using precomputed and continuously updated data, can determine very fast if a given on-track via or wire segment is legal or not.

Let $p \in P$ w.l.o.g. be a horizontal wiring plane, and $\bar{\mathcal{M}} \subseteq \mathcal{M}_p$ be a subset of the routing shape models that can occur in plane p. The fast grid data structure will store information for each $m \in \bar{\mathcal{M}}$. We discuss how to choose $\bar{\mathcal{M}}$ at the end of this section. Let $\mathcal{M}_p^w := \{t(p) : t \in \mathcal{T}_p\}$ be the wire models that can be used on plane p. We write $\bar{\mathcal{M}} = \{m_1, \dots, m_{|\bar{\mathcal{M}}|}\}$ with $\bar{\mathcal{M}} \cap \mathcal{M}_p^w = \{m_1, \dots, m_{|\bar{\mathcal{M}} \cap \mathcal{M}_p^w|}\}$.

As above, let $T_p = \{t_1, \dots, t_{|T_p|}\}$ with $t_1 < \dots < t_{|T_p|}$, and $Q_p = \{q_1, \dots, q_{|Q_p|}\}$ with $q_1 < \dots < q_{|Q_p|}$. For $m = (R, c) \in \bar{\mathcal{M}}$, $1 \le i \le |T_p|$ and $1 \le j \le |Q_p|$ let

$$S(i,j,m) \quad := \quad \big(\{b(i,j,p)\} + R, c\big),$$

be shapes induced by potential on-track vias or zero-dimensional wire segments in plane p, i.e. not contained in $\mathcal{W}_p$ yet, and

$$\begin{aligned} S_x(i,j,m) &:= \big(\big[b(i,j,p), b(i, \min\{j+1, |Q_p|\}, p)\big] + R, c\big), \\ S_y(i,j,m) &:= \big(\big[b(i,j,p), b(\min\{i+1, |T_p|\}, j, p)\big] + R, c\big) \end{aligned}$$

for $m \in \mathcal{M}_p^w$ be shapes induced by potential minimal one-dimensional wire segments in plane p. Further, let

$$\beta(S) := \begin{cases} 0 : S \text{ is legal w.r.t. } (\mathcal{B}_p, \mathcal{W}_p) \\ 1 : \text{otherwise} \end{cases}$$

for any wire or via shape S. Figure 2.5 shows, in a small example, the stick figures that induce

- the illegal shapes $S_x(i,j,m)$ (i.e. with β-value 1) for a wire model m and
- the illegal shapes $S(i,j,m')$ for a via model m'.

The fast grid stores a set $\mathcal{F}_p$ of intervals $\{(i,j_1), \dots, (i,j_2)\}$ for plane p that form a partition of the track coordinates on plane p, i.e. $\dot\bigcup_{I \in \mathcal{F}_p} = \{1, \dots, |T_p|\} \times \{1, \dots, |Q_p|\}$. For each $1 \le i \le |T_p|$ and $1 \le j \le |Q_p|$ we define

$$\beta(i,j) := \sum_{a=1}^{|\bar{\mathcal{M}} \cap \mathcal{M}_p^w|} 2^a \beta(S_x(i,j,m_a)) + \sum_{a=|\bar{\mathcal{M}} \cap \mathcal{M}_p^w|+1}^{|\bar{\mathcal{M}}|} 2^a \beta(S(i,j,m_a)),$$

and intervals are chosen maximal such that for each $I \in \mathcal{F}_p$ and $(i,j), (i,j') \in I$ we have $\beta(i,j) = \beta(i,j')$. We define $\beta(I) := \beta(i,j)$ $((i,j) \in I)$ for $I \in \mathcal{F}_p$ and store $\beta(I)$ together with I.

With this information, legality of a shape induced in plane p by an on-track via or wire segment in preferred direction (w,t) can be checked in

$$O(\log |\mathcal{F}_p| + |\{I \in \mathcal{F}_p : b(I) \cap w \neq \emptyset\}|)$$

time if $\mathcal{F}_p$ is stored as a balanced search tree and neighbouring intervals on the same track are linked with each other, where

$$b(X) := \text{bbox}(\{b(i,j,p) : (i,j) \in X\})$$

for any $X \subseteq \{1, \dots, |T_p|\} \times \{1, \dots, |Q_p|\}$.

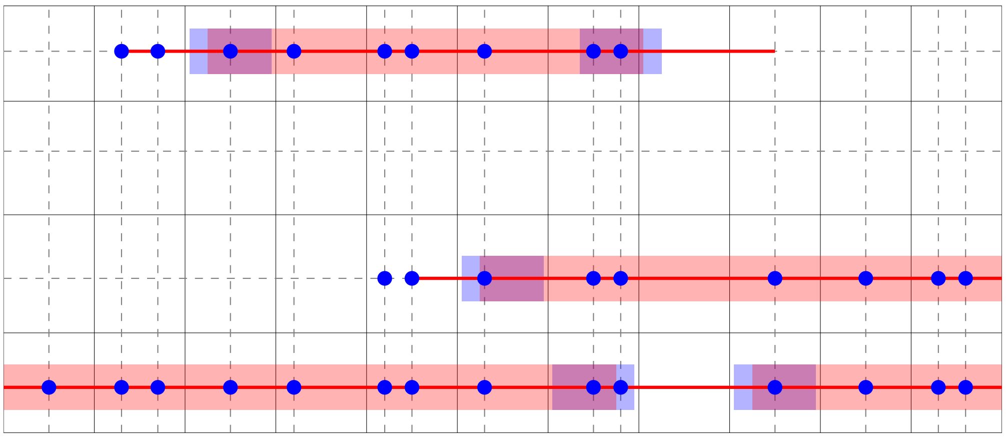


Figure 2.5: For some existing wire and via shapes, this figure shows the wire segments (red lines) and vias (blue dots) that induce the illegal shapes $S_x(i,j,m)$ for some wire model m and $S(i,j,m')$ for some via model m', respectively. Only 4 intervals are needed to represent them, while the shape grid stores 10 intervals. Dashed gray lines denote preferred tracks, black lines are cell borders.

2.3.1 On-Track Jogs

Of course we also need information on legality of on-track jogs. Explicitly storing the values $\beta(S_y(i,j,m))$ can considerably increase the number of intervals, even if stored only in intervals on the lower or higher track connected by the jog. Figure 2.6 shows an example. However, the equivalence

$$\beta(S_y(i,j,m)) = 0 \iff \beta(S(i,j,m)) = \beta(S(i+1,j,m)) = 0 \tag{2.3}$$

will hold for most $1 \le i < |T_p|$, $1 \le j \le |Q_p|$ and $m \in \bar{\mathcal{M}}$. E. g., if there were only shapes induced by on-track vias and wire segments on plane p, (2.3) would hold for all jogs in this plane, which follows from the definition of routing shape models and minimum distance rules.

Remark. The requirement that $(0,0) \in R$ for each $(R,c) \in \mathcal{M}_p$ can actually be relaxed to the requirement that R intersects the x-axis if p has horizontal preferred direction, or y-axis in the other case.

With blockages or shapes induced by off-track vias or wire segments on plane p, (2.3) remains true if neighbouring tracks are closer than $2d_p^y + w_p^y$ to each other and for each minimum distance rule $(c_1,c_2,D) \in \mathcal{R}$, D is an axis-parallel rectangle. In these cases it suffices to redefine

$$\beta(i,j) := \sum_{a=1}^{|\bar{\mathcal{M}}\cap\mathcal{M}_p^w|} 2^a \beta(S_x(i,j,m_a)) + \sum_{a=1}^{|\bar{\mathcal{M}}|} 2^{|\bar{\mathcal{M}}\cap\mathcal{M}_p^w|+a} \beta(S(i,j,m_a)), \tag{2.4}$$

for each $1 \le i \le |T_p|$ and $1 \le j \le |Q_p|$ to answer queries on legality of jogs.

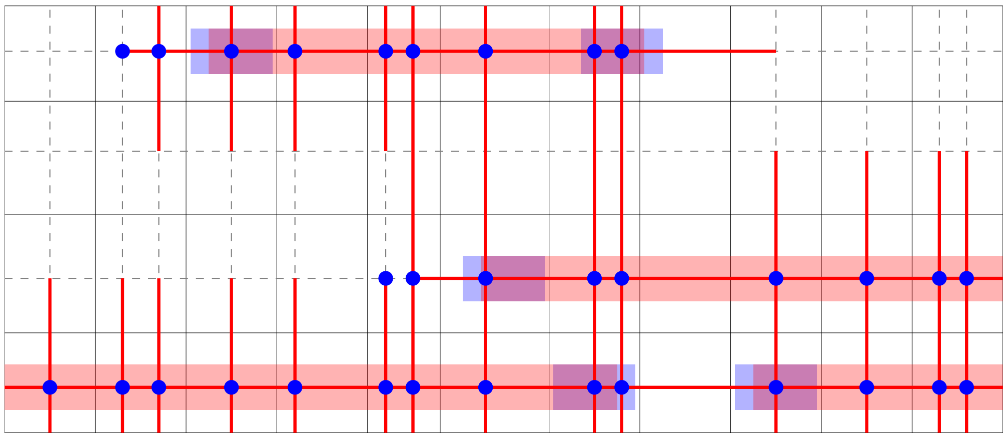


Figure 2.6: Wire segments inducing the sets $S_x(i,j,m)$ and $S_y(i,j,m)$ for a wire model m (red lines), and vias inducing the sets $S(i,j,m')$ for a via model m' (blue dots). Explicitly storing legality of jogs between neighbouring tracks can increase interval count.

Remark. In practice the number of intervals increases only marginally by defining β-values as in (2.4) since

$$\beta(S_x(i,j-1,m)) = 0 \,\text{or}\, \beta(S_x(i,j,m)) = 0 \Longrightarrow \beta(S(i,j,m)) = 0,$$

and only rarely $\beta(S_x(i,j-1,m)) = \beta(S_x(i,j,m)) = 1$ and $\beta(S(i,j,m)) = 0$.

In the general case, i.e. distances between tracks not bounded by $2d_p^y + w_p^y$ and arbitrary minimum distance rules, we modify $\beta(i,j)$ $(1 \leq i \leq |T_p|$ and $1 \leq j \leq |Q_p|)$ as defined in (2.4) by adding 1 iff (2.3) does not hold for i, j and some $m \in \bar{\mathcal{M}} \cap \mathcal{M}_p^w$. So if $\beta(i,j)$ is an odd number, we answer a query on $\beta(S_y(i,j,m))$ for any $m \in \bar{\mathcal{M}} \cap \mathcal{M}_p^w$ by returning 1 if $|\bar{\mathcal{M}} \cap \mathcal{M}_p^w| = 1$ or otherwise querying the shape grid, i.e. querying all cell shapes in a sufficiently large area around $S_y(i,j,m)$, and checking all minimum distance rules that apply.

In practice, there are extremely few situations in which (2.3) is violated. The lowest two planes that are used for off-track pin access are an exception to this, however these planes are not used by the on-track path search at all, so the fast grid data structure does not have to be built on these planes.

Remark. In BonnRoute®, for each bit encoded in $\beta(i,j)$ $(1 \leq i \leq |T_p|$ and $1 \leq j \leq |Q_p|)$, a second bit is stored that encodes if the value of the original bit could change by removing shapes from $\mathcal{W}_p$, i.e. ripping up wires of some already routed connections. In ripup path searches, this saves querying the shape grid for net identifiers at locations where a desired shape is prohibited by a non-removable shape.

2.3.2 Updating the Fast Grid

When a shape $W = (R, c)$ is added to $\mathcal{W}_p$, the fast grid must be updated. If any shape $S = (R', c') \in \{S(i, j, m) : m \in \bar{\mathcal{M}}\} \cup \{S_x(i, j, m) : m \in \bar{\mathcal{M}} \cap \mathcal{M}_p^w\}$ $(1 \le i \le |T_p|$ and $1 \le j \le |Q_p|)$ intersects $R + D$ for the (unique) minimum distance rule (c, c', D), $\beta(S)$ is set to 1. If $\beta(i, j)$ is even, $R + D$ intersects $S_y(i, j, m)$, but not $(S(i, j, m)$ or $S(i+1, j, m))$, and $\beta(S(i, j, m)) = \beta(S(i+1, j, m)) = 0$, $\beta(i, j)$ is incremented by 1. Since β-values are stored at intervals and all track coordinates in an interval are supposed to have the same values, these operations may require intervals to be splitted. On the other hand, two neighbouring intervals I, I' in the same track can be joined if $\beta(I) \neq \beta(I')$ before adding W to $\mathcal{W}_p$, but $\beta(i, j) = \beta(i, j')$ for all $(i, j) \in I \cup I'$ afterwards.

If a shape $W = (R, c)$ is removed from $\mathcal{W}_p$, shapes in $\mathcal{B}_p \cup \mathcal{W}_p$ in the proximity of W must be considered in order to update the fast grid data structure. Let A be the set of track coordinates $(i, j) \in \{1, \dots, |T_p|\} \times \{1, \dots, |Q_p|\}$ for which $\beta(i, j)$ may change. Then the values $\beta(i, j)$ for each $(i, j) \in A$ can be computed from all cell shapes intersecting

$$b(A + \{(0,0), (1,0), (0,1)\}) + \left(\bigcup_{\substack{(R', c') \in \bar{\mathcal{M}} \\ (c', c'', D') \in \mathcal{R}}} R' + D' \right).$$

We do not have to iterate over the cell shapes in each cell separately, but can deal with entire intervals in the shape grid at once:

Proposition 2.21. *Let $p \in P$ be a plane with horizontal preferred direction. If cells in the shape grid have size at most $2w_p^x$ in $x-$direction and the anchor points $a_{c_1}, \dots, a_{c_2}$ of cells in an interval $I := \{(c_1, r, p), \dots, (c_2, r, p)\} \in \mathcal{I}_p$ are equal relative to the lower left cell corner, cell shapes in $\cup_{c_1 \le c \le c_2} C(c, r, p)$ can be replaced by the shapes*

$$\bigcup_{S \in \mathcal{S}_{k(I)}} (S + [a_{c_1}, a_{c_2}]) \tag{2.5}$$

in the computation of β-values without affecting the result. Here $\mathcal{S}_{k(I)}$ denotes the set of cell configuration shapes of cell configuration $k(I)$. □

An analogous result of course holds for planes with vertical preferred direction. Usually the shapes in (2.5) are covered by the corresponding original shapes that induced the cell shapes, but not always, as the example in figure 2.7 shows. Proposition 2.21 shows that nonetheless, due to our choice of cell boundaries and anchor points, we do not have to iterate over single cells if $d_p^x \le w_p^x$. In practice, we have $d_p^x = w_p^x$ in most cases, but if $d_p^x > w_p^x$ holds, cell boundaries can be defined such that cell sizes in x-direction are at most $2w_p^x$.

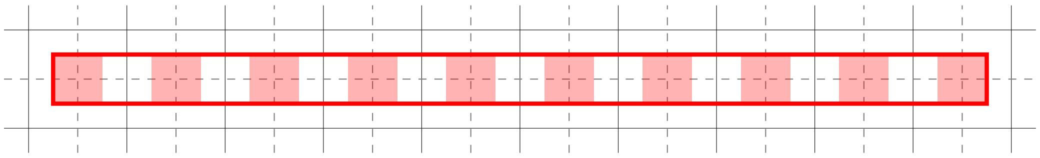


Figure 2.7: A shape in (2.5), drawn with a thick red line, which is not covered by the original shapes (small squares) that induced this interval of identical cell configurations.

2.3.3 Choosing Wire and Via Models for the Fast Grid

We finally discuss the choice of $\bar{\mathcal{M}}$. There are two aspects to consider: the more frequently on-track legality queries for a certain wire or via model are made, the more time can be saved by including this model in $\bar{\mathcal{M}}$ and querying the fast grid data structure instead of the shape grid. On the other hand, some routing shape models require a relatively large update radius when shapes are added to or removed from $\mathcal{W}_p$ and the information stored in the fast grid needs to be updated, leading to a considerable overhead. In practice, the most frequently used routing shape models require a relatively low updating overhead.

In BonnRoute® a routing shape model (R,c) on plane p is included in $\bar{\mathcal{M}}$ only if it occurs in the wire type used for routing a certain minimum fraction of all nets, weighted by an estimate of the net length as e.g. half of the perimeter of the bounding box of the net's pin shapes. The model is not included in $\bar{\mathcal{M}}$ if (still assuming w.l.o.g. that p is a horizontal plane)

$$\max_{(c,c',D)\in\mathcal{R}} |R+D|_y > a\Delta_p^{\min},$$

where a is some constant and $\mathcal{R}$ is the set of minimum distance rules that apply on plane p. Currently for each interval $I \in \mathcal{F}_p$ in the fast grid data structure BonnRoute® uses 32 bits to store $\beta(I)$, so $\bar{\mathcal{M}}$ is limited to the most frequent routing shape models if necessary.

2.3.4 Results

We now present results on some state-of-the-art industrial chips that demonstrate the benefits of the fast grid data structure. Table 2.1 shows the design characteristics. In table 2.2 we present sequential runtimes of the on-track path search engine of BonnRoute® with and without using the fast grid data structure, and the runtime overhead needed to update it when wires are added or removed. Including the overhead for updating, we obtain a speedup factor of roughly 5.5× on average.

Planes 0 and 1, i.e. the lowest wiring and via plane, respectively, are used almost only for pin access, which is done by a special gridless routing subroutine for connecting a pin to a close on-grid point on plane 2, limiting the influence of complex pin geometries and design rules to a small scope. The on-track path search engine of BonnRoute® then connects these *access points* on planes ≥ 2. Because the pin access subroutine uses its own representation of routing space, the fast grid data structure thus is initialized only for planes ≥ 2, too.

Chip	**Technology**	**Die Size**	**# Nets**
Jacques	65nm	0.44 mm × 1.33 mm	47,031
Brigitte	65nm	1.16 mm × 1.21 mm	120,248
Inaya	65nm	6.01 mm × 6.01 mm	170,582
Timo	65nm	1.45 mm × 2.01 mm	276,552
Jo	65nm	3.07 mm × 1.65 mm	515,044
Renate	65nm	3.25 mm × 2.95 mm	529,004
Dorothea	65nm	6.07 mm × 6.07 mm	679,820
Georg	65nm	2.67 mm × 4.10 mm	783,685
Tomoko	65nm	14.81 mm × 15.61 mm	5,340,088
Andre	65nm	15.61 mm × 14.81 mm	7,039,094

Table 2.1: The testbed used for evaluating the fast grid data structure.

Chip	**Runtime (SG)**	**Runtime (FG)**	**FG updating time**	**Speedup**
Jacques	0:17:55	0:02:59	0:01:10	4.31×
Brigitte	1:38:49	0:16:16	0:02:32	5.25×
Inaya	1:51:37	0:21:06	0:09:41	3.62×
Timo	1:25:28	0:13:29	0:04:16	4.81×
Jo	5:16:39	0:50:33	0:12:41	5.00×
Renate	4:28:39	0:35:08	0:09:40	5.99×
Dorothea	17:11:53	3:08:33	0:17:21	5.01×
Georg	6:31:13	1:10:18	0:20:16	4.31×
Tomoko	70:51:56	10:19:55	2:38:42	5.46×
Andre	100:35:53	14:39:56	3:04:47	5.66×
Sum	**210:17:44**	**31:39:11**	**7:01:25**	**5.43×**

Table 2.2: Sequential runtimes (in hours) of the on-track path search routine of BonnRoute® using the shape grid (SG) only, and using the fast grid (FG) for frequent wire types. The third column shows the time needed for initializing and updating the fast grid data structure when wires are added or removed. The last column is the quotient of the first and the sum of the second and third column.

Table 2.3 shows that the peak memory consumption increases by less than 4 percent on average and less than 9 percent on all instances if the fast grid data structure is built. Finally, table 2.4 compares the number of intervals stored in the shape grid and in the fast grid data structure.

Chip	**Memory (SG only)**	**Memory (with FG)**
Jacques	0.53 GB	0.55 GB (+5.1600 %)
Brigitte	1.19 GB	1.28 GB (+7.8700 %)
Inaya	8.16 GB	8.55 GB (+4.7400 %)
Timo	2.22 GB	2.31 GB (+3.8700 %)
Jo	4.19 GB	4.49 GB (+7.3200 %)
Renate	4.97 GB	5.25 GB (+5.5500 %)
Dorothea	9.92 GB	10.42 GB (+5.0800 %)
Georg	6.68 GB	7.23 GB (+8.2800 %)
Tomoko	57.57 GB	59.51 GB (+3.3700 %)
Andre	72.67 GB	73.96 GB (+1.7600 %)
Sum	**168.43 GB**	**173.85 GB (+3.2200 %)**

Table 2.3: Peak memory consumption of BonnRoute® with and without the fast grid data structure.

Chip	**# Intervals (SG)**	**# Intervals (FG)**
Jacques	10,860,283	4,611,859
Brigitte	28,822,333	11,359,473
Inaya	284,315,488	97,587,743
Timo	48,745,463	21,213,095
Jo	104,643,825	45,027,055
Renate	132,022,423	53,414,281
Dorothea	282,675,573	105,239,391
Georg	181,697,799	73,906,055
Tomoko	1,855,635,639	702,059,680
Andre	2,334,605,100	873,469,276
Sum	**5,266,896,418**	**1,989,158,451**

Table 2.4: Number of intervals stored in the shape grid (SG) and in the fast grid (FG) data structures at the end of the routing.

2.4 Determining Routing Tracks

We now discuss how to determine routing tracks, given a set of blockages, pins to be accessed and already existing wiring that should align with the tracks. Roughly speaking, the goal is to maximize a weighted sum of

- the total track length usable for wiring,
- the number of on-track pin access stubs (defined later) and
- the total length of on-track wire segments in the input.

We will go into details in section 2.4.2. Certainly, neighbouring tracks should be usable at the same time, so a certain minimum distance between tracks is required. As we shall see, the problem can be formulated as an interval covering problem which we define and efficiently solve in the next section.

2.4.1 The Maximum Weighted Stable Interval Covering Problem

MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM

Instance: A set $\mathcal{I}$ of intervals in $\mathbb{R}$, weights $w : \mathcal{I} \to \mathbb{R}_+ \setminus \{0\}$ and a number $d \in \mathbb{R}_+ \setminus \{0\}$.

Task: Find numbers $X \subseteq [\min_{I\in\mathcal{I}} l(I), \max_{I\in\mathcal{I}} u(I)]$ with $|x_1 - x_2| \geq d$ for any $x_1, x_2 \in X$ with $x_1 \neq x_2$, maximizing the sum

$$\tilde{w}(X) := \sum_{x\in X} \sum_{\substack{I\in\mathcal{I} \\ x\in I}} w(I).$$

For an interval $I = [a,b]$, let $l(I) := a$ and $u(I) := b$ its boundary coordinates, $I^\circ := I \setminus \{l(I), u(I)\}$ the interior of I and $|I| := b - a$ the length of I. We define

$$U := d \left\lceil \frac{\max_{I\in\mathcal{I}} u(I)}{d} \right\rceil.$$

In the following w.l.o.g. we assume $U > 0$ and $\min_{I\in\mathcal{I}} l(I) = 0$. We will show the following result in this section:

Theorem 2.22. *The* MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM *can be solved in $O((U/d + |\mathcal{I}|)\log|\mathcal{I}|)$ time.*

This result implies a complexity polynomial in $|\mathcal{I}|$, or in $|\mathcal{I}| + |X|$ if the elements of the output set have to be explicitly listed:

Corollary 2.23. *Given an instance $(\mathcal{I},w,d)$ of the* MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM, *the value of an optimum solution and a set $R \subseteq [0,U] \times \mathbb{N}$ can be found in $O(|\mathcal{I}|\log|\mathcal{I}|)$ time such that*

$$X := \bigcup_{(x,r)\in R} \{x+(i-1)d : 1 \le i \le r\}$$

is an optimum solution to $(\mathcal{I},w,d)$.

Of course constructing X from R takes $O(|X|)$ time, assuming $|R| = O(|X|)$. We need the following definition:

Definition 2.24. *Let $\mathcal{J}$ be a set of pairwise disjoint intervals in $\mathbb{R}$, and $\overline{\mathcal{J}} := \mathbb{R} \setminus \cup_{J\in\mathcal{J}} J^\circ$. We define $c_{\mathcal{J}} : \overline{\mathcal{J}} \to \mathbb{R}$ by*

$$c_{\mathcal{J}}(x) := x - \sum_{J\in\mathcal{J}:u(J)\le x} |J|$$

for $x \in \overline{\mathcal{J}}$, and denote the reverse mapping by $c_{\mathcal{J}}^{-1}$.

For an interval I with $l(I),u(I) \in \overline{\mathcal{J}}$, we write $c_{\mathcal{J}}(I) := [c_{\mathcal{J}}(l(I)),c_{\mathcal{J}}(u(I))]$, and $c_{\mathcal{J}}^{-1}(I') := I$ if $I' = c_{\mathcal{J}}(I)$. Finally, for a set $\mathcal{I}$ of intervals I with $l(I),u(I) \in \overline{\mathcal{J}}$, we write $c_{\mathcal{J}}(\mathcal{I}) := \{c_{\mathcal{J}}(I) : I \in \mathcal{I}\}$. We say that $c_{\mathcal{J}}(\mathcal{I})$ arises from $\mathcal{I}$ by cutting $\mathcal{J}$ from $\mathcal{I}$.

Proof of Corollary 2.23. W.l.o.g. we can assume $d = 1$. For an instance $(\mathcal{I},w,1)$ of the MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM, let

$$\mathcal{J} := \Big\{ \big[l(J)+2+|J|-\lfloor|J|\rfloor\,,\; u(J)\big] : J \subseteq [0,U] \text{ maximal s.t. for each } I \in \mathcal{I} \text{ holds } J \subseteq I \text{ or } J^\circ \cap I = \emptyset\,, |J| \ge 3 \Big\}.$$

Then $|\mathcal{J}| = O(|\mathcal{I}|)$, and we can construct $\mathcal{J}$ and sort the elements of $\mathcal{J}$ and $\mathcal{I}$ w.r.t. non-decreasing left boundaries in $O(|\mathcal{I}|\log|\mathcal{I}|)$ time. Observe that $|J| \in \mathbb{N}$ for each $J \in \mathcal{J}$.

We construct $\mathcal{I}' := c_{\mathcal{J}}(\mathcal{I})$ in $O(|\mathcal{I}|)$ time and solve the instance $(\mathcal{I}',w',1)$, where $w'(I') := w(c_{\mathcal{J}}^{-1}(I'))$ for each $I' \in \mathcal{I}'$. If $X \subseteq [0,U]$ is an optimum solution to $(\mathcal{I},w,1)$, then there is a solution X' to $(\mathcal{I}',w',1)$ with value

$$\tilde{w}(X) - \sum_{J\in\mathcal{J}} |J|\tilde{w}(\{l(J)\}),$$

as there is no $J \in \mathcal{J}$ with $l(J)$ on a boundary of any $I \in \mathcal{I}$, and hence $\tilde{w}(\{l(J)\}) = \tilde{w}(\{x\})$ for each $J \in \mathcal{J}$ and $x \in J \setminus \{u(J)\}$. Let now X' be an optimum solution to $(\mathcal{I}',w',1)$, $a(J) := [c_{\mathcal{J}}(l(J))-2, c_{\mathcal{J}}(l(J))]$ for each $J \in \mathcal{J}$, and observe that by construction $X' \cap$

$a(J) \neq \emptyset$ for each $J \in \mathcal{J}$ contained in some $I \in \mathcal{I}$ because all weights are strictly positive. Let

$$R := \left\{ \left(c_{\mathcal{J}}^{-1}(x), 1\right) : x \in X' \right\} \cup \left(\bigcup_{J \in \mathcal{J} : \exists I \in \mathcal{I} : J \subseteq I} \left\{ \left(c_{\mathcal{J}}^{-1}\left(\max\{x \in X' \cap a(J)\}\right), |J|+1\right) \right\} \right).$$

Then

$$X := \bigcup_{(x,r) \in R} \{x+i-1 : 1 \leq i \leq r\}$$

is a solution to $(\mathcal{I}, w, 1)$ with value

$$\tilde{w}(X) = \sum_{x \in X'} \sum_{\substack{I \in \mathcal{I}' \\ x \in I}} w'(I) + \sum_{J \in \mathcal{J}} |J| \tilde{w}(\{l(J)\}),$$

hence X is optimal. We have $|R| = O(|X'|)$, and it is easy to verify that

$$|X'| - 1 \leq \max_{I \in \mathcal{I}'} u(I) = U - \sum_{J \in \mathcal{J}} |J| = O(|\mathcal{I}|),$$

so the claim follows from Theorem 2.22. □

Before we prove Theorem 2.22, we observe that if interval boundaries and d are integral numbers, the MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM can be solved by dynamic programming in $O(U + |\mathcal{I}|)$ time:

Lemma 2.25. *If interval boundaries and the value d in an instance $(\mathcal{I}, w, d)$ of the* MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM *are integral numbers, the* INTEGRAL MAXIMUM WEIGHTED STABLE INTERVAL COVERING ALGORITHM *shown below computes an optimum solution to $(\mathcal{I}, w, d)$ in $O(U + |\mathcal{I}|)$ time.*

Proof. The runtime claim is trivial. To see correctness, let w_{tot}^i be the value of w_{tot} at the end of the i-th iteration in step 2 and observe that $w_{\text{tot}}^i = \sum_{I \in \mathcal{I} : i \in I} w(I)$ for $0 \leq i \leq U$ and $s(j) = s(j-d) + w_{\text{tot}}^j$ for any value of j in step 3. □

By the same arguments as in the proof of Corollary 2.23, we can assume w.l.o.g. that $U = O(d|\mathcal{I}|)$. This still gives only a pseudo-polynomial runtime bound for the INTEGRAL MAXIMUM WEIGHTED STABLE INTERVAL COVERING ALGORITHM, but the number of different values of $s(\cdot)$ over $\{0, \ldots, U\}$ can then be bounded by $O(|\mathcal{I}|^2)$:

Lemma 2.26. *Let $(\mathcal{I}, w, d)$ an instance of the* MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM *with $U = O(d|\mathcal{I}|)$, and $s : [0, U] \to \mathbb{R}_+$ the function that for any $y \in [0, U]$ gives the optimum value of a solution X to $(\mathcal{I}, w, d)$ under the restriction that $X \subseteq [0, y]$. Then the number of different function values of $s(\cdot)$ over $[0, U]$ is bounded by $O(|\mathcal{I}|^2)$.*

```
INTEGRAL MAX. WEIGHTED STABLE INTERVAL COVERING ALGORITHM
    Input  : Intervals $\mathcal{I}$ with boundaries $l : \mathcal{I} \to \mathbb{N}$ and $u : \mathcal{I} \to \mathbb{N}$, and weights
             $w : \mathcal{I} \to \mathbb{R}_+ \setminus \{0\}$. A number $d \in \mathbb{N}$.
    Output: Integral numbers $X \subseteq [0,U]$ maximizing $\tilde{w}(X)$, with $|x_1 - x_2| \geq d$ for
             any $x_1, x_2 \in X$ with $x_1 \neq x_2$.

1   Set $w_{\text{tot}} := 0$ and $\delta_i := 0$ for each $0 \leq i \leq U+1$.
    foreach $I = [a,b] \in \mathcal{I}$ do
        Set $\delta_a := \delta_a + w(I)$.
        Set $\delta_{b+1} := \delta_{b+1} - w(I)$.
2   for $i = 0$ to $U$ do
        Set $w_{\text{tot}} := w_{\text{tot}} + \delta_i$.
        if $i < d$ then
            Set $s(i) := w$.
        else
            Set $s(i) := \max\{s(i-1), s(i-d) + w_{\text{tot}}\}$.
3   Set $X := \emptyset$ and $i := U$.
    while $i \geq 0$ do
        Set $j := \min\{0 \leq j' \leq i : s(j') = s(i)\}$.
        Set $X := X \cup \{j\}$.
        Set $i := j - d$.
```

Proof. A simple induction over $i \in \{1, \dots, U/d\}$ shows that if $s(y) > s(y - \varepsilon)$ for some $(i-1)d < y \leq id$ and all $\varepsilon > 0$, there must be an interval $I \in |\mathcal{I}|$ with $l(I) \in \{y - jd : 0 \leq j < i\}$. Hence $s(\cdot)$ can increase at most at $O(|\mathcal{I}|^2)$ points in $[0,U]$. □

This bound is sharp: Let $d \in \mathbb{N}$ and $\mathcal{I} := \{[2a, 2a+1] : 0 < a < d/2\} \cup \{[d, d^2]\}$ with weights $w([2a, 2a+1]) := a$ for $0 < a < d/2$ and $w([d, d^2]) := d$. Then the number of different values of $s(\cdot)$ over $\{0, \dots, U\}$ is in $\Theta(|\mathcal{I}|^2)$. We shall see that nonetheless a runtime of $O(|\mathcal{I}| \log |\mathcal{I}|)$ can be achieved, even if not restricting interval boundaries to integral coordinates:

Proof of Theorem 2.22. We propose an algorithm for the general MAXIMUM WEIGHTED STABLE INTERVAL COVERING PROBLEM, i.e. not restricted to integral interval boundaries. Like the INTEGRAL MAXIMUM WEIGHTED STABLE INTERVAL COVERING ALGORITHM, this algorithm will (implicitly) compute a non-decreasing step function $s : [0,U] \to \mathbb{R}_+$ that for any $y \in [0,U]$ gives the optimum value of a solution X under the restriction that $X \subseteq [0,y]$. However, the function values of s at the *step points*

$$Z := \{0\} \cup \{z \in (0,U] : s(z - \varepsilon) < s(z)\ \forall \varepsilon > 0\},$$

are not stored explicitly. Instead, our algorithm performs U/d phases and maintains a set V that at the end of the i-th phase is equal to

$$V_i := \{z - (i-1)d : z \in Z \cap B_i\},$$

where $B_i := [(i-1)d, id)$, and a function $\tilde{s} : V \to \mathbb{R}_+$ with

$$\tilde{s}(v) = \tilde{s}_i(v) := s(v + (i-1)d)$$

for $v \in V$ at the end of phase i, $0 \le i \le U/d$ (the initialization step before the first phase starts will be called phase 0). For this purpose it will consider the intervals

$$\mathcal{J}_i := \{I \in \mathcal{I} : \{l(I), u(I)\} \cap B_i \neq \emptyset\}$$

and

$$\mathcal{J}'_i := \{I \in \mathcal{I} : B_i \subseteq I \text{ and } \{l(I), u(I)\} \cap B_i = \emptyset\}.$$

in phase i, $1 \le i \le U/d$. Our algorithm generates backtrace information for elements in $V_{i+1} \setminus V_i$ that allows to construct an optimal solution after the last phase has been executed.

We store the set V as vertices of a binary search tree $T := (V, E)$. We also write $V(T) := V$. T will be an AVL tree (see Adel'son-Vel'skiĭ and Landis [1962]), which is a binary search tree that maintains a *balance value* for each vertex that is defined as the difference of the subtree heights at this vertex. If there is a vertex with balance value not in $\{-1, 0, 1\}$ after insertion or deletion of an element, a series of rebalancing steps (called rotations and double rotations) is carried out to ensure that all balance values are -1, 0 or 1. Adel'son-Vel'skiĭ and Landis [1962] show that by this balancing the height of an AVL tree with n vertices can be bounded by $O(\log n)$, and rebalancing the tree after an insertion or deletion requires $O(\log n)$ rotation or double rotation operations each of which takes constant time.

The function $\tilde{s}$ is not stored explicitly. Instead we store a value $\sigma(v)$ for each $v \in V$ and define $\tilde{s}(v) := \sum_{v' \in V(P)} \sigma(v')$, where P is the path from v to the root of T in T. Whenever a rotation or double rotation operation is carried out to rebalance T, we modify σ such that $\tilde{s}$ does not change. This can be done in constant time. Also if an element is removed from $V(T)$, σ is modified such that $\tilde{s}$ does not change for the remaining elements. Again, this can be done in constant time.

We need the following definitions to present our algorithm. $\text{root}(T) \in V$ denotes the root node of T. For two adjacent nodes v_1 and v_2 in T, let

$$\text{rel}(v_1, v_2) := \begin{cases} \nearrow : v_1 \text{ is left child of } v_2 \\ \nwarrow : v_1 \text{ is right child of } v_2 \\ \searrow : v_2 \text{ is right child of } v_1 \\ \swarrow : v_2 \text{ is left child of } v_1 \end{cases}$$

denote the relationship between v_1 and v_2.

The algorithm is shown on page 38. Iterations of the outer loop starting in line 2 are called *phases* 1 to U/d. Phase 0 consists of line 1. To prove correctness, first observe that for each $z \in Z$ with $z - d \notin Z$, there must be an $I \in \mathcal{I}$ with $z = l(I)$, and thus the number $z' := z \bmod d$ is inserted into $V(T)$ in line 4 in phase $\lfloor z/d \rfloor + 1$. We now inductively show that at the end of phase i ($0 \le i \le U/d$) we have $V(T) = V_i$ and

MAXIMUM WEIGHTED STABLE INTERVAL COVERING ALGORITHM

```
Input  : A set $\mathcal{I}$ of intervals in $\mathbb{R}$, weights $w : \mathcal{I} \to \mathbb{R}_+ \setminus \{0\}$, a number $d > 0$.
Output: Numbers $X \subseteq [0,U]$ maximizing $\tilde{w}(X)$, with $|x_1 - x_2| \geq d$ for any
        $x_1, x_2 \in X$ with $x_1 \neq x_2$.
1  Set $T := (\emptyset, \emptyset)$. Set $\rho := -1$ and $\hat{v}_0 := 0$.
2  for $i = 1$ to $U/d$ do
3      foreach $I \in \mathcal{J}_i$ with non-increasing $l(I)$ do
4          if $v := l(I) - (i-1)d \geq 0$ and $v \notin V(T)$ then
5              Set $\psi(v) := i$, $\sigma(v) := 0$ and insert $v$ as a leaf into $T$. Rebalance $T$.
6              if $\exists v' \in V(T) : v' < v$ then
7                  Let $v' := \max\{v'' \in V(T) : v'' < v\}$. Set $p_i(v) := (v', \psi(v'), i-1)$.
8              else if $i > 2$ then
9                  Let $j := -\lfloor \hat{v}_{i-2}/d \rfloor$. Set $p_i(v) := (\hat{v}_{i-2} + jd, \phi_{i-2}, i-2-j)$.
10             else
11                 Set $p_i(v) := (0,0,0)$.
12     Set $\sigma(\mathrm{root}(T)) := \sigma(\mathrm{root}(T)) + w(\mathcal{J}'_i)$.
13     foreach $I \in \mathcal{J}_i$ with non-increasing $u(I)$ do
14         Let $v^l := \min\{w \in V(T) : w \geq l(I) - (i-1)d\}$.
15         Let $v^u := \max\{w \in V(T) : w \leq u(I) - (i-1)d\}$.
16         Let $v_1, \ldots, v_k$ $(k \in \mathbb{N})$ be the vertex sequence on the $v^l$-$v^u$-path in $T$.
17         Set dir $:= \nearrow$ and $\Delta := w(I)$.
18         for $n = 1$ to $k-1$ do
19             if $dir \neq rel(v_n, v_{n+1})$ then
20                 Set $\sigma(v_n) := \sigma(v_n) + \Delta$.
21                 Set $\Delta := -\Delta$.
22             Set dir $:= \mathrm{rel}(v_n, v_{n+1})$.
23         if $dir \in \{\nearrow, \swarrow\}$ then $\sigma(v_k) := \sigma(v_k) + w(I)$.
24         if $\exists v$ left child of $v^l$ then $\sigma(v) := \sigma(v) - w(I)$.
25         if $\exists v$ right child of $v^u$ then $\sigma(v) := \sigma(v) - w(I)$.
26         foreach $v \in V(T)$ with $v > v^u$ and $\tilde{s}(v) \leq \tilde{s}(v^u)$ do
27             Delete $v$ from $T$. Rebalance $T$.
28     foreach $v \in V(T)$ with $\tilde{s}(v) \leq \rho$ do
29         Delete $v$ from $T$. Rebalance $T$.
30     Set $\hat{v}_i := \max\{v : v \in V(T) \cup \{\hat{v}_{i-1} - d\}\}$.
31     if $\hat{v}_i \geq 0$ then set $\phi_i := \psi(\hat{v}_i)$ and $\rho := \tilde{s}(\hat{v}_i)$ else set $\phi_i := \phi_{i-1}$.
32 Set $X := \emptyset$, $v := \hat{v}_{U/d}$, $i_{\mathrm{start}} := \psi(v)$ and $i_{\mathrm{end}} := U/d$.
33 while $(v, i_{start}, i_{end}) \neq (0,0,0)$ do
34     Set $X := X \cup \{v + (i-1)d : i_{\mathrm{start}} \leq i \leq i_{\mathrm{end}}\}$.
35     Set $(v, i_{\mathrm{start}}, i_{\mathrm{end}}) := p_{i_{\mathrm{start}}}(v)$.
36 return $X$.
```

$\tilde{s}(v) = s(f_i(v))$ for all $v \in V(T)$, where $f_i(v) := v + (i-1)d$ for $v \in V(T)$ and $0 \leq i \leq U/d$. Clearly, this claim is true for $i = 0$.

So let now $1 \leq i \leq U/d$. For convenience let us define $s(y) := 0$ for $y < 0$. We have $V_i \subseteq V(T)$ immediately before line 12 is executed in phase i, and further (by the induction hypothesis) $\tilde{s}(v) = s(f_{i-1}(v))$ for each $v \in V(T)$. In line 12 and lines 17 to 25 σ is then modified such that we have

$$\tilde{s}(v) = s(f_{i-1}(v)) + \tilde{w}(\{f_i(v)\}) \leq s(f_i(v)) \tag{2.6}$$

for each $v \in V(T)$ when line 28 is reached, with equality for $v \in V_i$. To verify this, observe that Δ is always positive in line 20 if v_n is the vertex closest to the root on the v^l-v^u-path in T. For an element $x \in V(T)$ with $x > v^u$ in line 26, $\tilde{s}(x)$ cannot increase any more in phase i. As $\tilde{s}$ never decreases, $x \notin Z$ if $\tilde{s}(v^u) \geq \tilde{s}(x)$, so x can be removed from $V(T)$ in this case. Similarly, $\tilde{s}(x) \leq \rho$ for an element $x \in V(T)$ in line 28 implies $x \notin Z$ since by induction hypothesis there is a $0 \leq y < (i-1)d$ with $s(y) = \rho$, so x can be removed from $V(T)$ (observe that this does not happen in phase 1). So we have $V_i \subseteq V(T)$ also at the end of phase i. When line 30 is reached, $\tilde{s}(v) > \rho$ for each $v \in V(T)$, and $\tilde{s}$ is strictly increasing on $V(T)$, so we have indeed $V_i = V(T)$. As (2.6) holds with equality for $v \in V_i$, this proves the claim.

We now verify that the returned set X is an optimum solution to the given instance. Let X' be an optimum solution, and $x \in X'$. If $x \geq d$, we can assume w.l.o.g. that

$$z' := \max\{z \in Z : z \leq x - d\} \in X' \tag{2.7}$$

(observe that $0 \in Z$), because by definition of Z and s there is a feasible solution $X'' \subseteq [0, z']$ with

$$\tilde{w}(X'') \geq \tilde{w}\left(X' \cap [0, x-d]\right),$$

hence $X'' \cup (X' \cap [x, U])$ is an optimum solution.

At the end of the last phase we have

$$\max\{z \in Z\} = \hat{v}_{U/d} + U - d.$$

As $s(x) \leq s(\max\{z \in Z\})$ for any $x \in [0, U]$, there is an optimum solution that contains the number $\hat{v}_{U/d} + U - d$, so we add it to X the first time line 34 is executed. Using (2.7), the solution can now be completed as follows: If $x = v + (i-1)d$ is added to X ($0 \leq v < d$ and $1 \leq i \leq U/d$), we have $v \in V_i$. If also $v \in V_{i-1}$, $x - d$ is in Z and by (2.7) can be added to X as well. Otherwise v has been added to $V(T)$ in phase i. If there is an element $v' \in V(T)$ with $v' < v$ at the time v is added to $V(T)$ in phase i, we have $v' \in V_{i-1}$ because the intervals $I \in \mathcal{J}_i$ are processed in the order of non-increasing left boundaries. Therefore in line 7 we have $v' := \max\{v'' \in V(T) : v'' < v\} = \max\{z \in Z : z \leq x - d\} - (i-2)d$. Moreover, $v' \in V_{i'}$ for each $\psi(v') \leq i' \leq i - 1$, but $v' \notin V_{\psi(v')-1}$. The values v', $\psi(v')$ and $i - 1$ are stored in $p_i(v)$ and used in lines 32 to 35 in order to add $\{v' + (i'-1)d : \psi(v') \leq i' \leq i - 1\}$ to X.

If there is no $v' \in V(T)$ with $v' < v$ in line 6, but there is a $z \in Z$ with $z < v + (i-2)d$, we must have $i > 2$ and

$$z_{\max} := \max\{z \in Z : z \leq x - d\} = \hat{v}_{i-2} + (i-3)d.$$

Similarly as above, $p_i(v)$ is set to $(z_{\max} \bmod d, \phi_{i-2}, i-2-j)$ in line 9, and $\{(z_{\max} \bmod d) + (i'-1)d : \phi_{i-2} \leq i' \leq i-2-j\}$ can be added to X.

We finally prove the claim on runtime. It is straightforward to see that generating the sets $\mathcal{J}_i$ and $\mathcal{J}'_i$ for each $1 \leq i \leq U/d$ is possible in time $O((U/d + |\mathcal{I}|)\log|\mathcal{I}|)$ in total. Lines 4 to 11 are executed $O(|\mathcal{I}|)$ times, so the height of T is bounded by $O(\log|\mathcal{I}|)$ and hence total runtime spent in lines 4 to 11 is in $O(|\mathcal{I}|\log|\mathcal{I}|)$. The time needed in lines 12 to 29 for updating σ and deleting elements from $V(T)$ can be bounded by $O(U/d + |\mathcal{I}|\log|\mathcal{I}|)$: Line 12 is executed U/d times, and total runtime spent in lines 14 to 25 clearly can be bounded by $O(|\mathcal{I}|\log|\mathcal{I}|)$. The same holds for the runtime spent in lines 26 and 27: Because intervals are processed in the order of non-increasing right boundaries, $\tilde{s}$ strictly increases on $\{v \in V(T) : v > v^u\}$. Therefore the runtime spent in line 26 to find all elements $v \in V(T)$ with $v > v^u$ and $\tilde{s}(v) \leq \tilde{s}(v^u)$ can be bounded by the runtime of line 27 which obviously is in $O(|\mathcal{I}|\log|\mathcal{I}|)$ since the algorithm does only $|\mathcal{I}|$ insertions into $V(T)$ in total and thus can perform at most $|\mathcal{I}|$ deletions. Similarly, the runtime of lines 28 and 29 can be bounded by $O(|\mathcal{I}|\log|\mathcal{I}|)$. As construction of X at the end of the algorithm takes linear time, the claim follows. □

2.4.2 Application

We now discuss how the MAXIMUM WEIGHTED STABLE INTERVAL COVERING ALGORITHM can be used to determine routing tracks. Given a set $\mathcal{B}_p$ of blockage shapes in plane $p \in P$, assume we want to find routing track locations such that the total length of those parts of the tracks that are simultaneously usable (i.e. without violating any minimum distance rules) by on-track wire segments with a certain standard wire type t is maximized.

Assume plane p is vertical, and there is no existing wiring yet. Let $t(p) = (R, c)$ be the wire model of wire type t on plane p, and $\mathcal{R}$ be the set of minimum distance rules that apply on plane p. First we compute a set

$$\mathcal{B}' := \left\{B - D - R : (B, c') \in \mathcal{B}, (c, c', D) \in \mathcal{R}\right\}$$

of *extended blockages* that define an area

$$\mathcal{A}_p \setminus \bigcup_{B \in \mathcal{B}'} B^\circ$$

of points where a wire segment with wire type t can be legally placed. We approximate each element $B \in \mathcal{B}'$ by a constant number of rectangles and call the resulting set $\mathcal{B}''$. Since in practice most elements of B extend very long in preferred direction, this simplification introduces only a small error. By a standard sweep line method as in the book of Preparata and Shamos [1988] we now compute a set $\bar{\mathcal{B}}$ of disjoint axis-parallel rectangles within $\mathcal{A}_p$ such that

$$\bigcup_{B\in\bar{\mathcal{B}}}(B\cap\mathbb{Z}^3) = \left(\mathcal{A}_p \setminus \bigcup_{B\in\mathcal{B}''} B^\circ\right)\cap\mathbb{Z}^3.$$

We can compute $\bar{\mathcal{B}}$ in $O(|\bar{\mathcal{B}}| + |\mathcal{B}|\log|\mathcal{B}|)$ time. Although $|\bar{\mathcal{B}}|$ can be in $\Omega(|\mathcal{B}|^2)$, it is linear in $O(|\mathcal{B}|)$ in practice. As all vias and wire segments shall begin and end on integral coordinates, the restriction to integer points makes no difference.

Now it is easy to compute a set of $O(|\bar{\mathcal{B}}|)$ intervals $\mathcal{I}$ and weights $w : \mathcal{I} \to \mathbb{R}_+$ such that

$$\tilde{w}(\{x\}) = \sum_{B\in\bar{\mathcal{B}}:x\in(x(B))} |B|_y.$$

for each $x \in x(\mathcal{A}_p)$. We finally discard intervals with zero weight and start the MAXIMUM WEIGHTED STABLE INTERVAL COVERING ALGORITHM with $d := \Delta_p^{\min}$, i.e. the minimum pitch on plane p.

If some wires already exist in the input, we might want the tracks to align with existing wire segments. To this end, let $\mathcal{S}_p$ be the wire segments on plane p running in preferred (i.e. vertical) direction. We then add the zero-length intervals

$$\mathcal{I}' := \{x(\text{area}(S)) : S \in \mathcal{S}_p\}$$

to $\mathcal{I}$ with weight

$$w(I) := \sum_{\substack{S\in\mathcal{S}_p\\ x(\text{area}(S))=I}} k_1|\text{area}(S)|_y$$

for $I \in \mathcal{I}'$, where $k_1 \in \mathbb{R}_+ \setminus \{0\}$ is some constant value given as parameter.

Further we want to align tracks with pins in such a way that for a large fraction of pins there is an on-track pin access stub that obeys all design rules (including same net rules). An on-track pin access stub is a via or wire segment (w,t) running in preferred direction and of length within a specified interval, and is required to start at a point contained in one of the pin's shapes and to end at an on-track point outside the pin shapes. $t \in \mathcal{T}$ here denotes the wire type of the corresponding net.

We do not specify the conditions under which a pin access stub obeys all design rules. For our purposes it suffices that we can compute a (possibly empty) interval $I \subseteq x(\mathcal{A}_p)$ for each pin with the property that if a track is created at coordinate $x \in I$ in plane p, there is an on-track pin access stub for this pin which ends on this track and obeys all design rules. We add all intervals obtained in this way to $\mathcal{I}$ and set their weights to some constant value $k_2 \in \mathbb{R}_+ \setminus \{0\}$. Of course we might assign a higher weight if there is a stub running in the "right" direction, e.g. towards the center of gravity of all pin shapes of the net or towards the closest other pin of the net.

2.5 Discussion and Outlook

Restricting all connections to run on-track with exceptions allowed only for pin access of course may cause an otherwise routable design to become unroutable. However, the design characteristics of current technologies for which BonnRoute® is used are in favour of this simplification as almost no routing space is wasted in practice. One key property exhibited by far by most current designs is that there is one prevalent standard wire type $t \in \mathcal{T}$ that is used for the greatest part of the wiring, and this wire type attains $\Delta_p(t,t) = \Delta_p^{\min}$ on all wiring planes. A second property is that, at least if stick figures are not required to be centered within wire shapes, for each wiring plane $p \in P$ there is a number $\delta_p \geq \Delta_p^{\min}$ such that $\Delta_p(t_1,t_2)$ is an (almost) integral multiple of δ_p for pairs $(t_1,t_2) \in \mathcal{T}_p \times \mathcal{T}_p$ of "substantially often used" wire types, i. e. there is a small $\varepsilon \geq 0$ such that

$$\delta_p k(t_1,t_2,p) - \varepsilon \leq \Delta_p(t_1,t_2) \leq \delta_p k(t_1,t_2,p) \tag{2.8}$$

with $k : \mathcal{T} \times \mathcal{T} \times P \to \mathbb{N}$. In practice usually $\delta_p = \Delta_p^{\min}$ holds.

In future technologies, these properties may not be granted any more, making it more difficult to use routing resources efficiently without increasing computing time too much. Specifically, without a prevalent value of $\Delta_p(t_1,t_2)$ within a layer $p \in P$ ($t_1,t_2 \in \mathcal{T}_p$), one problem to be addressed in a sequential routing approach is to control gaps unusable for any wire shapes between wires that are not closest possible to each other. In some regions of the chip — where enough space is available — such gaps may be desirable to reduce coupling, while in other more "dense" regions routability may be achieved only after closing such gaps by a compactification of already routed wires, or avoiding emergence of such gaps as far as possible. It is questionable if standard ripup-and-reroute techniques suffice to achieve this goal. We remark that for this reason, without substantial modifications in BonnRoute®, routability might actually degrade by adding tracks between the currently used ones, even though it improves in theory.

If the second property does not hold any more, even on easily routable chips the restriction to on-track vias and wire segments may prohibit the *existence* of a routing solution, at least unless the distance between tracks is considerably reduced. Reducing the track-to-track distances however increases the size of the track graph, the number of intervals stored in the shape grid and fast grid data structures, and thus also the runtime of the interval based Dijkstra algorithm used in BonnRoute®.

To keep the size of the track graph manageable, an extended fast grid data structure could store precomputed information on legality of a wire segment of a certain wire type in a region around a track. Whereever the width of this region changes, intervals have to be splitted, and short jogs might be necessary to continue a wire (if this is possible at all). Of course special care is required in the path search algorithm to keep track of the wiring length (which is to be minimized, but is increased by these jogs), and to avoid so-called short-edge errors which are a class of same net rule violations (see Peyer [2007] and Cho et al. [2009b]).

The fast grid data structure may help to support more complex design rules in the future. Such rules e.g. might prescribe a minimum distance between two shapes which depends on the existence of another shape of certain type close to one of the shapes. Also more general two-dimensional patterns that are difficult to produce with the available lithography technology may be prohibited in the future, or should at least be used only at a high penalty cost. See Dai et al. [2009] for examples, and Cho et al. [2009b] and Pan et al. [2008] for an overview over manufacturing-related issues and further references. The information if placing a wire segment or via somewhere would generate such a pattern may be considerably more difficult to compute than checking minimum distance requirements between pairs of shapes, so precomputing (and updating) this information can save even more time.

Chapter 3

Resource Sharing

The problem of sharing a set of limited resources between users (customers) in an optimal way is fundamental. The common mathematical model has been called the min-max resource sharing problem. Well-studied special cases are the fractional packing problem and the maximum concurrent flow problem. The only known exact algorithms for these problems use general linear (or convex) programming. Shahrokhi and Matula [1990] were the first to design a combinatorial approximation scheme for an important special case, the maximum concurrent flow problem. Subsequently, this result was improved, simplified, and generalized many times.

The results presented in this chapter are a further step on this line. In particular we provide a simple algorithm and a simple proof of the best performance guarantee in significantly smaller running time. For even further reducing running time, we present an efficient lock-free shared-memory-parallel variant of this algorithm. This algorithm is non-deterministic as it allows concurrent allocations of the same resource by different processors, each of them unaware of the others. Although keeping the frequency of such concurrent accesses low is important for obtaining good parallelization speedups, our algorithm does not rely on this for correctness.

The MIN-MAX RESOURCE SHARING PROBLEM has many applications (e.g., Garg and Könemann [2008], Grigoriadis and Khachiyan [1994], Jansen and Zhang [2008]). In section 4 we show how the resource sharing algorithm can be applied to global routing of VLSI chips to optimize practically relevant objectives such as power and manufacturing yield while at the same time obeying constraints on signal delay or crosstalk sensitivity. These constraints and objectives naturally give rise to instances of the (nonlinear) MIN-MAX RESOURCE SHARING PROBLEM. An implementation of the shared-memory-parallel version of our algorithm developed as part of the BonnRoute® program package at the Research Institute for Discrete Mathematics in Bonn is used today at IBM for routing almost all chip designs, solving problem instances with several million resources and customers in less than an hour. The experimental results presented in chapter 4 also demonstrate that a very good parallelization speedup can be reached in practice.

This chapter is organized as follows. Section 3.1 introduces the MIN-MAX RESOURCE SHARING PROBLEM and gives an overview of the previously known results and of our contribution. Section 3.2 presents the sequential version of our algorithm and analyzes its runtime, also showing that the worst-case runtime bound w.r.t. the approximation parameter ε obtained in the analysis is tight. With small modifications, this algorithm and its analysis correspond to our publication in Müller and Vygen [2008]. In section 3.3 we present the new lock-free shared-memory-parallel variant of our algorithm. In many applications, including VLSI routing, a discrete problem has to be solved, but the formulation as a resource sharing problem actually yields a fractional relaxation of the original problem. In this case a *randomized rounding* step generates an integral solution from the fractional solution obtained from the MIN-MAX RESOURCE SHARING ALGORITHM. This is discussed in section 3.4.

3.1 Problem Statement and Overview

The MIN-MAX RESOURCE SHARING PROBLEM is defined as follows.

MIN-MAX RESOURCE SHARING PROBLEM

Instance: Finite sets $\mathcal{R}$ of *resources* and $\mathcal{C}$ of *customers*, a convex set $\mathcal{B}_c$, called *block*, of feasible solutions for each customer $c \in \mathcal{C}$, and a nonnegative continuous convex function $g_c : \mathcal{B}_c \to \mathbb{R}^{\mathcal{R}}_{+}$ for $c \in \mathcal{C}$ specifying the *resource consumptions* of each element $b \in \mathcal{B}_c$.

Task: Find $b_c \in \mathcal{B}_c$ $(c \in \mathcal{C})$ approximately attaining

$$\lambda^* := \inf\left\{ \max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(b_c))_r \,\middle|\, b_c \in \mathcal{B}_c \,(c \in \mathcal{C}) \right\}, \tag{3.1}$$

i.e., approximately minimizing the largest resource consumption.

We assume that g_c can be computed efficiently and we have a constant $\varepsilon_0 \geq 0$ and oracle functions $f_c : \mathbb{R}^{\mathcal{R}}_{+} \to \mathcal{B}_c$, called *block solvers*, which for $c \in \mathcal{C}$ and $\omega \in \mathbb{R}^{\mathcal{R}}_{+}$ return an element $b_c \in \mathcal{B}_c$ with $\omega^\top g_c(b_c) \leq (1+\varepsilon_0)\mathrm{opt}_c(\omega)$, where $\mathrm{opt}_c(\omega) := \inf_{b\in\mathcal{B}_c} \omega^\top g_c(b)$. Block solvers are called *strong* if $\varepsilon_0 = 0$ or $\varepsilon_0 > 0$ can be chosen arbitrarily small, otherwise they are called *weak*.

Note that previous authors often required that $\mathcal{B}_c$ is compact, but we do not need this assumption. Some algorithms require *bounded block solvers*: for $c \in \mathcal{C}$, $\omega \in \mathbb{R}^{\mathcal{R}}_{+}$, and $\mu > 0$, they return an element $b_c \in \mathcal{B}_c$ with $g_c(b_c) \leq \mu\mathbb{1}$ and $\omega^\top g_c(b_c) \leq (1+\varepsilon_0)\inf\{\omega^\top g_c(b) \mid b \in \mathcal{B}_c, g_c(b) \leq \mu\mathbb{1}\}$ (by $\mathbb{1}$ we denote the all-one vector). They can also be strong or weak.

All algorithms that we consider are fully polynomial approximation schemes relative to ε_0, i.e., for any given $\varepsilon > 0$ they compute a solution $b_c \in \mathcal{B}_c$ $(c \in \mathcal{C})$ with $\max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}}(g_c(b_c))_r \leq (1+\varepsilon_0+\varepsilon)\lambda^*$, and the running time depends polynomially on ε^{-1}. By θ we denote the time for an oracle call (to the block solver). Moreover, we write

$$\rho := \max\left\{1, \sup\left\{\frac{(g_c(b))_r}{\lambda^*} \mid r \in \mathcal{R}, c \in \mathcal{C}, b \in \mathcal{B}_c\right\}\right\},$$

denoting the *width* of the problem instance. Often $\rho = 1$ in practical applications.

3.1.1 Previous work

Grigoriadis and Khachiyan [1994] were the first to present an algorithm for the general MIN-MAX RESOURCE SHARING PROBLEM. Their algorithm uses $O(|\mathcal{C}|^2 \log|\mathcal{R}|(\varepsilon^{-2}+\log|\mathcal{C}|))$ calls to a strong bounded block solver. They also have a faster randomized version.

In a later work (Grigoriadis and Khachiyan [1996]) they proposed an algorithm which needs $O(|\mathcal{C}||\mathcal{R}|(\varepsilon^{-2}\log\varepsilon^{-1}+\log|\mathcal{R}|))$ calls to a strong, but not bounded, block solver. They also showed that $O(|\mathcal{C}|^2 \log|\mathcal{R}|(\varepsilon^{-2}+\log|\mathcal{R}|))$ calls to a strong bounded block solver suffice.

Jansen and Zhang [2008] generalized this and allowed *weak block solvers*. Their algorithm needs $O(|\mathcal{C}||\mathcal{R}|(\log|\mathcal{R}|+\varepsilon^{-2}\log\varepsilon^{-1}))$ calls to a block solver.

Khandekar [2004] needs only $O((\varepsilon^{-2}+\log\log|\mathcal{R}|)(|\mathcal{C}|+|\mathcal{R}|)(\log(|\mathcal{C}|+|\mathcal{R}|)))$ calls to an unbounded oracle, which matches our runtime if $|\mathcal{R}| = O(\rho|\mathcal{C}|)$. It is not clear from his work however if this bound can also be achieved with weak block solvers.

	block solver	running time
Grigoriadis and Khachiyan [1994]	strong, bounded	$\tilde{O}(\varepsilon^{-2}|\mathcal{C}|^2\theta)$
Grigoriadis and Khachiyan [1996]	strong, unbounded	$\tilde{O}(\varepsilon^{-2}|\mathcal{C}||\mathcal{R}|\theta)$
Khandekar [2004]	strong, unbounded	$\tilde{O}(\varepsilon^{-2}(|\mathcal{C}|+|\mathcal{R}|)\theta)$
Jansen and Zhang [2008]	weak, unbounded	$\tilde{O}(\varepsilon^{-2}|\mathcal{C}||\mathcal{R}|\theta)$
our algorithm	weak, unbounded	$\tilde{O}(\varepsilon^{-2}\rho|\mathcal{C}|\theta)$
our algorithm	weak, bounded	$\tilde{O}(\varepsilon^{-2}|\mathcal{C}|\theta)$

Table 3.1: Approximation algorithms for the MIN-MAX RESOURCE SHARING PROBLEM. Running times are shown for fixed $\varepsilon_0 \geq 0$, and logarithmic terms are omitted.

3.1.2 Fractional packing

The special case where the functions g_c $(c \in \mathcal{C})$ are linear is often called the FRACTIONAL PACKING PROBLEM (although sometimes this name is used for different problems). For this special case faster algorithms using unbounded block solvers are known.

Plotkin, Shmoys and Tardos [1995] require a strong block solver and at most $O(\varepsilon^{-2}\rho|\mathcal{C}|(\log|\mathcal{R}|+\varepsilon^{-1}))$ calls to the block solver to solve the feasibility version (where $\lambda^*=1$ is known). The algorithm of Young [1995] needs $O(\varepsilon^{-2}\rho|\mathcal{C}|(1+\varepsilon_0)^2 \ln|\mathcal{R}|)$ calls to a weak block solver. Charikar et al. extended the result of Plotkin, Shmoys and Tardos [1995] to weak block solvers resulting in $O(\varepsilon^{-2}\rho|\mathcal{C}|(1+\varepsilon_0)^2\log(\rho(1+\varepsilon_0)\varepsilon^{-1}))$ oracle calls.

Bienstock and Iyengar [2006] managed to reduce the dependence on ε from $O(\varepsilon^{-2})$ to $O(\varepsilon^{-1})$. Their algorithm does not call a block solver, but requires the resource consumption functions to be explicitly specified by a $|\mathcal{R}|\times\dim(\mathcal{B}_c)$-matrix G_c for each $c\in\mathcal{C}$. So their algorithm does not apply to the general MIN-MAX RESOURCE SHARING PROBLEM, but to an interesting special case which includes the MAXIMUM CONCURRENT FLOW PROBLEM. The algorithm solves $O(\varepsilon^{-1}\sqrt{Kn}\log|\mathcal{R}|)$ separable convex quadratic programs, where $n:=\sum_{c\in\mathcal{C}}\dim(\mathcal{B}_c)$, and $K:=\max_{1\le i\le|\mathcal{R}|}\sum_{c\in\mathcal{C}}k_i^c$, with k_i^c being the number of nonzero entries in the i-th row of G_c.

	block solver	running time
Plotkin, Shmoys and Tardos [1995] $*$	strong, unbounded	$\tilde{O}(\varepsilon^{-2}\rho|\mathcal{C}|\theta)$
Young [1995]	weak, unbounded	$\tilde{O}(\varepsilon^{-2}\rho|\mathcal{C}|\theta)$
Charikar et al. [1998] $*$	weak, unbounded	$\tilde{O}(\varepsilon^{-2}\rho|\mathcal{C}|\theta)$
Bienstock and Iyengar [2006]	—	$\tilde{O}(\varepsilon^{-1}\sqrt{Kn}T_{QP})$
our algorithm	weak, unbounded	$\tilde{O}(\varepsilon^{-2}\rho|\mathcal{C}|\theta)$
our algorithm	weak, bounded	$\tilde{O}(\varepsilon^{-2}|\mathcal{C}|\theta)$

Table 3.2: Approximation algorithms for the fractional packing problem. Entries with $*$ refer to the feasibility version ($\lambda^*=1$). Running times are shown for fixed $\varepsilon_0\ge 0$, and logarithmic terms are omitted. T_{QP} is the time for solving a convex separable quadratic program over $\mathcal{B}_{c_1}\times\ldots\times\mathcal{B}_{c_{|\mathcal{C}|}}$.

3.1.3 Our Results

We describe an algorithm for the general MIN-MAX RESOURCE SHARING PROBLEM. It uses ideas of Grigoriadis and Khachiyan [1996], Young [1995], Albrecht [2001a,b], and Vygen [2004]. The same algorithm and a quite simple analysis yields two results: With a weak unbounded block solver we obtain a running time of $O(|\mathcal{C}|\theta\rho(1+\varepsilon_0)^2\log|\mathcal{R}|(\log|\mathcal{R}|+\varepsilon^{-2}(1+\varepsilon_0)))$. This generalizes several results for the linear case and improves on results for the general case for moderate values of ρ. With a weak bounded block solver the running time is $O(|\mathcal{C}|\theta(1+\varepsilon_0)^2\log|\mathcal{R}|(\log|\mathcal{R}|+\varepsilon^{-2}(1+\varepsilon_0)))$. This improves on previous results by roughly a factor of $|\mathcal{C}|$ or $|\mathcal{R}|$.

We incorporate a speed-up technique that drastically decreases the number of oracle calls in practice. We generalize the randomized rounding paradigm to our problem and obtain an improved bound. For experimental results, we refer to section 4.9. It turns out that instances from current chips, with millions of customers and resources can be solved efficiently in practice.

Remark. Consider the instance where $|\mathcal{C}| = 1$, $c \in \mathcal{C}$ is the only customer, g_c is the identity function, and $\mathcal{B}_c$ the convex hull of the unit vectors $\chi(r)$, $r \in \mathcal{R} \setminus \{r^*\}$, where r^* is a special resource which is not used by any of the feasible solutions $b \in \mathcal{B}_c$ for c. This corresponds to s-t-flows in the digraph containing only parallel arcs (s,t) of unit capacity. We call this example the parallel-arc example.

In this example the only (near-) optimal fractional solutions are convex combinations of (almost) all of the $|\mathcal{R} \setminus \{r^*\}|$ vectors. Thus the running time of any algorithm that outputs an explicit convex combination will depend at least linearly on $|\mathcal{R}|$. Moreover, any algorithm that has access to $\mathcal{B}_c$ only via the oracle does not know r^* and hence needs at least $\frac{|\mathcal{R}|}{2}$ oracle calls for an approximation ratio of 2. To get a speed-up we need some bound on ρ, or a bounded block solver.

3.2 A Sequential Algorithm

Our algorithm will perform a number $t \in \mathbb{N}$ of *phases*, in each phase finding a near-optimum element $b \in \mathcal{B}_c$ for each $c \in \mathcal{C}$ w.r.t. resource prices which we will define below depending on cumulative resource utilizations of all previously found solutions for each customer. For each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$, it counts in a variable $x_{c,b}$ how often solution b has been selected for customer c. In the end these variables, divided by t, will provide the coefficients of a convex combination of elements $b \in \mathcal{B}_c$ for each customer $c \in \mathcal{C}$ in a near-optimum solution to (3.1).

Our algorithm maintains a variable α_r for each $r \in \mathcal{R}$ which at any point in time before dividing the x-variables by t at termination stores the sum

$$\sum_{c \in \mathcal{C}} \left(\sum_{b \in \mathcal{B}_c} x_{c,b} g_c(b) \right)_r$$

of utilizations of resource r by the solutions selected so far. Although we will call $\alpha := (\alpha_r)_{r \in \mathcal{R}}$ the vector of *current resource utilizations*, because of convexity of the resource consumption functions it is an *upper bound* on resource utilizations

$$\sum_{c \in \mathcal{C}} \left(g_c \left(\sum_{b \in \mathcal{B}_c} x_{c,b} b \right) \right)$$

by the corresponding *convex combinations* (scaled by $\sum_{b \in \mathcal{B}_c} x_{c,b}$) for each customer $c \in \mathcal{C}$. We define resource *prices* that depend exponentially on given resource utilizations:

Definition 3.1. *Let $\psi \in \mathbb{R}_+^{\mathcal{R}}$ be a vector of* weights *with $\psi_r \geq 1$ for each $r \in \mathcal{R}$, and let $\varepsilon_2 > 0$. We define resource prices by $y_r : \mathbb{R}_+ \to \mathbb{R}_+$ with*

$$y_r(\zeta) := \psi_r e^{\varepsilon_2 \zeta}$$

for each resource $r \in \mathcal{R}$ with utilization $\zeta \in \mathbb{R}_+$.

The weights $\psi \in \mathbb{R}_+^{\mathcal{R}}$ will not affect the optimality guarantee given by our algorithm, and will increase worst-case runtime only by a factor $\log_{|\mathcal{R}|} \Psi$, where $\Psi := \sum_{r\in\mathcal{R}} \psi_r$. In practical applications one can achieve considerable runtime reductions by weighting resource prices, having to accept an increase of the worst-case bound by only a very small (constant) factor (see section 4.7.1).

For each $c \in \mathcal{C}$, we assume to have lower bounds $l_c \in \mathbb{R}_+^{\mathcal{R}}$ on resource consumption, i.e. satisfying $(l_c)_r \leq (g_c(b))_r$ for each $b \in \mathcal{B}_c$ and $r \in \mathcal{R}$. We can set $(l_c)_r = 0$ for all c and r without affecting the worst-case running time, but better lower bounds are essential for obtaining good running times in practice.

The algorithm is shown on page 51. Of course x is not stored explicitly, but rather by a set of triples $(c, b, x_{c,b})$ for those $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$ for which $x_{c,b} > 0$. Also $\tilde{\omega}$ does not need to be stored explicitly, but is computed on-the-fly from α and $L := \sum_{c\in\mathcal{C}} l_c$, which is computed once during initialization.

We will call the outer iterations ($p = 1, \ldots, t$) of the algorithm *phases*. To simplify notation we assume $\mathcal{C} = \{1, \ldots, k\}, k \in \mathbb{N}$. Let $\alpha^{(p,c)}$, $\tilde{\omega}^{(p,c)}$, $a_c^{(p)}$, $z_c^{(p)}$ and $\phi_c^{(p)}$ denote the respective values of α, $\tilde{\omega}$, a_c, z_c and ϕ_c at the end of the c-th inner iteration within the p-th phase ($c \in \mathcal{C}$, $p = 1, \ldots, t$). Let $\alpha^{(p)} := \alpha^{(p,k)}$ be the resource utilizations at the end of phase p. We write $\alpha^{(p,0)} := \alpha^{(p-1)}$ etc., and $\alpha^{(0)}$ is the value of α after initialization.

From a theoretical point of view, it makes no difference if the block solvers are called with resource prices

$$\tilde{\omega} = \big(y_r\big(\max\big\{\alpha_r, \alpha_r^{(p-1)} + L_r\big\}\big)\big)_{r\in\mathcal{R}} \tag{3.2}$$

in phase p ($1 \leq p \leq t$), or with $\big(y_r\big(\alpha_r\big)\big)_{r\in\mathcal{R}}$. With the latter choice however, in early iterations of the algorithm prices of resources which are shared by many customers are extremely small for the first customers that are processed. Making use of our knowledge that resource prices will be at least $\big(y_r\big(\alpha_r^{(p-1)} + L_r\big)\big)_{r\in\mathcal{R}}$ at the end of phase p ($1 \leq p \leq t$) can help the algorithm to converge faster towards optimality in practice.

Remark. In many applications, $L_r > 0$ only for a very small number of resources $r \in \mathcal{R}$, and of course storing $\alpha_r^{(p-1)}$ is necessary only for these resources to compute the maximum in (3.2). If there are many resources $r \in \mathcal{R}$ with $L_r > 0$, it might be preferable to use resource prices $\big(y_r\big(\max\big\{\alpha_r, pL_r\big\}\big)\big)_{r\in\mathcal{R}}$ instead of $\big(y_r\big(\max\big\{\alpha_r, \alpha_r^{(p-1)} + L_r\big\}\big)\big)_{r\in\mathcal{R}}$ to save memory (and memory bandwidth).

```
RESOURCE SHARING ALGORITHM
Input : An instance of the MIN-MAX RESOURCE SHARING PROBLEM, i.e.,
        finite sets $\mathcal{R}$ and $\mathcal{C}$, and for each $c \in \mathcal{C}$ an oracle function $f_c : \mathbb{R}_+^{\mathcal{R}} \to \mathcal{B}_c$
        with the property $\omega^\top g_c(f_c(\omega)) \le (1+\varepsilon_0)\,\mathrm{opt}_c(\omega)$ for all $\omega \in \mathbb{R}_+^{\mathcal{R}}$ and
        some $\varepsilon_0 \ge 0$. Lower bounds $l_c \in \mathbb{R}_+^{\mathcal{R}}$ for each customer $c \in \mathcal{C}$, and
        weights $\psi_r \ge 1$ for each resource $r \in \mathcal{R}$. Parameters $\varepsilon_1 \ge 0$, $\varepsilon_2 > 0$, and
        $t \in \mathbb{N}$.
Output: For each $c \in \mathcal{C}$ a convex combination of vectors in $\mathcal{B}_c$ , given by
        $\sum_{b \in \mathcal{B}_c} x_{c,b} b$. A cost vector $\omega \in \mathbb{R}_+^{\mathcal{R}}$.
Set $\alpha_r := 0$ for each $r \in \mathcal{R}$.
Set $x_{c,b} := 0$ for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$.
Compute $L := \sum_{c \in \mathcal{C}} l_c$.
for $p := 1$ to $t$ do
    foreach $c \in \mathcal{C}$ do
        AllocateResources($c$).
Set $x_{c,b} := \frac{1}{t} x_{c,b}$ for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$.
Set $\omega_r := y_r(\alpha_r)$ for each $r \in \mathcal{R}$.
Return $(x, \omega)$.

Procedure AllocateResources($c \in \mathcal{C}$):
begin
    Let $\tilde{\omega} \in \mathbb{R}_+^{\mathcal{R}}$ with $\tilde{\omega}_r := y_r\left(\max\left\{\alpha_r, \alpha_r^{(p-1)} + L_r\right\}\right)$ for $r \in \mathcal{R}$.
    if $p = 1$ or $\tilde{\omega}^\top a_c > (1+\varepsilon_1)(z_c + (1+\varepsilon_0)(\tilde{\omega}^\top l_c - \phi_c))$ then
        Set $b_c := f_c(\tilde{\omega})$.
        Set $a_c := g_c(b_c)$.
        Set $z_c := \tilde{\omega}^\top a_c$.
        Set $\phi_c := \tilde{\omega}^\top l_c$.
    Set $x_{c,b_c} := x_{c,b_c} + 1$ and $\alpha := \alpha + a_c$.
end
```

3.2.1 Analysis

The first Lemma provides us with a lower bound on the optimum value of (3.1).

Lemma 3.2. *Let $\omega \in \mathbb{R}_+^{\mathcal{R}}$ be some cost vector with $\omega^\top \mathbb{1} \neq 0$. Then*

$$\frac{\sum_{c\in\mathcal{C}} \mathrm{opt}_c(\omega)}{\omega^\top \mathbb{1}} \leq \lambda^*.$$

Proof. Let $\delta > 0$ and $(b_c \in \mathcal{B}_c)_{c\in\mathcal{C}}$ with $\max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(b_c))_r < (1+\delta)\lambda^*$. We have

$$\frac{\sum_{c\in\mathcal{C}} \mathrm{opt}_c(\omega)}{\omega^\top \mathbb{1}} \leq \frac{\sum_{c\in\mathcal{C}} \omega^\top g_c(b_c)}{\omega^\top \mathbb{1}} < \frac{\omega^\top (1+\delta)\lambda^* \mathbb{1}}{\omega^\top \mathbb{1}} = (1+\delta)\lambda^*.$$

□

Lemma 3.3. *Let (x,ω) be the output of the* RESOURCE SHARING ALGORITHM, *and let*

$$\lambda_r := \sum_{c\in\mathcal{C}} \left(g_c \left(\sum_{b\in\mathcal{B}_c} x_{c,b} b \right) \right)_r$$

and $\lambda := \max_{r\in\mathcal{R}} \lambda_r$. Then

$$\lambda \leq \frac{1}{\varepsilon_2 t} \ln \sum_{r\in\mathcal{R}} e^{\varepsilon_2 t \lambda_r} \leq \frac{1}{\varepsilon_2 t} \ln\big(\omega^\top \mathbb{1}\big).$$

Proof. The first inequality is trivial. Using the convexity of the functions g_c we have for $r \in \mathcal{R}$:

$$\lambda_r \leq \sum_{c\in\mathcal{C}} \sum_{b\in\mathcal{B}_c} x_{c,b} (g_c(b))_r = \frac{1}{t} \sum_{p=1}^{t} \sum_{c\in\mathcal{C}} (a_c^{(p)})_r = \frac{1}{t} \alpha_r.$$

We thus get

$$\lambda \leq \frac{1}{\varepsilon_2 t} \ln \sum_{r\in\mathcal{R}} e^{\varepsilon_2 \alpha_r} \leq \frac{1}{\varepsilon_2 t} \ln \sum_{r\in\mathcal{R}} \psi_r e^{\varepsilon_2 \alpha_r} = \frac{1}{\varepsilon_2 t} \ln\big(\omega^\top \mathbb{1}\big),$$

proving the claim. □

We can now bound the approximation ratio guaranteed by the RESOURCE SHARING ALGORITHM for fixed parameters ε_1, ε_2 and t (and ε_0 implicitly given by the block solvers):

Theorem 3.4. *Let $\Lambda \in \mathbb{R}$ with $g_c(b) \leq \Lambda \mathbb{1}$ for $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$. Let $\varepsilon_0, \varepsilon_1 \geq 0$ and $\varepsilon_2 > 0$. Let $\varepsilon' := \frac{1}{\Lambda}(e^{\varepsilon_2 \Lambda} - 1)(1+\varepsilon_1)(1+\varepsilon_0)$. Assume that $\varepsilon'\lambda^* < 1$, where λ^* is the optimum value of (3.1). Let $t \in \mathbb{N}$ and $\xi := \frac{\varepsilon'}{\varepsilon_2(1-\varepsilon'\lambda^*)} + \frac{\ln \Psi}{\varepsilon_2 t \lambda^*}$. Then the* RESOURCE SHARING ALGORITHM *computes a ξ-approximate solution to (3.1).*

Proof. By construction we have $x_{c,b} \geq 0$ for all $b \in \mathcal{B}_c$ and $\sum_{b \in \mathcal{B}_c} x_{c,b} = 1$. Hence we have a convex combination of vectors in $\mathcal{B}_c$.

We will consider the term $(\omega^{(p)})^\top \mathbb{1}$ for all phases p, where $\omega^{(p)} := \omega^{(p,|\mathcal{C}|)}$ and

$$\omega^{(p,k)} := \left(y_r\left(\alpha_r^{(p,k)}\right)\right)_{r \in \mathcal{R}}$$

for $0 \leq k \leq |\mathcal{C}|$. Initially we have

$$(\omega^{(0)})^\top \mathbb{1} = \Psi. \tag{3.3}$$

We can estimate the increase $\left(\omega^{(p)} - \omega^{(p-1)}\right)^\top \mathbb{1}$ as follows ($1 \leq p \leq t$).

$$\omega_r^{(p,c)} \leq \omega_r^{(p,c-1)} e^{\varepsilon_2 (a_c^{(p)})_r} \leq \omega_r^{(p,c-1)} + \frac{1}{\Lambda}(e^{\varepsilon_2 \Lambda} - 1)\omega_r^{(p,c-1)}(a_c^{(p)})_r \tag{3.4}$$

for each $r \in \mathcal{R}$ because $(a_c)_r^p \leq \Lambda$ for $r \in \mathcal{R}$, $\varepsilon_2 \Lambda \leq 1$, and $e^x \leq 1 + \frac{e^{\varepsilon_2\Lambda}-1}{\varepsilon_2\Lambda} x$ for $0 \leq x \leq \varepsilon_2 \Lambda$. We get

$$(\omega^{(p)})^\top \mathbb{1} \;\leq\; (\omega^{(p-1)})^\top \mathbb{1} + \frac{1}{\Lambda}(e^{\varepsilon_2\Lambda} - 1) \sum_{c \in \mathcal{C}} (\tilde{\omega}^{(p,c-1)})^\top a_c^{(p)}, \tag{3.5}$$

as clearly $\tilde{\omega}^{(p,c-1)} \geq \omega^{(p,c-1)}$.

Next we observe that for each $c \in \mathcal{C}$ we have

$$\left(\tilde{\omega}^{(p,c-1)}\right)^\top a_c^{(p)} \;\leq\; (1+\varepsilon_1)\left(z_c^{(p)} + (1+\varepsilon_0)\left(\left(\tilde{\omega}^{(p,c-1)}\right)^\top l_c - \phi_c^{(p)}\right)\right) \tag{3.6}$$

and

$$z_c^{(p)} + (1+\varepsilon_0)\left(\left(\tilde{\omega}^{(p,c-1)}\right)^\top l_c - \phi_c^{(p)}\right) \;\leq\; (1+\varepsilon_0)\,\mathrm{opt}_c(\tilde{\omega}^{(p,c-1)}). \tag{3.7}$$

Both inequalities are evident if the oracle f_c is called in phase p and

$$z_c^{(p)} = \left(\tilde{\omega}^{(p,c-1)}\right)^\top g_c\left(f_c\left(\tilde{\omega}^{(p,c-1)}\right)\right) = \left(\tilde{\omega}^{(p,c-1)}\right)^\top (a_c)^{(p)}$$

and $\phi_c^{(p)} = \left(\tilde{\omega}^{(p,c-1)}\right)^\top l_c$. Inequality (3.6) also holds in the other case, i.e. if $a_c^{(p)} = a_c^{(p-1)}$, $\phi_c^{(p)} = \phi_c^{(p-1)}$, and $z_c^{(p)} = z_c^{(p-1)}$. Inequality (3.7) continues to hold when ϕ_c and z_c remain constant and the prices $\tilde{\omega}$ increase, because $(l_c)_r \leq a_r$ for each $a \in \mathcal{A}_c$ and $r \in \mathcal{R}$.

By combining (3.5), (3.6), and (3.7), and using that the monotonicity of $\tilde{\omega}$ implies $\mathrm{opt}_c(\tilde{\omega}^{(p,c-1)}) \leq \mathrm{opt}_c(\tilde{\omega}^{(p,|\mathcal{C}|)})$, we obtain:

$$(\omega^{(p)})^\top \mathbb{1} \leq (\omega^{(p-1)})^\top \mathbb{1} + \varepsilon' \sum_{c \in \mathcal{C}} \mathrm{opt}_c(\tilde{\omega}^{(p,|\mathcal{C}|)}), \tag{3.8}$$

where $\varepsilon' := \frac{1}{\Lambda}(e^{\varepsilon_2\Lambda}-1)(1+\varepsilon_1)(1+\varepsilon_0)$.

Now we use that $\tilde{\omega}^{(p,|\mathcal{C}|)} = \omega^{(p,|\mathcal{C}|)}$ because $\alpha^{(p-1)}+L$ clearly is a lower bound on $\alpha^{(p)}$, and get

$$(\omega^{(p)})^\top \mathbb{1} \le (\omega^{(p-1)})^\top \mathbb{1} + \varepsilon' \sum_{c\in\mathcal{C}} \mathrm{opt}_c(\omega^{(p)}). \tag{3.9}$$

By Lemma 3.2

$$\lambda_{lb}^{(p)} := \frac{\sum_{c\in\mathcal{C}} \mathrm{opt}_c(\omega^{(p)})}{(\omega^{(p)})^\top \mathbb{1}} \tag{3.10}$$

is a lower bound on the optimum value λ^* of (3.1). As $\varepsilon'\lambda_{lb}^{(p)} \le \varepsilon'\lambda^* < 1$, inequality (3.9) yields

$$(\omega^{(p)})^\top \mathbb{1} \le \frac{1}{1-\varepsilon'\lambda^*}(\omega^{(p-1)})^\top \mathbb{1}. \tag{3.11}$$

Combining (3.3) and (3.11) we get:

$$(\omega^{(t)})^\top \mathbb{1} \le \frac{\Psi}{(1-\varepsilon'\lambda^*)^t} = \Psi\left(1+\frac{\varepsilon'\lambda^*}{1-\varepsilon'\lambda^*}\right)^t \le \Psi e^{t\varepsilon'\lambda^*/(1-\varepsilon'\lambda^*)}, \tag{3.12}$$

as $1+x \le e^x$ for $x \ge 0$. Lemma 3.3 and (3.12) yield:

$$\sum_{c\in\mathcal{C}} \left(g_c \left(\sum_{b\in\mathcal{B}_c} x_{c,b} b \right) \right)_r \le \frac{1}{\varepsilon_2 t}\left(\ln\Psi + \frac{t\varepsilon'\lambda^*}{(1-\varepsilon'\lambda^*)} \right). \tag{3.13}$$

□

We now show how to set ε_1, ε_2 and t:

Lemma 3.5. *Let $\Lambda \ge 1$ and $\delta, \delta' > 0$. Suppose that we have $g_c(b) \le \Lambda\mathbb{1}$ for all $b \in \mathcal{B}_c$ and $c \in \mathcal{C}$ and $\lambda^* \le 1$. Then we can compute an $(1+\varepsilon_0+\delta+\frac{\delta'}{\lambda^*})$-approximate solution in $O((\delta\delta')^{-1}|\mathcal{C}|\theta\Lambda(1+\varepsilon_0)^2\log\Psi)$ time, where θ is the time for an oracle call.*

Proof. We choose $\varepsilon_1 := \frac{\delta}{2(1+\varepsilon_0)+\delta}$ and $\varepsilon_2 := \frac{\min\{1,\delta/2\}}{3\Lambda(1+\varepsilon_0)^2(1+\varepsilon_1)}$, and $t := \left\lceil \frac{3\Lambda(1+\varepsilon_0)^2(1+\varepsilon_1)\ln\Psi}{\delta'\min\{1,\delta/2\}} \right\rceil$. We have $\varepsilon_2\Lambda \le \frac{1}{3}$, $e^{\frac{1}{3}} < \frac{7}{5}$, and thus $e^{\varepsilon_2\Lambda}-1 \le \varepsilon_2\Lambda\left(1+\frac{3}{5}\varepsilon_2\Lambda\right)$. We get

$$\begin{aligned}
\varepsilon' &= \tfrac{1}{\Lambda}(e^{\varepsilon_2\Lambda}-1)(1+\varepsilon_0)(1+\varepsilon_1) \\
&\le \varepsilon_2\left(1+\tfrac{3}{5}\varepsilon_2\Lambda\right)(1+\varepsilon_0)(1+\varepsilon_1) \\
&\le \tfrac{2\min\{1,\delta/2\}}{5\Lambda(1+\varepsilon_0)} \le \tfrac{2\min\{1,\delta/2\}}{5(1+\varepsilon_0)} \le \tfrac{2}{5}
\end{aligned}$$

and

$$\tfrac{1}{1-\varepsilon'\lambda^*} \le \tfrac{1}{1-\varepsilon'} \le 1+\tfrac{5}{3}\varepsilon' \le 1+\tfrac{\delta}{3(1+\varepsilon_0)}.$$

We obtain an approximation ratio of

$$\begin{aligned}
\xi &\leq \tfrac{\varepsilon'}{\varepsilon_2(1-\varepsilon'\lambda^*)} + \tfrac{\ln\Psi}{\varepsilon_2 t\lambda^*} \\
&\leq (1+\varepsilon_0)(1+\varepsilon_1)\left(1+\tfrac{3}{5}\varepsilon_2\Lambda\right)\left(1+\tfrac{\delta}{3(1+\varepsilon_0)}\right) + \tfrac{\delta'}{\lambda^*} \\
&\leq (1+\varepsilon_1)\left(1+\varepsilon_0+\tfrac{\delta}{10}+\left(1+\tfrac{1}{5}\right)\tfrac{\delta}{3}\right) + \tfrac{\delta'}{\lambda^*} \qquad (3.14)\\
&= (1+\varepsilon_1)\left(1+\varepsilon_0+\tfrac{\delta}{2}\right) + \tfrac{\delta'}{\lambda^*} \\
&= 1+\varepsilon_0+\delta+\tfrac{\delta'}{\lambda^*}.
\end{aligned}$$

The claim follows as $\varepsilon_1 \leq 1$. □

Remark. We can alternatively set $\varepsilon_1 := 0$, $\varepsilon_2 := \frac{\min\{1,\delta\}}{3\Lambda(1+\varepsilon_0)^2}$ and $t := \left\lceil \frac{3\Lambda(1+\varepsilon_0)^2 \ln\Psi}{\delta' \min\{1,\delta\}} \right\rceil$ in the proof of Lemma 3.5, which reduces the number t of phases which have to be performed by the algorithm roughly by a factor of 2 for small values of δ (we do not repeat the proof for this choice of parameters). In practice however allowing $\varepsilon_1 > 0$ offers an enormous potential for runtime reduction, often even more than a factor of 2, because in many cases the last solution computed for a customer can be reused instead of calling the block solver again (i.e. branching into the body of the **if**-statement in the procedure `AllocateResources`). If $\varepsilon_0 = 0$ and $\varepsilon_1 = 0$, such reuse is possible for a customer $c \in \mathcal{C}$ only if the last solution is equal to the vector l_c of lower bounds for c, which is almost never the case in practice. Moreover, allowing $\varepsilon_1 > 0$ is of advantage for the PARALLEL RESOURCE SHARING ALGORITHM which we will present in section 3.3.

Together with an iterative scaling of resource consumption functions we get a fully polynomial approximation scheme (relative to ε_0) for the case that $\frac{\Lambda}{\lambda^*}$ is bounded. More precisely:

Theorem 3.6. *Let $\rho \geq 1$. Suppose that for all $b \in \mathcal{B}_c$ and $c \in \mathcal{C}$ we have $g_c(b) \leq \rho\lambda^*\mathbb{1}$. Assume a $(1+\varepsilon_0)$-optimal block solver for some $\varepsilon_0 \geq 0$. Then a $(1+\varepsilon_0+\varepsilon)$-approximate solution can be computed in*

$$O(|\mathcal{C}|\theta\rho(1+\varepsilon_0)^2 \log\Psi(\log\Psi + \varepsilon^{-2}(1+\varepsilon_0)))$$

time, or

$$O(|\mathcal{C}|\theta\rho(1+\varepsilon_0)^2 \log|\mathcal{R}|(\log|\mathcal{R}| + \varepsilon^{-2}(1+\varepsilon_0)))$$

time if $\sum_{r\in\mathcal{R}} \psi_r \leq |\mathcal{R}|^k$ for a constant value $k \geq 1$, where θ is the time for an oracle call.

Proof. We first compute $f_c(\tilde{\omega}^{(0)})$ for all $c \in \mathcal{C}$ and set

$$\lambda^{ub} := \max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(f_c(\tilde{\omega}^{(0)})))_r.$$

W.l.o.g. $\lambda^{ub} > 0$ since otherwise we already have an optimum solution. Of course $\lambda^* \leq \lambda^{ub}$. Moreover, by Lemma 3.2 we have

$$\begin{aligned}
\lambda^* &\geq \frac{\sum_{c\in\mathcal{C}} \mathrm{opt}_c(\tilde{\omega}^{(0)})}{(\tilde{\omega}^{(0)})^\top \mathbb{1}} \\
&\geq \frac{1}{(1+\varepsilon_0)(\tilde{\omega}^{(0)})^\top \mathbb{1}} \sum_{c\in\mathcal{C}} (\tilde{\omega}^{(0)})^\top g_c(f_c(\tilde{\omega}^{(0)})) \\
&= \frac{1}{\Psi(1+\varepsilon_0)} \sum_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(f_c(\tilde{\omega}^{(0)})))_r \\
&\geq \frac{1}{\Psi(1+\varepsilon_0)} \lambda^{ub}.
\end{aligned}$$

To make sure that the RESOURCE SHARING ALGORITHM is called only on instances for which $\lambda^* \leq 1$ is known (allowing us to apply Lemma 3.5 and Theorem 3.4), we iteratively scale resource consumption functions as follows. For $j = 0, \ldots, \lceil \log_2 \Psi \rceil$ we define a new instance $I^{(j)}$ by setting $g_c^{(j)}(b) := 2^j g_c(b)/\lambda^{ub}$ for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$. Let $\lambda^{*(j)}$ be the optimum of the instance $I^{(j)}$. We start with $j = 0$. We have $\lambda^{*(0)} \leq 1$. We apply Lemma 3.5 with $\delta = \delta' = \frac{1}{4}$ and $\Lambda = \rho$ (note that $g_c^{(j)}(b) = \frac{g_c(b)2^j}{\lambda^{ub}} \leq \frac{\rho\lambda^* 2^j}{\lambda^{ub}} \mathbb{1} \leq \rho\mathbb{1}$). Let $\lambda^{(j)}$ be the corresponding solution value.

If $j < \lceil \log_2 \Psi \rceil$ and $\lambda^{(j)} \leq \frac{1}{2}$, then $\lambda^{*(j+1)} \leq 1$, so we increment j by one and iterate.

If $\lambda^{(j)} > \frac{1}{2}$, then $\lambda^{(j)} \leq (\frac{5}{4} + \varepsilon_0)\lambda^{*(j)} + \frac{1}{4}$. Hence $1 \geq \lambda^{*(j)} \geq \frac{1}{\frac{5}{4}+\varepsilon_0}(\lambda^{(j)} - \frac{1}{4}) \geq \frac{1}{5+4\varepsilon_0}$. If $j = \lceil \log_2 \Psi \rceil$ we have $1 \geq \lambda^{*(j)} \geq \frac{1}{1+\varepsilon_0}$. In both cases $1 \geq \lambda^{*(j)} \geq \frac{1}{5+4\varepsilon_0}$. We then apply Lemma 3.5 once more, now with $\delta := \frac{\varepsilon}{6}$ and $\delta' := \frac{\varepsilon}{6(1+\varepsilon_0)}$. We get an approximation guarantee of $1 + \varepsilon_0 + \delta + \frac{\delta'}{\lambda^*} \leq 1 + \varepsilon_0 + \frac{\varepsilon}{6} + \frac{(5+4\varepsilon_0)\varepsilon}{6(1+\varepsilon_0)} \leq 1 + \varepsilon_0 + \varepsilon$.

The total running time is dominated by applying Lemma 3.5 $O(\log\Psi)$ times with $\delta = \delta' = \frac{1}{4}$ and once with $\delta = \frac{\varepsilon}{6}$ and $\delta' = \frac{\varepsilon}{6(1+\varepsilon_0)}$, and is hence bounded by $O(|\mathcal{C}|\theta\rho(1+\varepsilon_0)^2 \log\Psi(\varepsilon^{-2}(1+\varepsilon_0) + \log\Psi))$. □

Sometimes variants of the algorithm can be useful:

- Splitting up a customer, e.g. replacing $c \in \mathcal{C}$ by two customers c', c'' with $\mathcal{B}_{c'} := \mathcal{B}_{c''} := \mathcal{B}_c$ and $g_{c'} := g_{c''} := \frac{1}{2} g_c$, does not change the problem. This can be useful if $(g_c(b))_r > 1$ for some $b \in \mathcal{B}_c$ and $r \in \mathcal{R}$. It can even be done during the course of the algorithm. Note that the analysis only requires updates of the dual variables before a resource r is used more than Λ. Splitting up a customer also increases the running time by a factor of Λ in the worst case, but in fact only by $\frac{1}{|\mathcal{C}|} \sum_{c\in\mathcal{C}} \max\{1, \max\{(g_c(b))_r \mid r \in \mathcal{R}, b \in \mathcal{B}_c\}\}$. In VLSI routing, for example, there are usually only very few customers $c \in \mathcal{C}$ with $\max\{(g_c(b))_r \mid r \in \mathcal{R}, b \in \mathcal{B}_c\} > 1$, so splitting these up in the described way is a viable approach to the fractional problem. It must be noted however that this can increase the integrality gap.

- Parallelize (performing the if-statements of c, c' in parallel): Replacing c, c' by one customer $\bar{c}$ and setting $\mathcal{A}_{\bar{c}} := \mathcal{A}_c \times \mathcal{A}_{c'}$ and $f_{\bar{c}} := f_c \times f_{c'}$ does not change the problem. Note, however, that ρ may increase. Note that this variant updates the cost function only once for c, c'. Uniting all customers thus amounts to updating costs only once per phase.

Remark. Even without an increase of ρ, this parallelization approach in practice will not result in good speedups in most cases as the time needed for an oracle call often is very different for each customer. We present a parallel algorithm for shared-memory machines in section 3.3, which turns out to be very efficient in practice.

We finally show that if we have a bounded (weak) block solver, we can get rid of ρ in the running time bound of Theorem 3.6:

Theorem 3.7. *Suppose that we have $(1+\varepsilon_0)$-optimal bounded block solvers for some $\varepsilon_0 \geq 0$. Then a $(1+\varepsilon_0+\varepsilon)$-approximate solution can be computed in*

$$O(|\mathcal{C}|\theta(1+\varepsilon_0)^2 \log\Psi(\log\Psi + \varepsilon^{-2}(1+\varepsilon_0)))$$

time, or

$$O(|\mathcal{C}|\theta(1+\varepsilon_0)^2 \log|\mathcal{R}|(\log|\mathcal{R}| + \varepsilon^{-2}(1+\varepsilon_0)))$$

time if $\sum_{r\in\mathcal{R}} \psi_r \leq |\mathcal{R}|^k$ for a constant value $k \geq 1$, where θ is the time for an oracle call.

Proof. Note that the algorithm is called only for instances where we know $\lambda^* \leq 1$. Hence we can restrict $\mathcal{B}_c$ to $\mathcal{B}'_c := \{b \in \mathcal{B}_c \mid g_c(b) \leq \frac{\lambda^{ub}}{2^j}\mathbb{1}\}$ in iteration j (in the proof of Theorem 3.6) without changing the optimum. This means that we can choose $\Lambda = 1$ in Lemma 3.5. With the bounded block solver we can optimize over $\mathcal{B}'_c$. □

3.2.2 An Example Attaining the Worst Case Runtime

The worst-case runtime obtained in the analysis of the RESOURCE SHARING ALGORITHM provides a tight bound w.r.t. the approximation parameter ε. It is easy to see this by looking at the following instance of the RESOURCE SHARING PROBLEM which requires $\Theta(\varepsilon^{-2})$ phases to compute a $(1+\varepsilon_0+\varepsilon)$-optimal solution to (3.1): Let $0 < \varepsilon \leq 6$ and $k \in \mathbb{N}$, $k \geq 2$, such that $\frac{18 \ln k}{\varepsilon} \notin \mathbb{N}$, and consider the instance with

- $\mathcal{C} = \{c\}$
- $\mathcal{R} = \{r_1, \ldots, r_{k+1}\}$
- $\mathcal{B}_c = \mathrm{conv}(\{b_1, b_2\})$ with
  - i) $g_c(b_1) := (1, 0, \ldots, 0)$
  - ii) $g_c(b_2) := (0, 1, \ldots, 1)$
  - iii) $g_c(ab_1 + (1-a)b_2) := ag_c(b_1) + (1-a)g_c(b_2)$ for $0 \leq a \leq 1$

For simplicity, let $l_c :\equiv 0$ and $\psi_r := 1$ for each $r \in \mathcal{R}$. In an *optimum solution* clearly $x_{c,b_1} = x_{c,b_2}$, i.e. $\lambda^* = 0.5$. We have $\Lambda = 1$. Now assume $\varepsilon_0 = 0$, i.e. we have a strong block solver. In the final iteration of the binary search procedure in the proof of Theorem 3.6, we start the RESOURCE SHARING ALGORITHM with parameters $\varepsilon_1 := 0$ and $\varepsilon_2 := \frac{\min\{1,\varepsilon/6\}}{3\Lambda(1+\varepsilon_0)^2} = \frac{\varepsilon}{18}$ in order to obtain an $(1+\varepsilon)$-optimal solution (for simpler calculation we use the alternative parameters mentioned after the proof of Lemma 3.5). In the first $\lceil \frac{\ln k}{\varepsilon_2} \rceil = \lceil \frac{18 \ln k}{\varepsilon} \rceil$ phases b_1 is the cheapest element in $\mathcal{B}_c$, so $x_{c,b} = 0$ for all $b \in \mathcal{B}_c \setminus \{b_1\}$. After the $\lceil \frac{18 \ln k}{\varepsilon} \rceil$-th phase we have $\omega^\top b_2 < \omega^\top b_1 \leq e^{\varepsilon_2} \omega^\top b_2$, so starting with this iteration the algorithm alternatingly picks b_1 and b_2. Therefore $\Omega(\varepsilon^{-2})$ phases are needed until $x_{c,b_1} \leq (1+2\varepsilon) x_{c,b_2}$, which is a necessary condition for the solution to be $(1+\varepsilon)$-optimal.

It is important however to note that this example *does not preclude* existence of an algorithm with a runtime of $\tilde{O}(\varepsilon^{-k}(|\mathcal{C}| + |\mathcal{R}|)\theta)$ with $k < 2$. Indeed, as mentioned in section 3.1, Bienstock and Iyengar [2006] showed that a runtime proportional to ε^{-1} is possible for the fractional packing problem (i.e. all g_c, $c \in \mathcal{C}$ restricted to linear functions), although, as they rely on quadratic programming, their runtime is not bounded by a linear number (w.r.t. $|\mathcal{C}| + |\mathcal{R}|$) of calls to a block solver. Klein and Young [1999], on the other hand, showed that there are instances of the fractional packing problem which require at least $\Omega(\varepsilon^{-2})$ calls to a block solver by any algorithm that accesses the sets $\mathcal{B}_c$ ($c \in \mathcal{C}$) only indirectly by calling a block solver. However, their proof is restricted to instances with $\rho\varepsilon^{-2} = O\left(\sqrt{|\mathcal{R}|}\right)$, so they do not make a statement for an arbitrarily small $\varepsilon > 0$ with fixed $|\mathcal{R}|$ and ρ.

3.3 Parallelization

In this section we present a lock-free and non-blocking parallel resource sharing algorithm for shared-memory machines. Unlike in distributed algorithms running on independent machines and using message passing protocols for synchronization, in the shared-memory model all processors have direct access to the same physical memory.

The conventional method for synchronizing concurrent access to shared memory locations is *mutual exclusion*: All so-called *critical sections* of the program code that potentially change the content of a memory location (or a set of memory locations) not privately owned by one thread first has to *acquire* or *lock* a token, called *mutex*, in order to be allowed to do so. The mutex is shared among all threads and can be locked by at most one of them at a time. Therefore if a thread tries to acquire a mutex which is locked by another thread, it has to wait until the mutex is *released* or *unlocked* again. This imposes several problems:

- **Preemption:** If a thread is delayed and holds a lock that other threads are waiting for, these threads are delayed also. If they in turn hold locks, transitively even more threads are blocked. This is termed *lock convoying* in literature. Reasons for such delays include e.g. scheduling by the operating system or page faults (i.e. swapping).

- **Priority Inversion:** If threads have different priorities, and a low priority thread currently holding a lock is preempted, threads with higher priority are stalled.

- **Deadlocks:** Implementations of parallel algorithms based on mutual exclusion are susceptible to deadlocking, i.e. a circular dependency between threads trying to lock mutexes currently owned by other threads. Much care has to be taken to guarantee that such situations cannot occur.

Of course these issues are of different importance in different areas. In our case we are concerned mainly about performance degradation due to preemption of threads (presuming that a proof of impossibility of deadlocks in our case would be possible). In contrast, so-called real-time systems that guarantee response times for completion of certain operations preclude use of mutual exclusion for synchronization.

In addition to the preemption problem, there are more reasons why synchronization by mutual exclusion can cost performance: Standard mutex locking implementations as e.g. provided by the *Pthreads library* (cf. Nichols, Buttlar and Proulx Farrell [1996]) incur a large overhead for operating system calls and, if trying to lock an already locked mutex, for suspending and resuming thread execution. If the code section protected by the mutex takes very short time to complete, e.g. if just some shared counter variable is incremented, this overhead is enormous.

In such cases, so-called *spin locks* improve performance drastically: A *spin mutex* consists of a binary variable whose value is, say, 0 if it is not locked, and 1 otherwise. A thread that wants to lock a spin mutex basically loops (*spins*) until the variable is zero and then sets it to 1 to declare that it acquired the lock. To avoid that two threads concurrently see a value of zero and set it to 1 (both thinking they acquired the lock), practically all modern microprocessors provide machine instructions for atomically, i.e. uninterruptably, test if a variable contains a certain value (0 in this case) and, if the answer is positive, set it to a new value. For example, Intel and AMD x86 processors provide such a *compare-and-swap* (CAS) instruction. Other architectures, e.g. PowerPC, provide a stronger (also non-blocking) mechanism by a pair of *load-link* and *store-conditional* (LL/SC) instructions. We will explain CAS and LL/SC instructions in more detail below. Acquiring a spin mutex is by far cheaper than an operating system call if the mutex is not currently being held by another thread. If however a thread has to wait for the lock to be released, it consumes CPU time because it performs a loop until the mutex is unlocked. This is called *busy waiting* and is a disadvantage over operating-system based mutex locking (which suspends the thread in this case) if the time spent for busy waiting exceeds the operating system overhead. Using spin locks therefore can be regarded as an optimistic locking approach.

However, even spin locks impose a performance penalty, namely for the following two reasons:

i) There are additional variables (the spin mutexes) which have to be read from and written to. This can degrade performance if many resources (variables) shall be lockable independently from each other – in this case a new variable has to be introduced for each variable to be locked.

ii) Mutual exclusion constrains ordering of memory accesses. Because processors become increasingly faster compared to system memory, performance can be drastically improved by reordering memory accesses. Guaranteeing mutual exclusion, i.e. execution of instructions in a critical code section by at most one processor at a time, means that all effects of instructions performed by any other processor in a critical section protected by the same mutex must become visible (i.e. travel through the cache hierarchy) before a read from memory can be allowed. Unless this point is reached, execution of the corresponding instructions cannot be regarded as finished by all processors. This means that a so-called *memory barrier* must be introduced which prohibits reordering of memory access operations across it, and delays until memory operations of other threads become visible. We go into more detail in section 3.3.1

Because of the reasons mentioned above, *lock-free* algorithms and data structures are of considerable interest. Following the notion of Herlihy and Moss [1993], a lock-free algorithm or data structure is characterized by not requiring mutual exclusion. If a lock-free algorithm at each point during its execution guarantees that within a certain bounded time at least one thread among those that are not blocked by external influences (e.g. scheduling by the operating system) will make *progress*, i.e. perform an operation that would be necessary also in sequential processing and not only for serving synchronization among threads, it is called *non-blocking* (see e.g. Herlihy [1990] or Sundell [2004]). Assuming such external influences, algorithms based on mutual exclusion cannot be guaranteed to be non-blocking. Going one step further, *wait-free* algorithms at any point in time guarantee that *each* thread not stalled by external influences will make progress within a certain bounded time. While there are theoretical universality results (see e.g. Herlihy [1991]) showing that from each sequential algorithm a wait-free algorithm on an arbitrary number of processors can be constructed provided that sufficiently powerful instructions as e.g. CAS are available, developing practical wait-free algorithms is considerably harder than non-blocking algorithms. We do not go into details here and refer to Sundell [2004] for an overview.

Boehm [2005] shows a comparison of two parallel implementations of the Sieve of Eratosthenes, one based on mutex locks provided by the Pthreads library, and another using spin locks. As expected, the second version performs much better. In addition, a lock-free implementation is compared, showing even better performance than if using spin locks, and indeed almost optimum speedups.

There exist many lock-free algorithms and data structures for elementary tasks. For example, Sundell and Tsigas [2005] show how lock-free concurrent linked lists can be built upon the CAS primitive. In contrast to an earlier work by Valois [1995], they also support concurrent deletions of list items. Michael [2004] presents a scalable lock-free approach to concurrent dynamic memory allocation. There are also general frameworks for object-based transactional memory (Fraser and Harris [2007]) that allow atomic access to arbitrary objects without resorting to mutual exclusion. See also Sundell [2004] for an overview.

3.3.1 The Memory Model

Shared-memory computers are becoming more and more standard, with an increasing number of processors jointly using the same memory. This allows very fast communication among the processors compared to distributed computing where distinct machines communicate over a network. Developing a parallelized algorithm of course involves some synchronization protocol, and for shared-memory machines this requires a *memory model* which specifies certain guarantees that can be relied upon, or if not, can be enforced by using a special kind of synchronization instructions. This is particularly important as with multi-threaded programs, many intuitively correct assumptions on the ordering in which memory accesses performed by different processors are observed by other processors turn out to be invalid on modern architectures because of aggressive hardware optimizations.

Therefore without a memory model specification, it is not possible to reason about the behaviour of programs executing more than one thread concurrently on a shared-memory machine. Indeed, the current C++ language standard as e.g. described by Stroustrup [2000] does not define semantics of multi-threaded programs. However, many compilers provide high-level (machine independent) access to machine dependent instructions that can be used to enforce certain ordering restrictions on memory access and thus, if used correctly, provide sufficient guarantees in order to make a reasoning about the semantics of a shared-memory-parallel algorithm possible. The designated next C++ standard, commonly termed *C++0x* (see C++ Standards Committee [2009] for the current draft) will define a memory model, specifying some rather weak guarantees and providing a set of primitives that can be used to enforce stronger guarantees where needed. Boehm and Adve [2009] give a comprehensive introduction.

We will not define a complete memory model in this section, but provide enough background to describe the effects of certain synchronization operations on memory and the guarantees they provide, which we will use in the correctness proof of our algorithm in the next section. The assumptions that we will make correspond to or are weaker than the specifications of current microprocessors (see e.g. AMD64 Architecture Programmer's Manual [2007], Intel®64 and IA-32 Architectures Software Developer's Manual [2009], Power Instruction Set Archtitecture [2009] or SPARC Architecture Manual [2000]). The synchronization operations that we define correspond directly to machine instructions available on practically all current processors, and are similar to the synchronization primitives that will become part of the next C++ standard.

We will make a few simplifications for easier presentation which — for our purposes — do not affect validity of the model. First we assume that memory is partitioned into equally-sized cells or *locations*, each of which stores a variable that is read atomically, meaning that its content does not change while being read. If each variable is also atomically written (i.e. each write operation has exclusive access to the corresponding memory location while changing its contents), a read operation thus will always read a value that was the result of some write operation, and not a mixture of two or more write operations. We will need atomical write operations only for a special class of *synchronizing* write instructions which we will define below. However, because our algorithm will not use non-synchronizing write instructions on variables that are concurrently accessed by different threads, we can assume atomicity of write operations in general for simplifying presentation.

Remark. These assumptions are justified in practice by the specifications of current microprocessors for variables of machine word size that are aligned in memory at addresses which are integral multiples of the machine word size. All bits of such variables are read or written in a single step. If not told otherwise by using special commands, compilers usually adhere to this memory layout for efficiency reasons: An access to an aligned variable requires only one — atomic — memory access, whereas an access to an unaligned variable is performed by two or more accesses to aligned locations, which together appear as a non-atomic (i.e. interruptible) access. In our algorithm, we will use only variables of machine word size. Each memory location in our setting therefore corresponds to a machine word.

We distinguish between instructions and operations, which correspond to instruction executions by some processor. In the following, let $m \in \mathbb{N}$ be the number of memory locations.

Definition 3.8. *We define $\mathcal{I}_i^r$ as the set of instructions that read from memory location i, and $\mathcal{I}_i^w$ as the set of instructions that try to write to memory location i, $1 \leq i \leq m$.*

We write $\mathcal{I}^r := \cup_{1\leq i\leq m}\mathcal{I}_i^r$ and $\mathcal{I}^w := \cup_{1\leq i\leq m}\mathcal{I}_i^w$, and define $\mathcal{I}^x$ as the set of all other instructions.

Note that $\mathcal{I}^r \cap \mathcal{I}^w \neq \emptyset$ (see below). A *program text* or *program* is a sequence $(I_1,\ldots,I_n)$ of instructions ($n \in \mathbb{N}$). When *executed*, each instruction besides its primary purpose determines a next instruction to be executed. We assume for simplicity that execution starts with I_1 and call the resulting sequence $(I_{p_1},\ldots,I_{p_k})$ a *thread*, $k \in \mathbb{N}$, $p_1 = 1$ and $p_2,\ldots,p_k \in \{1,\ldots,n\}$.

Remark. In practice, $p_{j+1} = p_j + 1$ unless I_{p_j} is a special *jumping* instruction used for implementing loops, conditional branchings or function calls ($1 \leq j < k$).

For simplicity we assume Π threads running in parallel on Π processors throughout the execution of a program, for some $\Pi \in \mathbb{N}$, and each thread is bound to a processor whose number $1 \leq \pi \leq \Pi$ initially is stored in one of its registers.

Even for identical input, the instruction sequences defining each thread can be different in each execution of a program if $\Pi > 1$ (if $\Pi = 1$, this is normally not the case). The reason for this is that in a multi-threaded program execution, a thread may read from memory locations other threads write to, so it can depend on relative thread execution speeds which values are read. Each instruction in a thread *performs an operation*:

Definition 3.9. *Let P be a program and $\Pi \in \mathbb{N}$ a number of processors. An* execution *of P on Π processors is defined by its threads*

$$(I_1^\pi, \ldots, I_{k_\pi}^\pi) \in (\mathcal{I}^r \cup \mathcal{I}^x \cup \mathcal{I}^w)^{k_\pi}$$

concurrently executing a set of operations

$$\mathcal{O} := \bigcup_{1 \le \pi \le \Pi} \left\{ (I_1^\pi, \pi, 1), \ldots, (I_{k_\pi}^\pi, \pi, k_\pi) \right\},$$

and a reads-from mapping

$$\mathrm{rf} : \{(I, \pi, x) \in \mathcal{O} : I \in \mathcal{I}^r\} \to \{(I, \pi, x) \in \mathcal{O} : I \in \mathcal{I}^w\}$$

defining for each read operation $o_r \in \mathcal{O}$ a unique $o_w \in \mathcal{O}$ that writes to the same memory location and whose output is read from o_r.

Remark. Note that by the above definition we preclude programs that can have executions in which a value is read from an uninitialized memory cell.

In the following, let $(\mathcal{O}, rf)$ be a program execution. We define $\mathcal{O}_i^w \in \mathcal{O}$ as the set of all write operations and $\mathcal{O}_i^r \in \mathcal{O}$ the set of all read operations on memory location i performed in $(\mathcal{O}, rf)$, $1 \le i \le m$, and we have $\mathcal{O}^w := \cup_{1 \le i \le m} \mathcal{O}_i^w$ and $\mathcal{O}^r := \cup_{1 \le i \le m} \mathcal{O}_i^r$. Finally, $\mathcal{O}^x := \mathcal{O} \setminus \{\mathcal{O}^r \cup \mathcal{O}^w\}$ is the set of all other operations.

With a few important exceptions, the operations in $\mathcal{O}^x$ will not be relevant in our memory model considerations. They include e.g. arithmetic operations, i.e. all operations that do the actual work performed by an algorithm, or operations that move data among a set of processor registers privately owned by the same thread. We assume that memory is the only means of communication between threads, i.e. without using read or write operations, no operation $o \in \mathcal{O}^x$ has an effect observable by any other thread (except *fence* and *barrier* operations defined below). In particular, this means that we disallow operations that move data between processor registers owned by different threads running on the same processor.

Remark. Besides moving data between processor registers, different threads can also influence each other by *interrupt* instructions, not necessarily writing to memory. However, we will not use such instructions.

Instruction Reordering and Pipelining

Modern processors actually do not execute the instructions of a thread sequentially. A processor usually has several units for performing arithmetic operations, instruction decoding or memory operations, respectively, which are used in parallel. Further, *pipelining* leads to temporally overlapping instruction executions. However, the semantics of instructions are specified sequentially, i.e. within a thread instructions appear to be executed sequentially in an order conforming to the specification of each instruction which, as mentioned above, based on its input determines a *next* instruction to be executed (we do not go into details here as this would go beyond the scope of this work). The instruction sequence of each thread corresponds to this *program order* $\prec_{\text{prog}}$:

Definition 3.10. *For* $o_1 := (I_1, \pi, x_1), o_2 := (I_2, \pi, x_2) \in \mathcal{O}$ *with* $x_1 < x_2$ *we write* $o_1 \prec_{prog} o_2$ *and say that* o_1 *precedes* o_2 in program order.

The *partial execution order* $\prec_{\text{exec}}$ defines when operations performed in the same thread are actually executed relatively to each other:

Definition 3.11. *For* $o_1, o_2 \in \mathcal{O}$ *performed in the same thread (i.e.* $o_1 \prec_{prog} o_2$ *or* $o_2 \prec_{prog} o_1$*), we define the* partial execution order $\prec_{exec}$ *by* $o_1 \prec_{exec} o_2$ *iff execution of* o_1 *finishes before execution of* o_2 *starts.*

The processor guarantees that, if executed w.r.t. the partial order $\prec_{\text{exec}}$, all operations perform identically as if executed in program order $\prec_{\text{prog}}$. In a simplified view, if $o_1 \prec_{\text{prog}} o_2$ and o_2 depends on the outcome of o_1, o_2 is guaranteed to be executed after o_1. For example, for $r \in \mathcal{O}_i^r$ and $w \in \mathcal{O}_i^w$ for some $1 \le i \le m$, this enforces

$$w \prec_{\text{prog}} r \Leftrightarrow w \prec_{\text{exec}} r. \tag{3.15}$$

Similarly, of course

$$r \prec_{\text{prog}} w \Leftrightarrow r \prec_{\text{exec}} w \tag{3.16}$$

is guaranteed for $r \in \mathcal{O}_i^r$ and $w \in \mathcal{O}_i^w$.

Remark. This is a simplification for two reasons: Even if o_2 depends on the outcome of o_1,

i) there are cases in which the outcome of o_1 is known already before o_1 finishes (e.g. if o_1 writes to a memory location which o_2 subsequently reads from, o_2 can start already before the content of the memory cell is actually overwritten), and

ii) o_2 might be *speculatively* executed before o_1 ($o_2 \prec_{\text{exec}} o_1$) or concurrently with o_1 ($o_1 \not\prec_{\text{exec}} o_2$ and $o_2 \not\prec_{\text{exec}} o_1$), guessing a certain outcome of o_1. If o_1 produces a different outcome, either o_2 is executed again, or — in case of a branch misprediction — the outcome of o_2 is discarded and program execution continues in a different branch.

For the sake of simplicity, we do not go into further details. See Hennessy and Patterson [2006] for a thorough introduction to modern processor architectures.

The important consequence is that the program order $\prec_{\text{prog}}$ of operations can be assumed only from the point of view of the respective thread, and some intuitively correct deductions are invalid for multi-threaded programs. For example, while the condition that

$$rf(r_1) \neq w_2 \text{ for all } r_1, r_2 \in \mathcal{O}^r, w_1, w_2 \in \mathcal{O}^w : r_1 \prec_{\text{prog}} w_1, r_2 \prec_{\text{prog}} w_2, rf(r_2) = w_1 \tag{3.17}$$

is satisfied by (3.15) and (3.16) if all four operations are performed in the same thread, it is not necessarily true in general: Assume that r_i, w_i in (3.17) are executed in thread i, respectively ($i \in \{1,2\}$). Now it is possible that $w_1 \prec_{\text{exec}} r_1$ and $w_2 \prec_{\text{exec}} r_2$, so $rf(r_1) \neq w_2$ becomes possible. This example demonstrates that in multi-threaded programs a "happens-before"-relation defined by $\prec_{\text{prog}}$ and the reads-from mapping rf can contain cycles or contradictions because reading from and writing to a memory cell concurrently by different threads adds dependencies that are not considered by a processor when reordering instructions.

In order to make it possible to write correct multi-threaded programs despite instruction reordering, so-called *fence* instructions $\mathcal{I}^f \subseteq \mathcal{I}^x$ are provided for enforcing program order where necessary:

Definition 3.12. *Let* $\mathcal{O}^f := \{(I, \pi, x) \in \mathcal{O} : I \in \mathcal{I}^f\}$ *be the set of* fence operations *in the program execution* $(\mathcal{O}, rf)$. *Then*

$$o_1 \prec_{prog} f \prec_{prog} o_2 \Rightarrow o_1 \prec_{exec} f \prec_{exec} o_2$$

for all $o_1, f, o_2 \in \mathcal{O}$.

Of course executing a fence instruction means a performance penalty as fences limit concurrency within the same thread, so they should not be used more often than necessary for guaranteeing correctness. For example, as each locking and unlocking operation in mutual exclusion based parallelization approaches involves a fence to guarantee that operations working on a shared variable are not reordered out of the critical section, much performance can be lost with fine-grained locking.

Remark. Aside from the processor, also compilers that generate machine code from source code written in a high-level language like C++ can order instructions in the resulting machine code in a way that does not correspond to the original source code. If the language does not define a memory model for multi-threaded programs, equivalence can be guaranteed only for single-threaded programs (and indeed is not guaranteed by current C++ compilers for multi-threaded programs). However, most C++ compilers provide some means to suppress compiler reordering across a certain point in the source code.

Caching and Non-Uniform Memory Architectures

In order to mitigate the increasing gap between processor speed and main memory speed, increasingly deep cache hierarchies are used. A cache is a fast buffer memory that speeds up access to memory locations that have already been queried earlier, or are located close to memory locations already queried. If the processor performs a read

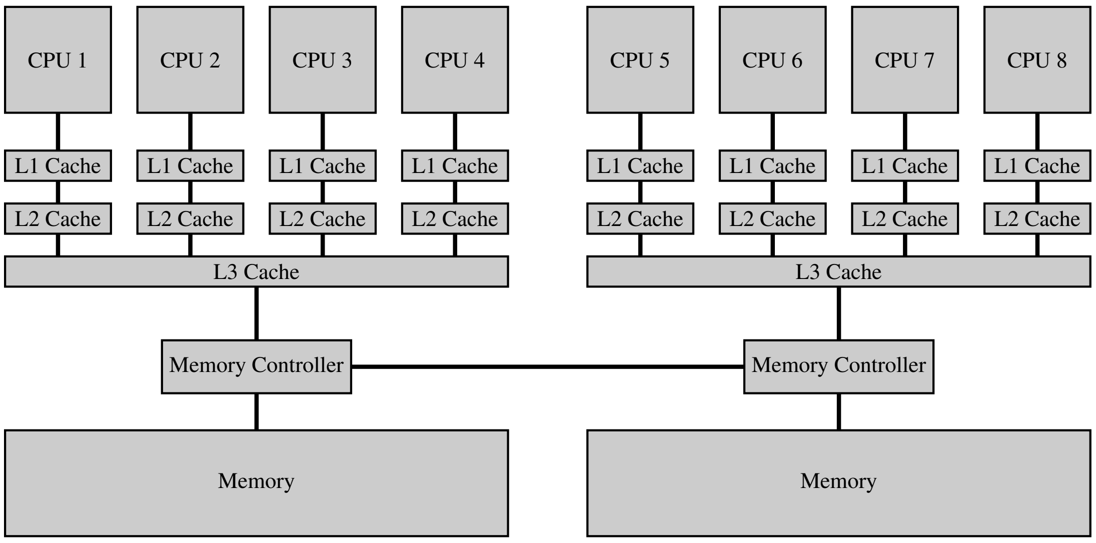


Figure 3.1: Non-uniform memory architecture (NUMA) with a typical cache hierarchy

from a memory location that has not been loaded to the cache yet, a whole *cache line* (e.g. 64 bytes on current Intel x86 processors) is loaded from the next cache level closer to memory, or from memory itself. Three cache levels are common in today's systems, and there are even systems with four cache levels as intermediary between processor and main memory, with caches becoming smaller but faster the closer they are to the processor in the hierarchy. We do not go further into details here, see e.g. Hennessy and Patterson [2006]. Caches are often *shared* between subsets of processors. Additionally, in so-called *non-uniform memory architectures*, memory is partitioned, and each part bound to a group of processors (see figure 3.1 for an illustration). Although all processors may access all memory locations, accesses to different memory parts may take a different amount of time. Because of this, and the cache sharing mentioned above, requiring write operations to be observable by all processors in the same order can mean a considerable performance penalty.

We say that a memory model provides *local view order consistency* if it guarantees that in any program execution $(\mathcal{O}, rf)$, write operations to the same memory location $1 \leq i \leq m$ cannot be observed in different orders on different processors, i.e.

$$\begin{aligned} \nexists r_1, r_2, r_1', r_2' \in \mathcal{O}_i^r \quad : \quad & r_1 \prec_{\text{exec}} r_2, r_1' \prec_{\text{exec}} r_2', rf(r_1) \neq rf(r_2), \\ & rf(r_1) = rf(r_2') \text{ and } rf(r_2) = rf(r_1'). \end{aligned} \tag{3.18}$$

Remark. AMD and Intel x86 processors provide a considerably stronger memory model than other architectures. In contrast e.g. to the PowerPC architecture they guarantee condition (3.18) (see e.g. AMD64 Architecture Programmer's Manual [2007], Intel®64 and IA-32 Architectures Software Developer's Manual [2009] and Owens, Sarkar and Sewell [2009]).

We cannot rely on local view order consistency in general, but we will enforce a slightly weaker form of it in the algorithm which we present in section 3.3.2 by using certain *synchronizing* write instructions (defined below) for writing to variables that are concurrently accessed by different threads. Assuming thus that local view order consistency holds, let $\prec_{\text{cell}}$ be the smallest ordering relation on $\mathcal{O}^w$ which for each memory location contains a total order of write operations to this location and

$$rf(r_1) \prec_{\text{cell}} rf(r_2) \text{ or } rf(r_1) = rf(r_2) \text{ for any } r_1, r_2 \in \mathcal{O}^r \text{ with } r_1 \prec_{\text{exec}} r_2, \quad (3.19)$$

i.e. write operations on different memory locations are not ordered w.r.t. each other by $\prec_{\text{cell}}$. For convenience we will write $w_1 \preceq_{\text{cell}} w_2$ in the following iff $w_1 = w_2$ or $w_1 \prec_{\text{cell}} w_2$.

The stronger condition of *global view order consistency* requires that local view order consistency is guaranteed, and additionally

$$\begin{aligned} \nexists r_1, r_2, r'_1, r'_2 \in \mathcal{O}^r \quad : \quad & r_1 \prec_{\text{exec}} r_2, r'_1 \prec_{\text{exec}} r'_2, \\ & rf(r_2) \prec_{\text{cell}} rf(r'_1) \text{ and } rf(r'_2) \prec_{\text{cell}} rf(r_1) \end{aligned} \quad (3.20)$$

holds for any program execution $(\mathcal{O}, rf)$, i.e. no pair of write operations to the same *or different* memory cells can be observed in different orders by any processors. This condition is not even guaranteed by the x86 architecture.

Remark. Intel x86 processors, in contrast to the AMD x86 architecture, guarantee a restricted version of (3.20) for read and write operations that are performed on different processors (see Owens, Sarkar and Sewell [2009] and section 8.2.3.7 of Intel®64 and IA-32 Architectures Software Developer's Manual [2009]). However, both architectures do not guarantee this condition in general, as they allow an early read of a value written to memory by the same processor, before it becomes visible to other processors (called *Intra-Processor Forwarding* in Intel®64 and IA-32 Architectures Software Developer's Manual [2009]).

We now define *synchronizing write instructions* (denoted $\mathcal{I}^{wsync} \subseteq \mathcal{I}^w$) and *synchronizing read instructions* (denoted $\mathcal{I}^{rsync} \subseteq \mathcal{I}^w$) in terms of the guarantees they provide:

Definition 3.13. *Let $\mathcal{O}^{wsync} \subseteq \mathcal{O}^w$ be the set of write operations in a program execution $(\mathcal{O}, \mathrm{rf})$ that perform a synchronizing write instruction, and analogously $\mathcal{O}^{rsync} \subseteq \mathcal{O}^r$ the set of synchronizing read operations.*

Let $1 \leq i \leq m$. If $\mathcal{O}^w_i \subseteq \mathcal{O}^{wsync}$, then

$$\begin{aligned} \nexists r_1, r_2, r'_1, r'_2 \in \mathcal{O}^r_i \cap \mathcal{O}^{rsync} \quad : \quad & r_1 \prec_{exec} r_2, r'_1 \prec_{exec} r'_2, \mathrm{rf}(r_1) \neq \mathrm{rf}(r_2), \\ & \mathrm{rf}(r_1) = \mathrm{rf}(r'_2) \textit{ and } \mathrm{rf}(r_2) = \mathrm{rf}(r'_1). \end{aligned} \quad (3.21)$$

Note that the difference between (3.18) and (3.21) is that the latter guarantees local view order consistency only for synchronizing read operations. So even if all write operations on a memory location are synchronizing, non-synchronizing read operations on this location performed by different processors might still observe different orders. We will only use synchronizing write instructions for variables accessed by different threads, so we do not require (3.18), but can rely on (3.21) and therefore from here on restrict (3.19) to $r_1, r_2 \in \mathcal{O}^{rsync}$ in the definition of $\prec_{\text{cell}}$ for memory locations accessed by more than one thread.

Remark. Synchronizing read and write instructions are often referred to as *atomic instructions*. We prefer the term "synchronizing" to distinguish between the synchronization aspect (i.e. an ordering agreement between processors) and atomicity in the sense that a read operation always reads the value written by some write operation, and never a mixture of bits written by two different write operations.

In order to enable the programmer to enforce certain visibility conditions where needed, current processors offer so-called *barrier* instructions which impose an ordering of memory access operations on either side of a barrier operation w.r.t. program order $\prec_{\text{prog}}$. There are two types of barrier instructions, *acquire barriers* (denoted $\mathcal{I}^{acq} \subseteq \mathcal{I}^{x}$) and *release barriers* (denoted $\mathcal{I}^{rel} \subseteq \mathcal{I}^{x}$), which provide the following guarantees (again assuming that only synchronizing write operations are used on memory locations accessed by different threads):

Definition 3.14. *Let* $\mathcal{O}^{acq} := \{(I,\pi,x) \in \mathcal{O} : I \in \mathcal{I}^{acq}\}$ *be the set of* acquire barrier operations *in a program execution* $(\mathcal{O}, rf)$*, and let* $1 \leq i \leq j \leq m$*,* $r_1 \in \mathcal{O}_j^{rsync}$*,* $r_2 \in \mathcal{O}_i^{r}$*,* $w_1 \in \mathcal{O}_i^{wsync}$ *and* $w_2 \in \mathcal{O}_j^{wsync}$*. If* $r_1 \prec_{exec} r_2$ *and* $w_1 \prec_{prog} b \prec_{prog} w_2$ *for some* $b \in \mathcal{O}^{acq}$*, then*

$$w_2 \preceq_{cell} \mathrm{rf}(r_1) \Rightarrow w_1 \preceq_{cell} \mathrm{rf}(r_2) \tag{3.22}$$

is guaranteed to hold.

I.e., if a synchronizing read operation r_1 performed on any processor reads the value written by some write operation w_2 (or a newer value w.r.t. $\prec_{\text{cell}}$), any later read operation (w.r.t. the execution order $\prec_{\text{exec}}$) must see the value (or a newer value w.r.t. $\prec_{\text{cell}}$) written by an operation w_1 that "happens before" w_2 w.r.t. the ordering constraint introduced by the barrier.

Similarly, release barriers provide the following guarantee:

Definition 3.15. *Let* $\mathcal{O}^{rel} := \{(I,\pi,x) \in \mathcal{O} : I \in \mathcal{I}^{rel}\}$ *be the set of* release barrier operations *in a program execution* $(\mathcal{O}, rf)$*, and let* $1 \leq i \leq j \leq m$*,* $r_1 \in \mathcal{O}_j^{r}$*,* $r_2 \in \mathcal{O}_i^{rsync}$*,* $w_1 \in \mathcal{O}_i^{wsync}$ *and* $w_2 \in \mathcal{O}_j^{wsync}$*. If* $r_1 \prec_{exec} r_2$ *and* $w_1 \prec_{prog} b \prec_{prog} w_2$ *for some* $b \in \mathcal{O}^{rel}$*, then*

$$\mathrm{rf}(r_2) \prec_{cell} w_1 \Rightarrow \mathrm{rf}(r_1) \prec_{cell} w_2 \tag{3.23}$$

is guaranteed to hold.

Note that these guarantees are provided also for non-synchronizing read operations on the right-hand side of (3.22) and (3.23), respectively. We shall see that this will allow us to restrict the use of synchronizing read operations, which may be more expensive than normal read operations on some architectures, to a small subset of variable accesses.

Remark. The commonly used names *acquire barrier* and *release barrier* match the fact that a barrier of the corresponding type is necessary in mutual exclusion based algorithms when *acquiring* or *releasing* a lock, for the reasons sketched at the beginning of section 3.3. Barriers are also called *memory fences* sometimes. Most processors provide also *full barrier* instructions which act both as an acquire and a release barrier.

Sequential Consistency

A widely adopted notion of consistency for multi-threaded programs is *sequential consistency*, first stated by Lamport [1979] as the requirement that

> "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

We do not define this formally as we will neither assume nor enforce sequential consistency in our algorithm. Although sequential consistency can make correctness proofs easier for multi-threaded programs, it *does not imply correctness* of a multi-threaded program if correctness has been shown for sequential executions. The reason for this is that the sequential order mentioned above might result in an interleaving of operations that cannot appear in any correct sequential execution. Examples for this are e.g. programs that have to update two memory locations atomically. Because ensuring sequential consistency causes considerable performance losses in practice, modern processors provide only much weaker guarantees and leave it to the programmer to strengthen them by using fence and barrier instructions as needed.

Atomic Addition

We will need to concurrently perform additions

$$i := i + j$$

by different threads on the same variable stored in memory location $1 \leq i \leq m$ (for convenience, we use variables and memory locations synonymously). This involves performing read instructions r_i and r_j to load the current values stored in memory cells i and j to some processor registers R_1 and R_2, respectively, then adding the values in R_1 and R_2 and finally writing the result back in a write operation w_i to memory location i.

Even if synchronized read and write operations are used, two threads might concurrently read the same value $x \in \mathbb{R}$ from location i, calculate $x+\Delta_1$ and $x+\Delta_2$, respectively, for some values $\Delta_1, \Delta_2 \in \mathbb{R}$ and write their respective result back to memory in some order, so at the end memory location i contains $x+\Delta_1$ or $x+\Delta_2$, but not $x+\Delta_1+\Delta_2$.

One solution to this problem is to prevent other threads from accessing memory location i between execution of the read and write operations. This would mean mutual exclusion, however, with all disadvantages mentioned above. Modern processors offer a better solution, either by a *compare-and-swap* instruction (`CAS`), or a *load-link* and *store-conditional* instruction pair (LL/SC), which we already sketched at the beginning of section 3.3.

The `CAS` instruction takes a memory location $1 \leq i \leq m$ and two processor registers R_1 and R_2 as input. By calling `CAS(`i`, `R_1`, `R_2`)` the value stored in R_2 is written to memory location i if the values stored in memory location i and in R_1 are equal; otherwise the instruction has no effect. In the first case we say that the operation is *successful*, and *unsuccessful* otherwise. Checking the condition and writing to memory in case of success is performed atomically, which means that

i) no other thread can read from or write to memory location i during execution of the CAS instruction, and

ii) the CAS instruction cannot be preempted, i.e. thread execution can be suspended only before or after, but not during execution of the CAS instruction, so even though it forbids other threads to read from or write to memory location i for some time, it is non-blocking.

The CAS instruction both reads from and (if successful) writes to memory location i. It is also a synchronizing instruction, so $\texttt{CAS} \in \mathcal{I}^{rsync} \cap \mathcal{I}^{wsync}$. For convenient notation, we denote by val(w) the value written by a write operation w, and synonymously use a processor register R to refer to the value stored in it.

An AtomicAddition procedure can now be implemented by iteratively reading the value stored in memory location i, perform the addition in the processor registers, and trying to write the result back with the compare-and-swap instruction until it succeeds:

```
procedure AtomicAddition($i, \Delta$)
  Requires: A memory location $1 \leq i \leq m$, and a value $\Delta \in \mathbb{R}$, given in some
            processor register $R$
  Assures : Exactly one write operation $w$ is performed that has an effect on
            memory. Returns val$(w)$ and guarantees that val$(w)$ = val$(w')+\Delta$,
            where $w'$ is the last write operation on $i$ before $w$ w.r.t. the ordering
            $\prec_{\text{cell}}$.
1 repeat
2     Sync_Read($i$, $R_{\text{old}}$).    /* Synchr. read of memory location $i$ to register $R_{\text{old}}$ */
3     Set $R_{\text{new}} := R_{\text{old}} + R$.    /* Perform addition and store result in $R_{\text{new}}$ */
4 until CAS($i, R_{old}, R_{new}$) = "Success".
5 return $R_{\text{new}}$.
```

For each unsuccessful CAS operation in an iteration $k \geq 2$, by definition 3.13 another thread must have written to memory location i *after* the CAS operation performed in iteration $k-1$ w.r.t. the $\prec_{\text{cell}}$ order. This means that if one thread is *busy waiting*, i.e. spinning in the loop of the AtomicAddition procedure, there is some other thread that makes progress. AtomicAddition is therefore non-blocking, but of course not wait-free because the number of iterations of the loop cannot be bounded.

Remark. It is not precluded that in a (final) successful iteration other threads change the content of memory location i from some value a to another value b and then back to a again between the Sync_Read and the CAS operation, still allowing the latter to succeed.

This can lead to problems commonly known as *ABA problems* in literature if the content of memory location i does not define the complete state of a shared object and thus atomically updating i does not guarantee the entire object to be in a consistent state. Valois [1995] illustrates this problem in detail for a lock-free concurrent linked list data structure, and shows how the ABA problem can be circumvented, allowing lock-free concurrent linked lists to be implemented on the basis of `CAS`. In our case, the entire object or variable is stored in memory location i, so `AtomicAddition` is not susceptible to the ABA problem.

Some processor architectures, e.g. PowerPC, provide a stronger mechanism than `CAS` by a pair of *load-link* (`LL`) and *store-conditional* (`SC`) instructions. Like `CAS`, both of them are synchronizing instructions. A load-link operation establishes a *reservation* for the memory location it reads from. Reservations for the same memory location can be established by different threads simultaneously. A store-conditional instruction then checks if the reservation established by the respective thread still exists and in this (successful) case kills all reservations for this location made by any threads. Therefore `LL/SC` instruction pairs do not establish a critical area as mutual exclusion approaches do, but avoid ABA problems as mentioned above. The only important fact in our case however is that `AtomicAddition` can be equivalently implemented by replacing the `Sync_Read` by an `LL` instruction, and the `CAS` by an `SC` instruction on platforms where `LL/SC` is available, but not `CAS`. Practically all modern microprocessors provide at least one of `CAS` or `LL/SC`.

As each call to `AtomicAddition` performs exactly one operation that modifies the content of the target memory location, we can say that a call a_1 to `AtomicAddition` precedes another call a_2 w.r.t. $\prec_{\text{cell}}$ iff the same holds for the corresponding successful `CAS` or `SC` operations, respectively.

3.3.2 A Lock-free Parallel Algorithm for the Resource Sharing Problem

The approximation algorithm that we presented in section 3.2 — naturally — leaves some room for suboptimality, which we will use for allowing the block solvers to work on outdated resource prices in the parallel variant of the algorithm that we are going to present in this section. A certain tolerance of volatility in resource prices will be essential as price increments caused by resource allocations made in one thread are seen with a delay by other threads, and of course two threads might also decide at the same time to use some resource for different customers. Because local view order consistency might be provided only for synchronizing read operations, resource prices seen by non-synchronizing read operations might not even be guaranteed to be non-decreasing, which was a crucial property in the analysis of the sequential RESOURCE SHARING ALGORITHM. To accomodate this, we introduce the notion of *volatility-tolerant* block solvers which provide some optimality guarantee also if prices change within certain bounds while they perform their computations. It will be possible that these bounds are exceeded and a sufficient optimality guarantee cannot be given. This will be detected by the algorithm, and the customer will then be scheduled for sequential recomputation at the end of the current phase. Thus low volatility of resource prices will not be relevant for correctness, but only for achieving good parallelization speedups.

Like the sequential RESOURCE SHARING ALGORITHM, the parallel algorithm maintains a vector α of (upper bounds on) resource utilizations stored in memory. However, now α is volatile (i.e. can change at any point in time) from each threads' perspective because other threads can modify it without prior coordination. To capture this formally, we will not pass a static vector to block solvers any more as in the sequential algorithm, but a resource price oracle $H_\alpha^\pi : \mathcal{R} \times \mathbb{R}_+ \to \mathbb{R}_+$ for each thread $\pi \in \{1,\dots,\Pi\}$ ($\Pi \in \mathbb{N}$) with

$$H_\alpha^\pi(r,\tau) := y_r(\max\{\alpha_r^\pi(\tau), \alpha_r^{(p-1)} + L_r\}),$$

where $\alpha_r^\pi(\tau) \in \mathbb{R}_+^{\mathcal{R}}$ for each $r \in \mathcal{R}$ is the value of α_r that thread π would observe in his view of memory at time τ, $1 \le p \le t$ is the current phase and $L = \sum_{c\in\mathcal{C}} l_c$ is the sum of lower bounds as in the sequential algorithm. As in the sequential algorithm, the value of $\alpha_r^{(p-1)} + L_r$ is constant throughout phase p ($1 \le p \le t$), but α_r can be changed concurrently by all threads. A query $H_\alpha^\pi(r,\tau)$ now amounts to a read operation on the variable α_r performed by thread π ($1 \le \pi \le \Pi$) at time $\tau \in \mathbb{R}_+$, and the computation of $y_r(\max\{x, \alpha_r^{(p-1)} + L_r\})$, where x is the value of α_r observed by this read operation. Of course thread $\pi \in \{1,\dots,\Pi\}$ will not call $H_\alpha^{\pi'}$ for $\pi' \neq \pi$. Moreover, we define that thread π ($\pi \in \{1,\dots,\Pi\}$) always calls the price oracle H_α^π with τ equal to the time at which the call is issued. Therefore the time parameter becomes redundant and is omitted in the following. Due to the weak memory model that we assume, resource prices observed by two threads can be different at the same time, i.e. it is possible that $H_\alpha^\pi(r,\tau) \neq H_\alpha^{\pi'}(r,\tau)$ for some $r \in \mathcal{R}$, $\tau \in \mathbb{R}_+$ and $1 \le \pi < \pi' \le \Pi$.

We can now define volatility-tolerant block solvers. A call to a block solver will be processed by exactly one thread, i.e. there is no multi-threading inside a block solver.

Definition 3.16. *Let $c \in \mathcal{C}$, and $l_c \in \mathbb{R}_+^{\mathcal{R}}$ lower bounds on resource consumptions $g_c(b)$, $b \in \mathcal{B}_c$. Let* `Solver`$_c$ *be a procedure that gets a resource price oracle H as input and produces a pair $(b,\gamma) \in \mathcal{B}_c \times \mathbb{R}_+$ as output.* `Solver`$_c$ *is called a* volatility-tolerant *block solver if for any call to it the following condition is guaranteed to be satisfied:*

Let Y_r ($r \in \mathcal{R}$) be the set of resource prices sampled by calling $H(r)$ in this execution of `Solver`$_c$*. Then there is a vector $\omega' \in \mathbb{R}_+^{\mathcal{R}}$ with $\omega'_r \in Y_r$ for each $r \in \mathcal{R}$ with $Y_r \neq \emptyset$, and $\omega'_r = 0$ otherwise, such that for each vector $\omega \in \mathbb{R}_+^{\mathcal{R}}$ with $\omega_r = \omega'_r$ for $r \in \mathcal{R}$ with $Y_r \neq \emptyset$ we have*

$$\gamma = \omega^\top g_c(b)$$

and

$$\gamma \le (1+\varepsilon_0)\mathrm{opt}_c(\omega).$$

A volatility-tolerant block solver thus guarantees that the result it returns is $(1+\varepsilon_0)$-optimal w.r.t. some resource prices ω that it observed (not necessarily at the same time) during its computations, or that cannot be relevant for optimality.

Trivially, from each block solver as defined in section 3.1 for static resource prices we can construct a volatility-tolerant block solver by querying the price of each resource once, thus creating a copy which we pass to the original block solver. Sometimes however this copying step is not necessary, e.g. if the block solver is implemented by Dijkstra's algorithm:

DIJKSTRA'S ALGORITHM WITH VOLATILE EDGE COSTS

```
Input  : An undirected graph $G := (V,E)$, vertex sets $S,T \subseteq V$, and a volatile
         edge cost oracle $H : E \to \mathbb{R}_+$.
Output: $(P,\gamma) \in 2^E \times \mathbb{R}_+$.
1  Set $d(s) := 0$ for $s \in S$ and $d(v) := \infty$ for each $v \in V \setminus S$.
2  Set $p(v) := v$ for each $v \in V$.
3  Set $Q := \emptyset$.
4  $\omega_e := 0$ for each $e \in E$.
5  while $T \cap Q = \emptyset$ do
6      Let $v \in V \setminus Q$ with $d(v) \leq d(v')$ for all $v' \in V \setminus Q$.
7      Set $Q := Q \cup \{v\}$.
8      foreach $e := \{v,w\} \in E$ with $w \notin Q$ do
9          Set $\omega_e := H(e)$.            /* Query current cost of edge e */
10         if $d(w) > d(v) + \omega_e$ then
11             Set $d(w) := d(v) + \omega_e$.
12             Set $p(w) := v$.
13 Choose $t \in T \cap Q$ and set $v := t$.
14 Set $P := \emptyset$.
15 while $p(v) \neq v$ do
16     Set $P := P \cup \{\{v, p(v)\}\}$.
17     Set $v := p(v)$.
18 Return $(P, d(t))$.
```

Proposition 3.17. *Let $G = (V,E)$ be an undirected graph, $H : E \to \mathbb{R}_+$ a volatile edge cost oracle and $S,T \subseteq V$ such that there is an S-T-path in G, i.e. a path from s to t for an $s \in S$ and $t \in T$. With a block being the convex hull of all incidence vectors of S-T-paths,* DIJKSTRA'S ALGORITHM WITH VOLATILE EDGE COSTS *is a volatility-tolerant block solver for the shortest paths problem and $\varepsilon_0 := 0$.*

Proof. Clearly, the algorithm returns an S-T-path. At the end of the algorithm, $d' : V \to \mathbb{R}_+$ with $d'(v) := \min\{d(v), \min_{t\in T} d(t)\}$ is a feasible potential w.r.t. the edge costs ω, so for each $e = \{v,w\} \in E$ we have

$$\omega_e + d'(v) - d'(w) \quad \begin{cases} = 0 : e \in P \\ \geq 0 : \text{otherwise} \end{cases} . \tag{3.24}$$

This follows from the fact that the cost of any edge is sampled at most once in line 9, and for each edge $\{v,w\} \in E$ whose cost has not been sampled we must have $\min\{d(v),d(w)\} \geq \min_{t\in T} d(t)$. Thus by (3.24) P is a shortest S-T-path w.r.t. the edge costs that have been sampled in line 9, and independent of the costs on those edges which have not been queried (increasing ω_e from zero to a positive value for an edge $e \in E$ whose cost has not been queried cannot violate (3.24)). □

Our shared-memory-parallel resource sharing algorithm will make use of the following simple fact:

Proposition 3.18. *Let $(b,\gamma) \in \mathcal{B}_c \times \mathbb{R}_+$ be the result of a call to* `Solver`$_c$*, and choose $\omega' \in \mathbb{R}_+^{\mathcal{R}}$ as in definition 3.16. Let $\omega'' \in \mathbb{R}_+^{\mathcal{R}}$ with $\omega'' \geq \omega'$, and $\varepsilon_1 \geq 0$. Then*

$$g_c(b)^\top \omega'' \leq (1+\varepsilon_1)\gamma \tag{3.25}$$

implies

$$g_c(b)^\top \omega'' \leq (1+\varepsilon_1)(1+\varepsilon_0)\mathrm{opt}_c(\omega'').$$

Proof. Let $g_c(b)^\top \omega'' \leq (1+\varepsilon_1)\gamma$. Then by definition 3.16 we have

$$g_c(b)^\top \omega'' \leq (1+\varepsilon_1)(1+\varepsilon_0)\mathrm{opt}_c(\omega').$$

The claim follows from the fact that $\mathrm{opt}_c(\omega') \leq \mathrm{opt}_c(\omega'')$ since $\omega'' \geq \omega'$. □

In the analysis of the algorithm, ω' will be unknown, but we will know the value $\gamma = g_c(b)^\top \omega'$ returned by the block solver, and $g_c(b)^\top \omega''$ for a vector ω'' that is guaranteed to satisfy $\omega'' \geq \omega'$. The algorithm will check if condition (3.25) is satisfied, i.e. resource prices did not increase too much. If this is the case, the solution returned by the block solver provides a sufficiently strong optimality guarantee, and the algorithm *accepts* it. Otherwise it *rejects* the solution and schedules the customer to be processed again sequentially at the end of the current phase. Therefore with higher volatility the algorithm becomes increasingly sequential, and better parallelization speedups can be obtained with lower volatility. However, we do not rely on low volatility for correctness. We will prove that a near-optimum solution is found independent of how fast resource prices change during the computation of a block solver. Results on large resource sharing instances from VLSI global routing presented in section 4.9 show that by a simple partitioning heuristic the "collision probability", i.e. the probability that block solvers called by two threads concurrently return a solution using the same resources (and thus increasing observed volatility), is sufficiently small to achieve very good speedups in practice.

The algorithm distinguishes between resource utilizations $\alpha : \mathcal{R} \to \mathbb{R}_+$ of *accepted* solutions and resource utilizations $\alpha' : \mathcal{R} \to \mathbb{R}_+$ of tentative solutions, i.e. solutions that have been computed already by a volatile price tolerant block solver, but have not yet been accepted. It will call two functions `Cost`(a) and `Cost'`(a,b) to sample the current cost of some resource allocation vector $a \in \mathbb{R}_+^{\mathcal{R}}$ w.r.t. costs induced by observed resource utilizations α and $\alpha + \alpha' - b$, respectively:

```
procedure Cost($a \in \mathbb{R}_+^{\mathcal{R}}$):
  begin
    Return $\sum_{r \in \mathcal{R}: a_r > 0} a_r H_\alpha^\pi(r)$.
  end
```

```
procedure Cost'($a \in \mathbb{R}_+^{\mathcal{R}}, b \in \mathbb{R}_+^{\mathcal{R}}$):
  begin
    Return $\sum_{r \in \mathcal{R}: a_r > 0} a_r \left(H_{\alpha'+\alpha-b}^\pi(r)\right)$.
  end
```

Similarly to the oracle H_α^π, $H_{\alpha'+\alpha-b}^\pi$ performs a read operation on α'_r, α_r and b_r, respectively, and computes $y_r(\max\{x'+x-b_r, \alpha_r^{(p-1)}+L_r\})$ if called for resource $r \in \mathcal{R}$, where x' and x are the values sampled by the read operations on α'_r and α_r (b_r is a local, i.e. non-volatile, variable owned by the thread). We will need that the read operation l' on α'_r ($r \in \mathcal{R}$) is *executed before* the read operation l on α_r, i.e. $l' \prec_{\text{exec}} l$. This requires a fence between the corresponding read instructions (note that even x86 processors do not guarantee that read operations are not reordered with each other).

The algorithm is shown on page 76. During each phase, it assigns a different number n_c to each customer $c \in \mathcal{C}$. This number is used for the correctness proof in Theorem 3.27 below, where we will analyze the increase in total resource prices sequentially with customers ordered w.r.t. increasing numbers assigned to them. Although the *value* stored in n_c for each $c \in \mathcal{C}$ in each phase of the algorithm is not of importance otherwise, the fact that all threads have to read from and write to the shared variable n in line 22 is used in the proofs of Lemmata 3.24 and 3.25 below. On some architectures, it might be possible to collapse lines 21 to 23 to a single full barrier instruction and a reordering fence. In our weaker model however we actually need the pivot line 22 for correctness.

We do not specify how the scheduling of customers to processors is done in lines 7 and 8. One possibility is e.g. to use a lock-free linked list as proposed by Sundell and Tsigas [2005] that holds all customers at the beginning of a phase, and let each thread concurrently remove the first item from the list to get the next customer to process, until the list is empty. In our application of the algorithm to VLSI routing we will employ a job selection heuristic for reducing collision probabilities, see section 4.7.2. With $\varepsilon_1 > 0$ as in the proof of Lemma 3.5, the number of solutions which have to be rejected in this way becomes extremely small in practice.

Lock-free linked lists can also be used to maintain the set $\mathcal{S}$ of customers which have to be processed again sequentially at the end of a phase. As mentioned above, we will make use of the numbers assigned to customers for the analysis of the algorithm:

```
PARALLEL RESOURCE SHARING ALGORITHM
Input  : An instance of the MIN-MAX RESOURCE SHARING PROBLEM as in
         the sequential RESOURCE SHARING ALGORITHM, but with volatile
         price tolerant block solvers. A number $\Pi$ of processors.
Output: For each $c \in \mathcal{C}$ a convex combination of vectors in $\mathcal{B}_c$, given by
         $\sum_{b \in \mathcal{B}_c} x_{c,b} b$. A cost vector $\omega \in \mathbb{R}_+^{\mathcal{R}}$.
1  Set $\alpha_r := 0$ and $\alpha'_r := 0$ for each $r \in \mathcal{R}$.
2  Set $x_{c,b} := 0$ for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$.
3  Compute $L := \sum_{c \in \mathcal{C}} l_c$.
4  for $p := 1$ to $t$ do
5      Set $\mathcal{S} := \emptyset$.
6      Set $n := 0$.
7      foreach $c \in \mathcal{C}$ do in parallel on processors $1 \le \pi \le \Pi$
8          ParallelAllocateResources ($c, \pi$).
9      acquire_memory_barrier.
10     foreach $c \in \mathcal{S}$ do sequentially
11         ParallelAllocateResources ($c, 1$).
12 Set $x_{c,b} := \frac{1}{t} x_{c,b}$ for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$.
----------------------------------------------------------------------
13 ParallelAllocateResources ($c \in \mathcal{C}, 1 \le \pi \le \Pi$):
14 begin
15     Set $\Phi_c^{lb} :=$ Cost($l_c$).
16     if $p = 1$ or $Z_c^{lb} :=$ Cost($a_c$) $> (1+\varepsilon_1)\big(z_c + (1+\varepsilon_0)(\Phi_c^{lb} - \phi_c)\big)$ then
17         Set $(\bar{b}_c, \bar{z}_c) :=$ Solver$_c(H_\alpha^\pi)$, $\bar{a}_c := g_c(\bar{b}_c)$ and $\bar{\phi}_c := \infty$.
18     else
19         Set $\bar{a}_c := a_c$, $\bar{z}_c := z_c$ and $\bar{\phi}_c := \phi_c$.
20     foreach $r \in \mathcal{R}$ with $(\bar{a}_c)_r > 0$ do AtomicAddition($\alpha'_r, (\bar{a}_c)_r$).
21     acquire_memory_barrier. fence.
22     Set $n_c :=$ AtomicAddition($n, 1$).
23     release_memory_barrier. fence.
24     Set $Z_c^{ub} :=$ Cost'($\bar{a}_c, \bar{a}_c$).
25     if $\bar{\phi}_c = \infty$ then
26         Set $\Phi_c^{ub} :=$ Cost'($l_c, \bar{a}_c$) and $\bar{\phi}_c := \Phi_c^{ub}$.
27     if $Z_c^{ub} \le (1+\varepsilon_1)\big(\bar{z}_c + (1+\varepsilon_0)(\Phi_c^{lb} - \bar{\phi}_c)\big)$ then
28         Set $b_c := \bar{b}_c$, $a_c := \bar{a}_c$, $z_c := \bar{z}_c$ and $\phi_c := \bar{\phi}_c$.          /* accept */
29         Set $x_{c,b_c} := x_{c,b_c} + 1$.
30         foreach $r \in \mathcal{R}$ with $(\bar{a}_c)_r > 0$ do AtomicAddition($\alpha_r, (\bar{a}_c)_r$).
31         acquire_memory_barrier.
32     else
33         Set $\mathcal{S} := \mathcal{S} \cup \{c\}$.                                      /* reject */
34     foreach $r \in \mathcal{R}$ with $(\bar{a}_c)_r > 0$ do AtomicAddition($\alpha'_r, -(\bar{a}_c)_r$).
35 end
```

Definition 3.19. *For each $c \in \mathcal{C}$ and $1 \leq p \leq t$, let $n_c^{(p)}$ be the value of n_c at the end of phase p. For each $1 \leq p \leq t$, let $n^{(p)}$ be the value of n at the end of phase p.*

We say that an execution of the procedure `ParallelAllocateResources` is *successful* iff it executes lines 28 to 31. We start with the following simple observation:

Lemma 3.20. *Assume that a call to* `ParallelAllocateResources` *is executed in some thread. If $\alpha'_r = 0$ for each $r \in \mathcal{R}$ initially, and no other thread is active throughout the call, it cannot be unsuccessful.*

Proof. Resource utilizations α and α' are only changed by the single active thread that executes `ParallelAllocateResources`, so we have $\alpha' = \bar{a}_c$ when reaching line 24. If line 17 has been executed, the statement follows because $Z_c^{ub} = \bar{z}_c$ and $\bar{\phi}_c = \Phi_c^{ub} = \Phi_c^{lb}$. Otherwise we have

$$Z_c^{lb} \leq (1+\varepsilon_1)\Big(z_c + (1+\varepsilon_0)(\Phi_c^{lb} - \phi_c)\Big)$$

by the check performed in line 16, $z_c = \bar{z}_c$, $\phi_c = \bar{\phi}_c$ and $Z_c^{lb} = Z_c^{ub}$ because $\alpha' = \bar{a}_c$. □

Lemma 3.21. *$\alpha'_r = 0$ for each $r \in \mathcal{R}$ immediately after line 9.*

Proof. The sum of values added to α'_r in a call to `ParallelAllocateResources` is clearly zero for each $r \in \mathcal{R}$. Updates to α' by different threads can be interleaved, but as `AtomicAddition` is used, no terms are lost, and therefore after all concurrent executions of `ParallelAllocateResources` have finished, $\alpha'_r = 0$ must become visible for each $r \in \mathcal{R}$ after the acquire barrier. □

Corollary 3.22. *In each phase $1 \leq p \leq t$, there is exactly one successful call to* `ParallelAllocateResources` *for each $c \in \mathcal{C}$.*

Proof. The statement follows from Lemmata 3.20 and 3.21 and the fact that c is added to $\mathcal{S}$ if the first call to `ParallelAllocateResources` for customer $c \in \mathcal{C}$ in phase p is unsuccessful. □

Definition 3.23. *For each $1 \leq p \leq t$ and $0 \leq i \leq n^{(p)}$, let*

$$\alpha^{(p,i)} := \alpha^{(p-1)} + \sum_{c \in \mathcal{C}: n_c^{(p)} \leq i} a_c^{(p)},$$

where $\alpha^{(0)} :\equiv 0$, $\alpha^{(p)} := \alpha^{(p,n^{(p)})}$ and $a_c^{(p)}$ is the resource allocation vector a_c at the end of the unique successful call to `ParallelAllocateResources` *for customer c in phase p. Further, let $\omega^{(p,i)}, \tilde{\omega}^{(p,i)} \in \mathbb{R}_+^{\mathcal{R}}$ with*

$$\omega_r^{(p,i)} := y_r(\alpha_r^{(p,i)})$$

and

$$\tilde{\omega}_r^{(p,i)} := y_r(\max\{\alpha_r^{(p,i)}, \alpha_r^{(p-1)} + L_r\}),$$

and $\omega^{(p)} := \omega^{(p,n^{(p)})}$ and $\tilde{\omega}^{(p)} := \tilde{\omega}^{(p,n^{(p)})}$ for each $1 \leq p \leq t$.

Remark. For given $1 \le p \le t$ and $0 \le i \le n^{(p)}$, there may be no point in time during execution of the PARALLEL RESOURCE SHARING ALGORITHM at which $\alpha = \alpha^{(p,i)}$. We need these definitions only for the correctness proof of our algorithm.

In the following, we consider an execution $(\mathcal{O}, rf)$ of the PARALLEL RESOURCE SHARING ALGORITHM. We will use the following Lemma to bound the resource prices observable by a block solver:

Lemma 3.24. *Let* $1 \le p \le t$ *and* $c \in \mathcal{C}$. *Let* $r \in \mathcal{R}$, *and* $\tilde{\omega}'_r$ *be the value returned by a resource price oracle call* $H^\pi_\alpha(r)$ *in the successful call to* `ParallelAllocateResources` *for customer c in phase p before line 21. Then*

$$\tilde{\omega}'_r \le \tilde{\omega}_r^{(p, n_c^{(p)} - 1)}.$$

Proof. Let l be the (non-synchronizing) read operation that the call to H^π_α performed on the variable α_r, sampling the value $x \in \mathbb{R}_+$ with $y_r(\max\{x, \alpha_r^{(p-1)} + L_r\}) = \tilde{\omega}'_r$. Assume that

$$\tilde{\omega}'_r > \tilde{\omega}_r^{(p, n_c^{(p)} - 1)}.$$

Let w_1 be the write operation whose output is read by l, i.e. $rf(l) = w_1$. As y_r is strictly monotone, there must be a write operation w_3 with $w_3 \preceq_{\text{cell}} w_1$ that is performed in line 30 in the successful call to `ParallelAllocateResources` for a customer $c' \in \mathcal{C}$ in phase p with $n_{c'}^{(p)} > n_c^{(p)}$. Let b denote the barrier operation performed in line 23 in that call, and w_2 the write operation that writes $n_{c'}^{(p)}$ to n in the call to `AtomicAddition` in phase p.

If l_2 is the synchronizing read operation performed by `AtomicAddition` immediately before writing $n_c^{(p)}$ to n in phase p, we clearly have $rf(l_2) \prec_{\text{cell}} w_2$. Since $w_2 \prec_{\text{prog}} b \prec_{\text{prog}} w_3$ and $l \prec_{\text{exec}} l_2$ because of the fence in line 21, we must have $rf(l) \prec_{\text{cell}} w_3$ by definition 3.15, which is a contradiction. $\square$

Similarly, we have:

Lemma 3.25. *Let* $1 \le p \le t$ *and* $c \in \mathcal{C}$. *Let* $r \in \mathcal{R}$, *and* $\tilde{\omega}''_r$ *be the value returned by a resource price oracle call* $H^\pi_{\alpha'+\alpha-\bar{a}_c}(r)$ *in the successful call to* `ParallelAllocate-Resources` *for customer c in phase p after line 23. Then*

$$\tilde{\omega}''_r \ge \tilde{\omega}_r^{(p, n_c^{(p)} - 1)}.$$

Proof. Let l and l', respectively, be the read operations that the call to the oracle $H^\pi_{\alpha'+\alpha-\bar{a}_c}$ performed on the variables α_r and α'_r, sampling the values x and x' with $y_r(\max\{x + x' - (\bar{a}_c)_r, \alpha_r^{(p-1)} + L_r\}) = \tilde{\omega}''_r$.

Let $c' \in \mathcal{C}$ be any customer with $n_{c'}^{(p)} \le n_c^{(p)}$ and $(a_c^{(p)})_r > 0$, and let w_1 be the write operation performed on the variable α'_r in line 20 in the successful call to `Parallel-AllocateResources` for customer c' in phase p. Similarly, let w_2 be the write operation on α_r in the same call in line 30, and w_3 the write operation on α'_r in line 34.

By a similar reasoning as in the proof of Lemma 3.24, we can deduce $w_1 \preceq_{\text{cell}} rf(l')$ because of the acquire barrier in line 21. If $w_3 \preceq_{\text{cell}} rf(l')$, then because of the acquire barrier in line 31 and our assumptions on the execution order of read operations performed in the procedure `Cost'`, we also have $w_2 \preceq_{\text{cell}} rf(l)$. Moreover, if $p > 1$, $w \preceq_{\text{cell}} rf(l)$ for any write operation w on variable α_r performed in phase $p-1$, so we have

$$x + x' \geq \alpha^{(p-1)} + \sum_{c'' \in \mathcal{C}: n_{c''}^{(p)} \leq n_c^{(p)}} a_c^{(p)} = \alpha^{(p, n_c^{(p)})}$$

and the claim follows by the monotonicity of y_r. □

We obtain the following Corollary:

Corollary 3.26. *Let $1 \leq p \leq t$ and $c \in \mathcal{C}$. Then*

$$\Phi_c^{lb} \leq \left(\tilde{\omega}^{(p, n_c^{(p)} - 1)}\right)^\top l_c \leq \Phi_c^{ub}$$

and

$$\max\left\{Z_c^{lb}, \bar{z}_c\right\} \leq \left(\tilde{\omega}^{(p, n_c^{(p)} - 1)}\right)^\top a_c^{(p)} \leq Z_c^{ub}$$

in the successful call to `ParallelAllocateResources` *for customer c in phase p.* □

We can now prove the same approximation guarantee as for the sequential RESOURCE SHARING ALGORITHM in Theorem 3.4 also for the PARALLEL RESOURCE SHARING ALGORITHM:

Theorem 3.27. *Let $\Lambda \in \mathbb{R}$ with $g_c(b) \leq \Lambda \mathbb{1}$ for $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$. Let $\varepsilon_0, \varepsilon_1 \geq 0$ and $\varepsilon_2 > 0$. Let $\varepsilon' := \frac{1}{\Lambda}(e^{\varepsilon_2 \Lambda} - 1)(1+\varepsilon_1)(1+\varepsilon_0)$. Assume that $\varepsilon' \lambda^* < 1$, where λ^* is the optimum value of (3.1). Let $t \in \mathbb{N}$ and $\xi := \frac{\varepsilon'}{\varepsilon_2(1-\varepsilon'\lambda^*)} + \frac{\ln(\Psi)}{\varepsilon_2 t \lambda^*}$. Then the* PARALLEL RESOURCE SHARING ALGORITHM *computes a ξ-approximate solution to (3.1).*

Proof. By Corollary 3.22 we have $\sum_{b \in \mathcal{B}_c} x_{c,b} = 1$ at the end of the algorithm. Clearly $x_{c,b} \geq 0$ for all $b \in \mathcal{B}_c$, so the algorithm determines a convex combination of vectors in $\mathcal{B}_c$ for each $c \in \mathcal{C}$.

Again we will consider the term $(\omega^{(p)})^\top \mathbb{1}$ for all phases p. Initially we have

$$(\omega^{(0)})^\top \mathbb{1} = \Psi. \tag{3.26}$$

We can estimate the increase $\left(\omega^{(p)} - \omega^{(p-1)}\right)^\top \mathbb{1}$ as follows $(1 \leq p \leq t)$. Let $1 \leq i \leq n^{(p)}$. If the call to `ParallelAllocateResources` in phase p in which i was written to variable n was unsuccessful, we have

$$(\omega^{(p,i-1)})^\top \mathbb{1} = (\omega^{(p,i)})^\top \mathbb{1}. \tag{3.27}$$

Otherwise, $i = n_c^{(p)}$ for a unique customer $c \in \mathcal{C}$. We have

$$\omega_r^{(p,i)} \leq \omega_r^{(p,i-1)} e^{\varepsilon_2 (a_c^{(p)})_r} \leq \omega_r^{(p,i-1)} + \frac{1}{\Lambda}(e^{\varepsilon_2\Lambda} - 1)\omega_r^{(p,i-1)}(a_c^{(p)})_r \tag{3.28}$$

for each $r \in \mathcal{R}$ because $(a_c)_r^p \leq \Lambda$ for $r \in \mathcal{R}$, $\varepsilon_2\Lambda \leq 1$, and $e^x \leq 1 + \frac{e^{\varepsilon_2\Lambda}-1}{\varepsilon_2\Lambda} x$ for $0 \leq x \leq \varepsilon_2\Lambda$. We get

$$(\omega^{(p)})^\top \mathbb{1} \;\; \leq \;\; (\omega^{(p-1)})^\top \mathbb{1} + \frac{1}{\Lambda}(e^{\varepsilon_2\Lambda} - 1) \sum_{c' \in \mathcal{C}} \left(\omega^{(p,n_{c'}^{(p)}-1)}\right)^\top a_{c'}^{(p)} \tag{3.29}$$

and thus

$$(\omega^{(p)})^\top \mathbb{1} \;\; \leq \;\; (\omega^{(p-1)})^\top \mathbb{1} + \frac{1}{\Lambda}(e^{\varepsilon_2\Lambda} - 1) \sum_{c' \in \mathcal{C}} \left(\tilde{\omega}^{(p,n_{c'}^{(p)}-1)}\right)^\top a_{c'}^{(p)}, \tag{3.30}$$

as clearly $\tilde{\omega}^{(p,n_{c'}^{(p)}-1)} \geq \omega^{(p,n_{c'}^{(p)}-1)}$ for each $c' \in \mathcal{C}$. Next we show that

$$\left(\tilde{\omega}^{(p,i-1)}\right)^\top a_c^{(p)} \;\; \leq \;\; (1+\varepsilon_1)\left(z_c^{(p)} + (1+\varepsilon_0)\left(\left(\tilde{\omega}^{(p,i-1)}\right)^\top l_c - \phi_c^{(p)}\right)\right) \tag{3.31}$$

and

$$z_c^{(p)} + (1+\varepsilon_0)\left(\left(\tilde{\omega}^{(p,i-1)}\right)^\top l_c - \phi_c^{(p)}\right) \;\; \leq \;\; (1+\varepsilon_0)\,\mathrm{opt}_c(\tilde{\omega}^{(p,i-1)}). \tag{3.32}$$

Inequality (3.31) follows directly from Corollary 3.26 and the fact that the condition checked in line 27 is satisfied. If `Solver`$_c$ is called in this execution of `Parallel-AllocateResources`, inequality (3.32) holds because by Lemma 3.24

$$z_c^{(p)} \leq (1+\varepsilon_0)\,\mathrm{opt}_c(\tilde{\omega}')$$

for some $\tilde{\omega}' \in \mathbb{R}_+^{\mathcal{R}}$ with $\tilde{\omega}' \leq \tilde{\omega}^{(p,i-1)}$, and $\phi_c^{(p)} \geq \left(\tilde{\omega}^{(p,i-1)}\right)^\top l_c$ by Corollary 3.26. Otherwise, we have $p > 1$, $z_c^{(p)} = z_c^{(p-1)}$ and $\phi_c^{(p)} = \phi_c^{(p-1)}$. Let $1 \leq j \leq n^{(p-1)}$ with $j = n_c^{(p-1)}$. Then we had

$$z_c^{(p-1)} + (1+\varepsilon_0)\left(\left(\tilde{\omega}^{(p-1,j-1)}\right)^\top l_c - \phi_c^{(p-1)}\right) \leq (1+\varepsilon_0)\,\mathrm{opt}_c(\tilde{\omega}^{(p-1,j-1)})$$

in phase $p-1$, so inequality (3.32) must hold because

$$\mathrm{opt}_c\left(\tilde{\omega}^{(p,i-1)}\right) - \mathrm{opt}_c\left(\tilde{\omega}^{(p-1,j-1)}\right) \geq \left(\tilde{\omega}^{(p,i-1)} - \tilde{\omega}^{(p-1,j-1)}\right)^\top l_c,$$

as $(l_c)_r \leq a_r$ for each $a \in \mathcal{A}_c$ and $r \in \mathcal{R}$.

By combining (3.30), (3.31), and (3.32), and using that the monotonicity of $\tilde{\omega}$ implies $\mathrm{opt}_c(\tilde{\omega}^{(p,i-1)}) \leq \mathrm{opt}_c(\tilde{\omega}^{(p)})$ we obtain:

$$(\omega^{(p)})^\top \mathbb{1} \leq (\omega^{(p-1)})^\top \mathbb{1} + \varepsilon' \sum_{c \in \mathcal{C}} \mathrm{opt}_c(\tilde{\omega}^{(p)}), \tag{3.33}$$

where $\varepsilon' := \frac{1}{\Lambda}(e^{\varepsilon_2\Lambda}-1)(1+\varepsilon_1)(1+\varepsilon_0)$. Because $\alpha^{(p-1)}+L \le \alpha^{(p)}$ and thus $\tilde{\omega}^{(p)} = \omega^{(p)}$, we get

$$(\omega^{(p)})^\top \mathbb{1} \le (\omega^{(p-1)})^\top \mathbb{1} + \varepsilon' \sum_{c\in\mathcal{C}} \mathrm{opt}_c(\omega^{(p)}). \tag{3.34}$$

The remaining part of the proof proceeds as the proof of Theorem 3.4 for the sequential RESOURCE SHARING ALGORITHM: By Lemma 3.2

$$\lambda_{lb}^{(p)} := \frac{\sum_{c\in\mathcal{C}} \mathrm{opt}_c(\omega^{(p)})}{(\omega^{(p)})^\top \mathbb{1}} \tag{3.35}$$

is a lower bound on the optimum value λ^* of (3.1). As $\varepsilon'\lambda_{lb}^{(p)} \le \varepsilon'\lambda^* < 1$, inequality (3.34) yields

$$(\omega^{(p)})^\top \mathbb{1} \le \frac{1}{1-\varepsilon'\lambda^*}(\omega^{(p-1)})^\top \mathbb{1}. \tag{3.36}$$

Combining (3.26) and (3.36) we get:

$$(\omega^{(t)})^\top \mathbb{1} \le \frac{\Psi}{(1-\varepsilon'\lambda^*)^t} = \Psi\left(1+\frac{\varepsilon'\lambda^*}{1-\varepsilon'\lambda^*}\right)^t \le \Psi e^{t\varepsilon'\lambda^*/(1-\varepsilon'\lambda^*)}, \tag{3.37}$$

as $1+x \le e^x$ for $x \ge 0$. Lemma 3.3 and (3.37) yield:

$$\sum_{c\in\mathcal{C}} \left(g_c \left(\sum_{b\in\mathcal{B}_c} x_{c,b} b \right) \right)_r \le \frac{1}{\varepsilon_2 t}\left(\ln\Psi + \frac{t\varepsilon'\lambda^*}{(1-\varepsilon'\lambda^*)} \right). \tag{3.38}$$

□

Using Theorem 3.27, Theorems 3.6 and 3.7 apply unchanged to the PARALLEL RESOURCE SHARING ALGORITHM.

We finally remark that it is possible to replace the sequential processing of the customers in $\mathcal{S}$ at the end of a phase by another parallel processing step (possibly with fewer threads and hopefully accepting most solutions) followed by a sequential processing of a smaller set of customers.

3.4 Randomized Rounding

Now we show how to round a fractional solution without increasing the worst congestion λ too much. We need the following Lemma by Raghavan and Spencer (see Raghavan [1988]), a variation of Chernoff's bound (Chernoff [1952]).

Lemma 3.28. *Let $X_1,\dots,X_k$ be independent random variables in $[0,1]$. Let μ be the sum of their expectations, and let $\varepsilon > 0$. Then $X_1+\cdots+X_k > (1+\varepsilon)\mu$ with probability less than $e^{-\mu f(\varepsilon)}$, where $f(\varepsilon) := (1+\varepsilon)\ln(1+\varepsilon)-\varepsilon$.*

Proof. Let Prob$[\cdot]$ denote the probability of an event, and let Exp$[\cdot]$ denote the expectation of a random variable. Using $(1+\varepsilon)^x \le 1+\varepsilon x$ for $0 \le x \le 1$ and $1+x \le e^x$ for $x \ge 0$ we compute

$$\begin{aligned}
\mathrm{Prob}[X_1+\cdots+X_k > (1+\varepsilon)\mu] &= \mathrm{Prob}\left[\frac{\prod_{i=1}^k (1+\varepsilon)^{X_i}}{(1+\varepsilon)^{(1+\varepsilon)\mu}} > 1\right] \\
&\le \mathrm{Prob}\left[\frac{\prod_{i=1}^k (1+\varepsilon X_i)}{(1+\varepsilon)^{(1+\varepsilon)\mu}} > 1\right] \\
&< \mathrm{Exp}\left[\frac{\prod_{i=1}^k (1+\varepsilon X_i)}{(1+\varepsilon)^{(1+\varepsilon)\mu}}\right] = \frac{\prod_{i=1}^k (1+\varepsilon\,\mathrm{Exp}[X_i])}{(1+\varepsilon)^{(1+\varepsilon)\mu}} \\
&\le \frac{\prod_{i=1}^k e^{\varepsilon\,\mathrm{Exp}[X_i]}}{(1+\varepsilon)^{(1+\varepsilon)\mu}} = \frac{e^{\varepsilon\mu}}{(1+\varepsilon)^{(1+\varepsilon)\mu}} = e^{-\mu f(\varepsilon)}
\end{aligned}$$

□

Note that $f(\varepsilon) > 0$ for $\varepsilon > 0$. Generalizing and strengthening results of Raghavan [1988], we can now bound the effect of randomized rounding:

Theorem 3.29. *Given numbers $x_{c,b} \ge 0$ for all $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$ with $\sum_{b\in\mathcal{B}_c} x_{c,b} = 1$ for all $c \in \mathcal{C}$. Let $\lambda := \max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} \sum_{b\in\mathcal{B}_c} x_{c,b}(g_c(b))_r$.*

Consider a "randomly rounded" solution, $\hat{b}_c \in \mathcal{B}_c$ for $c \in \mathcal{C}$, given as follows. Independently for all $c \in \mathcal{C}$ we choose $b \in \mathcal{B}_c$ as $\hat{b}_c$ with probability $x_{c,b}$. Let $\hat{\lambda} := \max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(\hat{b}_c))_r$.

For $r \in \mathcal{R}$ let $\rho_r \ge \frac{(g_c(b))_r}{\lambda}$ for all $b \in \mathcal{B}_c$ and $c \in \mathcal{C}$, and let $\rho := \max_{r\in\mathcal{R}} \rho_r$. Let $\Omega := \rho \max\left\{1, \ln\left(\sum_{r\in\mathcal{R}} e^{1-\frac{\rho}{\rho_r}}\right)\right\}$ and $\delta := (\Omega+e-2)\sqrt{\frac{\Omega}{f(\Omega+e-2)}}$.

Then $\hat{\lambda} \le \lambda(1+\delta)$ with positive probability.

Proof. Consider any resource $s \in \mathcal{R}$. Note that $\frac{(g_c(\hat{b}_c))_s}{\rho_s\lambda}$, $c \in \mathcal{C}$, are independent random variables in $[0,1]$. The sum of the expectations of these $|\mathcal{C}|$ random variables is at most $\frac{1}{\rho_s}$. Hence, by Lemma 3.28, their sum is greater than $\frac{1+\delta}{\rho_s}$ with probability less than $e^{-\frac{f(\delta)}{\rho_s}}$.

We will show that $f(\delta) \ge \Omega$. This implies that $\sum_{c\in\mathcal{C}} (g_c(\hat{b}_c))_s > \rho_s\lambda \cdot \frac{1+\delta}{\rho_s} = (1+\delta)\lambda$ with probability less than

$$e^{-\frac{f(\delta)}{\rho_s}} \le e^{-\frac{\rho}{\rho_s}\max\left\{1,\ln\left(\sum\limits_{r\in\mathcal{R}} e^{1-\frac{\rho}{\rho_r}}\right)\right\}} \le e^{-\frac{\rho}{\rho_s}-\max\left\{0,\ln\left(\sum\limits_{r\in\mathcal{R}} e^{-\frac{\rho}{\rho_r}}\right)\right\}} \le \frac{e^{-\frac{\rho}{\rho_s}}}{\sum\limits_{r\in\mathcal{R}} e^{-\frac{\rho}{\rho_r}}}.$$

Considering all resources we then conclude that with positive probability we have $\hat{\lambda} \le (1+\delta)\lambda$.

To show $f(\delta) \geq \Omega$, note that $\varepsilon \mapsto \frac{f(\varepsilon)}{\varepsilon^2}$ is monotonically decreasing for $\varepsilon > 0$ (as observed by Raghavan) and $f(\varepsilon) \geq \varepsilon - e + 2$ for all $\varepsilon \geq 0$. For $\Delta := \Omega + e - 2$ we have $\Omega = \Delta - e + 2 \leq f(\Delta)$ and hence $\delta = \Delta\sqrt{\frac{\Omega}{f(\Delta)}} \leq \Delta$. We conclude that $\frac{f(\delta)}{\delta^2} \geq \frac{f(\Delta)}{\Delta^2}$, i.e., $f(\delta) \geq \delta^2 \frac{f(\Delta)}{\Delta^2} \geq \Omega$. □

In a typical instance there are a few resources r for which ρ_r is 1, but for most resources this value is much smaller. The algorithm can be derandomized by the method of conditional probabilities; we omit the details here.

We remark that another attempt of improving the bound has been made by Leighton et al. [2001], who showed how the (de)randomized rounding can be improved by using an algorithmic version of the Lovász Local Lemma (Erdős and Lovász [1975]), exploiting the fact that most Steiner trees are short and their rounding does not affect regions that are far away.

3.5 Discussion and Outlook

In this chapter we presented an efficient fully polynomial approximation scheme for the MIN-MAX RESOURCE SHARING PROBLEM, providing the best currently known runtime for weak block solvers. We showed how to parallelize our algorithm on shared memory machines with weak memory models, making use of the approximation parameter to allow calculations to be performed on outdated data without affecting correctness, and obtaining very good speedup values over single-threaded execution (see experimental results in chapter 4).

An important direction of future research is to further reduce the runtime dependence on the approximation parameter ε. Bienstock and Iyengar [2006] managed to reduce this dependence to $O(\varepsilon^{-1})$ for fractional packing, and obtaining a similar result for the more general MIN-MAX RESOURCE SHARING PROBLEM would be a significant progress.

Another interesting question with practical relevance is how to obtain guarantees on the number of resources (approximately) attaining λ^* in a solution to the MIN-MAX RESOURCE SHARING PROBLEM generated by a fast combinatorial approximation algorithm, and on the maximum congestion of other resources.

Chapter 4

Global Routing

As described in the introduction, global routing is an abstraction of the actual routing problem in VLSI design to a coarser model of the routing space. The reduced instance sizes allow to perform a global coordination among all nets to find a feasible solution, and to globally optimize various objectives such as total wiring length, manufacturing yield or power consumption. Constraints are given not only by available space, but also by limits on detours, or more generally by delay bounds on groups of nets to avoid estimated signal propagation delay to increase uncontrollably due to detours in routing.

Global Routing largely determines the quality of the final routing solution, so it is a key step in the VLSI design process. The prime objective of course is design closure, i.e. meeting all constraints. In particular, available space in a region must not be exceeded by the wires running through it, including minimum spacing requirements (see figure 4.1 for an example of a *congestion plot* showing space utilization). Fast running times are important especially when global routing is run in early design stages to obtain good estimations on routability and achievable results w.r.t. various quality metrics. The classical metric for judging the quality of a routing solution, still used in most benchmarks, is wiring length. However, because of significant differences in the characteristics of wiring layers, and spacing dependency of the practically relevant objectives power and manufacturing yield, wiring length in most cases provides a very bad estimate on these metrics.

In this chapter we show how yield and power can be optimized directly in global routing using the (PARALLEL) RESOURCE SHARING ALGORITHM presented in chapter 3. While many global routers address optimization objectives heuristically, this algorithm guarantees to find a globally near-optimum solution. After giving a definition of the GLOBAL ROUTING PROBLEM and a survey on previous work on global routing in section 4.1, we show in section 4.2 how a fractional relaxation can be formulated as a RESOURCE SHARING PROBLEM. In section 4.3 we show how the global routing graph, which models routing space in global routing, is constructed, and define the global routing net model in section 4.4. Feasible routings for standard nets in global routing are Steiner forests in the global routing graph, but a more general net model allows to support certain *topology constraints*. We demonstrate in section 4.5 how an

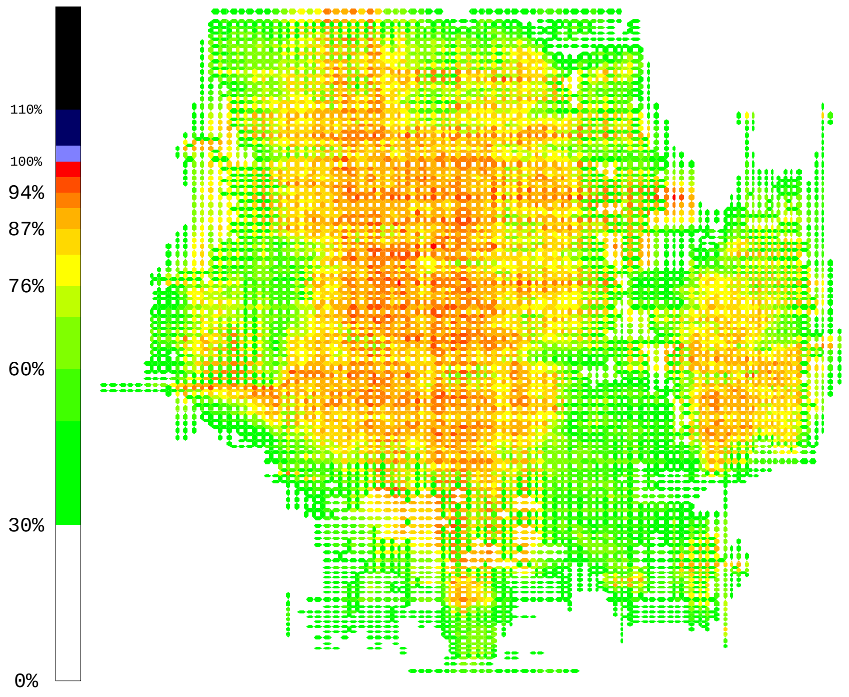


Figure 4.1: Edge capacity congestion map for a global routing instance. Utilization of edge capacities is very low in white and green areas, and close to (but at most) 100 percent in red areas.

important step in hierarchical design methodologies, the *port assignment* at hierarchy boundaries, can be addressed by global routing with topology constraints imposed on nets which cross hierarchy borders. In section 4.6 we present an optimum algorithm for routing a single global routing net under topology constraints. This algorithm is a dynamic programming approach generalizing the DIJKSTRA-STEINER ALGORITHM of Vygen [2001]. It is limited in practice to nets with few terminals, but by far most nets have five terminals or less in typical global routing instances. Section 4.7 discusses various implementation aspects and tradeoffs that have been made in the implementation of the BonnRoute® global router. Section 4.8 proposes an approach to implement a significantly faster block solver that limits the number of bends in a routing for a net. For long connections, this restriction might leave enough flexibility to avoid congested regions without unnecessarily long detours. Finally, we present in section 4.9 experimental results on state-of-the-art industrial chips, demonstrating that our global router can significantly improve on manufacturing yield and power consumption, and that our implementation of the PARALLEL RESOURCE SHARING ALGORITHM scales very well with the number of processors. The results also show that on most chips a feasible integral solution can be generated from a feasible fractional solution by randomized rounding and a simple ripup-and-reroute method. We conclude this chapter in section 4.10 with a discussion and outlook on future challenges in global routing.

4.1 Problem Statement and Overview

In a simplified form, the global routing problem can be stated as follows:

SIMPLIFIED GLOBAL ROUTING PROBLEM

Instance:

1. An undirected graph $G = (V, E)$ with *edge capacities* $u : E \to \mathbb{R}_+$ and *lengths* $l : E \to \mathbb{R}_+$.
2. A set $\mathcal{N}$ of *nets*, and a set $\mathcal{T}_N$ of *feasible Steiner forests* in G for each $N \in \mathcal{N}$.
3. *Wire widths* $w : \mathcal{N} \times E(G) \to \mathbb{R}_+ \setminus \{0\}$, including the minimum spacing required to neighbouring wires.

Task: Find a Steiner forest $T_N \in \mathcal{T}_N$ for each $N \in \mathcal{N}$ such that

$$\sum_{N \in \mathcal{N} : e \in E(T_N)} w(N, e) \leq u(e)$$

for each $e \in E(G)$, and total wire length

$$\sum_{N \in \mathcal{N}} \sum_{e \in E(T_N)} l(e)$$

is minimized, where $E(T_N)$ denotes the set of edges contained in T_N.

The elements of $\mathcal{T}_N$ for $N \in \mathcal{N}$ usually are Steiner trees if each pin is a single point or vertex in G. We deliberately choose a more general formulation here. Specifically, this allows to model larger pins and, depending on the type of pin, to use their metal shapes as “bridges” in the connection of a net. The set $\mathcal{T}_N$ is usually not specified explicitly. In this case it is implicitly given by all forests in G that establish *connectivity* among the pins of net N. We shall consider only routes without cycles, but remark that this would be easy as long as a block solver is available that can optimize over $\mathcal{T}_N$. Global routing nets and the connectivity model are more precisely defined in section 4.4.

We do not make any assumptions about the structure of the global routing graph G here, as this is not important for the abstract problem formulation. However, many global routers, including BonnRoute®, work on a graph with a grid-like structure, as this does not impose practical restrictions in most cases given the prevalent Manhattan routing methodology, and is convenient for a number of reasons that become clear later in this chapter.

The SIMPLIFIED GLOBAL ROUTING PROBLEM as defined above is still a widely adopted formulation of the global routing problem. Many academic global routers solve an even more restricted version where $w(N, e_1) = w(N, e_2)$ for each $N \in \mathcal{N}$ and $e_1, e_2 \in E$. See below for an overview of previous work on global routing.

While minimizing wiring length has been the classical objective in global routing for a long time, manufacturing yield and power consumption are the cost metrics which are important in practice. In both cases, using an edge $e \in E(G)$ by some $N \in \mathcal{N}$ induces costs (measuring defect sensitivity or electrical capacitance, respectively, or a weighted sum of both) which can be reduced by allocating extra space in addition to the minimally required value $w(N,e)$. We therefore consider the following more general problem in this chapter:

GLOBAL ROUTING PROBLEM

Instance:

1. An undirected graph $G = (V,E)$ with *edge capacities* $u : E \to \mathbb{R}_+$.
2. A set $\mathcal{N}$ of *nets*, and a collection $\mathcal{T}_N$ of multisets of edges in E induced by *feasible routings* for each $N \in \mathcal{N}$ under certain topology constraints (see section 4.4).
3. *Wire widths* $w : \mathcal{N} \times E(G) \to \mathbb{R}_+ \setminus \{0\}$, including the minimum spacing required to neighbouring wires.
4. Sets $P_1, \dots, P_q \subseteq \mathcal{N}$ ($q \in \mathbb{N}$) of *critical paths* with delay bounds Γ^{P_i} for each $1 \leq i \leq q$, and convex functions $\gamma^{P_i}_{N,e} : \mathbb{R}_+ \to \mathbb{R}_+$ for each $1 \leq i \leq q$, $N \in \mathcal{N}$ and $e \in E$, with $\gamma^{P_i}_{N,e}(s) = 0$ for each $s \in \mathbb{R}_+$ if $N \notin P_i$.
5. Convex functions $\gamma^{\text{obj}}_{N,e} : \mathbb{R}_+ \to \mathbb{R}_+$ for each $N \in \mathcal{N}$ and $e \in E$.

Task: Find a $T_N \in \mathcal{T}_N$ and extra space assignment $s_N : E(T_N) \to \mathbb{R}_+$ for each $N \in \mathcal{N}$ such that

$$\sum_{N \in \mathcal{N} : e \in E(T_N)} (w(N,e) + s_N(e)) \leq u(e)$$

for each $e \in E(G)$,

$$\sum_{N \in P_i} \sum_{e \in E(T_N)} \gamma^{P_i}_{N,e}(s_N(e)) \leq \Gamma^{P_i}$$

for each $1 \leq i \leq q$, and

$$\sum_{N \in \mathcal{N}} \sum_{e \in E(T_N)} \gamma^{\text{obj}}_{N,e}(s_N(e)) \tag{4.1}$$

is minimized, where $E(T_N)$ denotes the set of edges contained in T_N.

We shall define feasible routings w.r.t. topology constraints precisely in section 4.4. The only relevant fact at this point is that topology constraints might require wires of different "parts" of a net to cross the same edge, hence we need multisets of edges to represent feasible routings for a net under topology constraints.

For each $1 \leq i \leq q$, $N \in \mathcal{N}$ and $e \in E$, the function $\gamma_{N,e}^{P_i}$ specifies for all possible values of extra space how much using edge e by net N with the corresponding extra space contributes to total delay of path P_i. This assumes that the delay contributions of the edges are independent from each other and from routing topology, which of course is a simplification. Note that for $N \in P_i \cap P_j$ with $i \neq j$, $\gamma_{N,e}^{P_i}$ and $\gamma_{N,e}^{P_j}$ can be different because of slew effects.

Similarly, the *objective cost functions* $\gamma_{N,e}^{\text{obj}}$ for each $N \in \mathcal{N}$ and $e \in E$ specify the spacing-dependent contribution to the total objective value that is to be minimized. By setting $\gamma_{N,e}^{\text{obj}}(\sigma) := l(e)$ for each $N \in \mathcal{N}$, $e \in E$ and $\sigma \geq 0$, total wiring length is minimized, but the more general formulation of the GLOBAL ROUTING PROBLEM also includes optimization of manufacturing yield and minimization of power consumption. In both cases the objective cost functions are roughly $x \mapsto c_1 + \frac{c_2}{x}$ for constants $c_1, c_2 > 0$ and extra space $x \geq 0$. We show in detail in section 4.2.1 how to define objective cost functions for directly optimizing manufacturing yield in global routing.

4.1.1 Previous Work

Automated routing solutions have a long history, and so do global routing algorithms. Many algorithms that have been applied to global routing are based on heuristics, the most popular approaches being ripup-and-reroute methods that iteratively refine an initial solution until a feasible solution has been found, or a certain objective cannot be further improved. An early and influential ripup-and-reroute algorithm is described by Ting and Tien [1983].

The first to consider global routing as a multi-commodity flow problem were Shragowitz and Keel [1987]. Their algorithm also is an iterative refinement method, but they show that it is guaranteed to find a feasible solution if one exists. No guarantees are given though on running time and the achieved objective value.

Carden, Li and Cheng [1996] applied the approximation algorithm of Shahrokhi and Matula [1990] for the maximum concurrent flow problem to global routing, providing an approximate optimality guarantee for the first time. They solve a fractional relaxation of the SIMPLIFIED GLOBAL ROUTING PROBLEM in which each net is to be routed by a convex combination of feasible Steiner forests. At the end, an integral solution is generated from a provably near-optimum fractional solution by a *randomized rounding* technique which was first proposed by Raghavan and Thompson [1987, 1991]. With high probability, congestion increases only within a certain bound when applying this procedure, and the resulting violations of capacity constraints can easily be eliminated by a conventional ripup-and-reroute method.

Building on Carden, Li and Cheng [1996], Hong et al. [1997] incorporate delay budgets on critical paths. Although the authors claim to generate an $(1+\varepsilon)$-optimal solution for each $\varepsilon > 0$, this optimality guarantee refers only to edge capacity constraints, as delay budgets are distributed heuristically among the nets belonging to a critical path in the course of the algorithm.

Albrecht [2001a,b] applied the simpler and faster multicommodity-flow approximation algorithm of Garg and Könemann [2008] to the SIMPLIFIED GLOBAL ROUTING PROBLEM, allowing for the first time to give approximation guarantees on huge designs, which means between half a million and a million nets according to the standards at that time. This algorithm was targeted to optimize wiring length and did not support timing constraints. It was extended by Vygen [2004] to incorporate coupling, delay bounds and power consumption, and subsequently used by Müller [2006a] to optimize manufacturing yield. However, objective cost functions in these works are restricted to be linear.

Hu and Sapatnekar [2001] give a survey of global routing algorithms published until 2001, including many approaches that we did not mention here, such as hierarchical global routing. The book of Saxena, Shelar and Sapatnekar [2007] treats routing congestion in more generality, including the influences on congestion by early design stages such as logic synthesis and floorplanning, and interconnect planning, which means routing in conjunction with buffer insertion.

Moffitt, Roy and Markov [2008] give a survey on recently developed global routing approaches. Much of the research done in the last years has been stimulated by the ISPD global routing contest [2007, 2008]. The routing instances which have been made publicly available in this contest are anonymized and, as it seems, considerably simplified versions of industrial designs. Particularly, the wires of each net consume the same amount of routing space (metal width plus minimum spacing required to neighbouring wires) on each layer, and capacities of via edges are assumed to be infinite. The optimization objective is total wiring length, where all via edges are assigned the same length value $l \in \mathbb{N}$. Not surprisingly, with all layer characteristics being equal, most recent academic global routers solve the original problem by merging all layers with horizontal and vertical preferred direction, respectively, into one layer and subsequently perform a *layer assignment* step as described by Lee and Wang [2008, 2009], for example. Although the resulting "projected" problem instance comprises two layers and is not planar, it is commonly called a *2D global routing problem*, while the original problem is called a *3D global routing problem*.

All recently published global routing algorithms contain heuristic elements in central components and thus are far from providing global optimality guarantees. Still, empirically many of them prove to work well based on the results on publicly available benchmarks.

NTHU-Route, developed by Chang, Lee and Wang [2008] and winner of the ISPD global routing contest [2008], employs various techniques, ranging from low-effort pattern routing to close almost all connections in uncongested parts of a design, over iterative shifting of tree segments and monotonic routing (restricted to stair-case-like paths computed by dynamic programming), to computationally more expensive ripup-and-reroute based on computing shortest paths w.r.t. heuristically chosen edge costs. These costs contain an objective cost (in this case: wiring length and via count) component, and a congestion component that takes into account the history of congestion on each edge. Such heuristics are often called *history based* or *negotiation based* routing, and different variations are applied by most recently proposed global routing methods. These heuristics can be viewed as Lagrangean relaxation methods in the sense that they relax capacity constraints and penalize violations, but we are not aware of any global router for which convergence was proven given the step sizes used for updating Lagrange multipliers. NTHU-Route first solves the projected 2D problem and then generates a solution to the original 3D instance by the layer assignment method of Lee and Wang [2008]. MaizeRouter, winner of the 3D category of the ISPD global routing contest [2007] and developed by Moffitt [2008], mainly relies on an iterative edge shifting method. No details are given however on the cost function used to evaluate a candidate edge shift operation. FastRoute (Xu, Zhang and Chu [2009]) employs similar techniques as NTHU-Route, but extends the pattern routing part slightly to allow routes with at most 3 bends in order to resolve a significant part of congestion problems without resorting to shortest-paths computations. Other successful contestants include FGR ("Fairly Good Router") by Roy and Markov [2008], Archer (Ozdal and Wong [2009]) and BoxRouter (Cho et al. [2009a]), all containing a variety of methods similar to those mentioned above.

FGR is the only among the global routers that took part in the ISPD contest which additionally to a 2D routing approach followed by layer assignment offers a "full 3D" mode to route directly in the original graph. Roy and Markov [2008] observe considerably higher runtimes with full 3D routing, but better wiring length and via count results than by 2D routing on the easier routing instances of the contest, however at the cost of finding a feasible solution on much fewer routing instances.

BoxRouter, in contrast to the others, uses a progressive integer linear program approach to perform layer assignment. This can be thought of as a greedy method which assigns wire segments to layers within a small area ("box") of the chip, minimizing the number of vias by solving an integer linear program of tractable size, and then iteratively extending the box. In each iteration, the solution from the last iteration is fixed and augmented to a solution for the larger box, again by solving a relatively small integer linear program, until finally all wire segments have been assigned.

Wu, Davoodi and Linderoth [2009] also use integer linear programming, but follow a different strategy. Similar to multi-commodity flow based global routing algorithms, they generate a fractional solution by iteratively finding least-cost routes for each net w.r.t. some costs (not giving an approximation guarantee, however). Then they solve mixed integer-linear programs using CPLEX on small parts of a chip to generate an integral solution from the fractional solution.

4.2 Application of the Resource Sharing Algorithm to VLSI Global Routing

We first consider the SIMPLIFIED GLOBAL ROUTING PROBLEM. Solving this problem is hard since it contains the edge-disjoint paths problem. Relaxing integrality constraints, i.e. allowing convex combinations of feasible Steiner forests for each net, is a standard way to simplify the problem. The relaxed problem can be formulated as a RESOURCE SHARING PROBLEM as follows.

First the objective function is replaced by a constraint, imposing an upper bound Γ on $\sum_{N\in\mathcal{N}}\sum_{e\in E(T_N)} l(e)$ (binary search can be applied to find the optimum value of Γ). Let $E^+ := \{e_1,\dots,e_m\} \subseteq E(G)$ be the edges with positive edge capacity. We set $\mathcal{C} := \mathcal{N}$, $\mathcal{R} := \{1,\dots,m+1\}$ and

$$\mathcal{B}_c := \operatorname{conv}\left(\left\{\chi(T) \,:\, T \in \mathcal{T}_c \text{ with } |E(T)\cap(E(G)\setminus E^+)| \text{ minimal}\right\}\right) \subseteq \mathbb{R}^m_+$$

for each $c \in \mathcal{C}$, where $\chi(T)$ is the edge-incidence vector of a Steiner forest T with entries for edges in $E(G)\setminus E^+$ removed. If there is a net $N \in \mathcal{N}$ with

$$\min\{|E(T)\cap(E(G)\setminus E^+)| : T \in \mathcal{T}_N\} > 0, \tag{4.2}$$

there is of course no feasible solution to the SIMPLIFIED GLOBAL ROUTING PROBLEM. Because usually there are only very few such nets, and this is almost always due to local modeling inaccuracies (see section 4.4.1), one is often interested in a near-optimum global routing which uses as few zero capacity edges as possible and is feasible w.r.t. all other constraints. Finally, we set

$$g_c(b) := \left(\frac{b_1 w(c,e_1)}{u(e_1)},\dots,\frac{b_m w(c,e_m)}{u(e_m)},\frac{1}{\Gamma}\left(\sum_{i=1}^{m} b_i l(e_i)\right)\right)$$

for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$.

Since $\min_{b\in\mathcal{B}_c} \omega^\top g_c(b)$ is always attained by some element of $\{\chi(T) \,:\, T \in \mathcal{T}_c\}$ for a given $c \in \mathcal{C}$ and $\omega \in \mathbb{R}^{\mathcal{R}}_+$, the oracle can be implemented by any algorithm for solving the (group) Steiner tree problem in graphs (exact or approximate). Note that the number of terminals is small for most nets, and hence using an exact algorithm (i.e., $\varepsilon_0 = 0$) is not impossible.

For obtaining an integral solution, i.e. a Steiner forest for each $c \in \mathcal{C}$, one can pick one of the $b \in \mathcal{B}_c$ with positive coefficient $x_{c,b}$ as set by the RESOURCE SHARING ALGORITHM (see chapter 3) because the oracle returns only incidence vectors of Steiner forests. In section 3.4 we saw that by picking $b_c \in \mathcal{B}_c$ with probability x_{c,b_c} for each customer, respectively, the maximum relative resource utilization does not increase too much. In practice, only few violations occur, i.e. $\lambda_r > 1$ only for a small number of resources $r \in \mathcal{R}$, and these violations can easily be eliminated by iteratively replacing the Steiner forests for some of the nets ("ripup and reroute").

We now show that also the general GLOBAL ROUTING PROBLEM defined in section 4.1 can be formulated as a RESOURCE SHARING PROBLEM. First, feasible routings for a net in this setting are more general than Steiner forests because of topology constraints, and correspond to multisets of edges in $E(G)$. Let therefore $\chi(T) \in \mathbb{Z}_+^{E^+}$ be the vector of edge multiplicities in a multiset $T \in \mathcal{T}_N$ corresponding to a feasible routing for $N \in \mathcal{N}$, with entries for edges in $E(G) \setminus E^+$ removed (actually feasible routings are sets of trees whose unions are not necessarily forests, see the definition of the global routing net model in section 4.4).

Again, we set $\mathcal{C} := \mathcal{N}$. As for the SIMPLIFIED GLOBAL ROUTING PROBLEM, we impose some bound Γ^{obj} on the total cost instead of minimizing the objective function directly. In addition to the edge capacity resources and the resource for modeling the objective function, we introduce a resource for each of the critical paths $P_1, \ldots, P_q \subseteq \mathcal{C}$, so $\mathcal{R} := \{1, \ldots, m+q+1\}$. For running the RESOURCE SHARING ALGORITHM, we set $\mathcal{B}_c := \operatorname{conv}\left(\mathcal{B}_c^{\text{int}}\right)$ with

$$\mathcal{B}_c^{\text{int}} := \left\{ \begin{pmatrix} \chi(T) \\ s \end{pmatrix} \in \mathbb{R}_+^{2m} : \begin{array}{l} T \in \mathcal{T}_c \text{ with } |E(T) \cap (E(G) \setminus E^+)| \text{ minimal}, \\ s \geq 0,\, s_i = 0 \text{ for } e_i \notin E(T) \end{array} \right\}.$$

The resource consumption functions are defined as follows:

$$g_c(b) := \begin{pmatrix} (b_1 w(c,e_1) + b_{m+1})/u(e_1) \\ \vdots \\ (b_m w(c,e_m) + b_{2m})/u(e_m) \\ \left(\sum_{1 \leq i \leq m : b_i > 0} b_i \gamma_{c,e_i}^{P_1}(b_{m+i}/b_i)\right)/\Gamma^{P_1} \\ \vdots \\ \left(\sum_{1 \leq i \leq m : b_i > 0} b_i \gamma_{c,e_i}^{P_q}(b_{m+i}/b_i)\right)/\Gamma^{P_q} \\ \left(\sum_{1 \leq i \leq m : b_i > 0} b_i \gamma_{c,e_i}^{\text{obj}}(b_{m+i}/b_i)\right)/\Gamma^{\text{obj}} \end{pmatrix} \tag{4.3}$$

for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$. The structure of these resource consumption functions allows to optimize over $\mathcal{B}_c^{\text{int}}$ instead of $\mathcal{B}_c$ for each $c \in \mathcal{C}$:

Lemma 4.1. *Let $c \in \mathcal{C}$ and some price vector $\omega \in \mathbb{R}_+^{\mathcal{R}}$ be given. Then* $\inf_{b \in \mathcal{B}_c^{\text{int}}} \omega^\top g_c(b) = \inf_{b \in \mathcal{B}_c} \omega^\top g_c(b)$.

Proof. Assume there is a $b^* \in \mathcal{B}_c$ with $\omega^\top g_c(b^*) < \inf_{b \in \mathcal{B}_c^{\text{int}}} \omega^\top g_c(b)$. We have $b^* = \sum_{1 \leq j \leq k} \lambda_j b^j$ for some $b^1, \ldots b^k \in \mathcal{B}_c^{\text{int}}$, $\lambda_1, \ldots, \lambda_k \in \mathbb{R}_+ \setminus \{0\}$ with $\sum_{1 \leq j \leq k} \lambda_j = 1$, and $k \in \mathbb{N}$.

For $1 \leq j \leq k$, let $\bar{b}^j \in \mathbb{R}_+^{2m}$ with $\bar{b}_i^j := b_i^j$ for each $1 \leq i \leq m$ and

$$\bar{b}_{m+i}^j := \begin{cases} b_i^j b_{m+i}^* / b_i^* & : \ b_i^* > 0 \\ 0 & : \ b_i^* = 0 \end{cases}$$

Then $b^* = \sum_{1 \leq j \leq k} \lambda_j \bar{b}^j$. Using the fact that $g_c(b)_r$ as defined in (4.3) is linear for all $b \in \mathcal{B}_c$ and $1 \leq r \leq m$, and

$$\frac{\bar{b}_{m+i}^j}{\bar{b}_i^j} = \frac{b_{m+i}^*}{b_i^*}$$

for $1 \le j \le k$ and $1 \le i \le m$, we get $\omega^\top g_c(b^*) = \sum_{1\le j\le k} \lambda_j \omega^\top g_c(\bar{b}^j)$. Therefore we have $\omega^\top g_c(\bar{b}^j) \le \omega^\top g_c(b^*) < \inf_{b\in\mathcal{B}_c^{\mathrm{int}}} \omega^\top g_c(b)$ for at least one $1 \le j \le k$, which contradicts the assumption since $\bar{b}^j \in \mathcal{B}_c^{\mathrm{int}}$. □

Finding a $b^* \in \mathcal{B}_c^{\mathrm{int}}$ with $\omega^\top g_c(b^*) \le (1+\varepsilon_0)\inf_{b\in\mathcal{B}_c^{\mathrm{int}}} \omega^\top g_c(b)$ is equivalent to computing $\bar{\omega} := (\bar{\omega}_1, \ldots, \bar{\omega}_m)$ with

$$\bar{\omega}_i := \inf_{s_i \ge 0} \left(\omega_i \frac{w(c,e_i)+s_i}{u(e_i)} + \sum_{1\le j\le q} \omega_{m+j} \frac{\gamma_{c,e_i}^{P_j}(s_i)}{\Gamma^{P_j}} + \omega_{m+q+1} \frac{\gamma_{c,e_i}^{\mathrm{obj}}(s_i)}{\Gamma^{\mathrm{obj}}} \right) \tag{4.4}$$

for $1 \le i \le m$, and to find a $T^* \in \mathcal{T}_c$ with $\bar{\omega}^\top \chi(T^*) \le (1+\varepsilon_0)\min_{T\in\mathcal{T}_c} \bar{\omega}^\top \chi(T)$. Therefore, if feasible routings for $c \in \mathcal{C}$ are Steiner trees, a graph Steiner tree algorithm can be used to implement the oracle function f_c, and the GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM presented in section 4.6 can be used for the general case.

4.2.1 Yield Model

Defects occuring in the manufacturing process of a chip have a variety of reasons which contribute to *yield loss*. While some of these, e.g. wafer mishandling or mask misalignment, are independent of the physical design of a circuit, and can be well controlled in mature production processes, there is a class of defects caused by small particles which cannot be entirely eliminated and deposit on the wafer or on the photo masks (partly, these particles actually originate from the wafer being processed). In a simplified view, there are defects which cause metal to be replaced by insulator material (called "open" or "missing material" defects) or vice versa (called "short" or "extra material" defects). See figure 4.2 for an illustration.

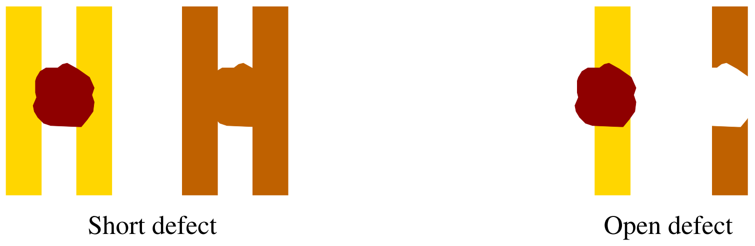


Figure 4.2: Short (extra material) and open (missing material) defects, and resulting damages to the chip layout ("faults")

Of course not every defect causes a chip to fail: Focusing on routing, only extra metallization which shorts wires of two different nets, or missing material defects that actually disconnect a wire make a chip unusable. We therefore distinguish between defects and faults caused by defects. Given a distribution of defect sizes and locations, the layout of the wiring therefore can influence the expected yield loss due to uncontrollable particles in the production process. We make some simplifications in the following:

First, we assume particles to be of quadratic shape, and second we assume a defect to cause a fault only if it entirely disconnects a wire, or *intersects* two wires belonging to different nets (instead of only causing minimum distance violations). A widely used defect size distribution is

$$f(r) := \begin{cases} 0, r < r_0 \\ \frac{\xi}{r^3}, r \geq r_0 \end{cases}$$

for some $r_0 \in \mathbb{R}_+$ smaller than the smallest possible particle that can cause a fault, and ξ such that $\int_0^\infty f(r)\mathrm{d}r = 1$. We also use this defect size distribution, and assume the spatial distribution of defects to be uniform. The *critical area* or *defect sensitivity* w.r.t. short-defects on a wiring or via plane p of a chip is

$$C^p_{\text{short}} := \kappa^p_{\text{short}} \int_x \int_y \int_{t_{\text{short}}(x,y,p)}^\infty f(r)\,\mathrm{d}r\,\mathrm{d}y\,\mathrm{d}x, \tag{4.5}$$

where $t_{\text{short}}(x,y,p)$ is the smallest size of a particle centered at location (x,y,p) that causes a short-defect, and κ^p_{short} is a weighting factor provided by manufacturing that encodes the relative probability of short defects on plane p. Similarly,

$$C^p_{\text{open}} := \kappa^p_{\text{open}} \int_x \int_y \int_{t_{\text{open}}(x,y,p)}^\infty f(r)\,\mathrm{d}r\,\mathrm{d}y\,\mathrm{d}x, \tag{4.6}$$

where $t_{\text{open}}(x,y,p)$ is the smallest size of a particle centered at location (x,y,p) that causes an open-defect, and κ^p_{open} again is a parameter provided by manufacturing. A simple and widely used model based on the Poisson distribution estimates the yield loss to be $1-e^{-C}$, where $C := \sum_{p\in P}(C^p_{\text{short}} + C^p_{\text{open}})$ is the total critical area summed over all planes. There are also other models which are not restricted to uniform spatial defect distributions; we do not go into details and refer to Koren and Koren [1998] for an overview and further references. Here we focus on minimizing critical area. In the rest of this chapter, we therefore synonymously speak of yield optimization and critical area minimization.

For simple situations like a straight wire without neighbours or straight wires running in parallel, critical area can be computed analytically. For yield optimization in global routing, we can compute spacing-dependent costs as follows: For a wire of width w in plane p surrounded by neighbouring wires at distance d on both sides, we define the contribution of this wire to critical area by integrating (4.5) and (4.6) over the Voronoi region of this wire, i.e. the set of points in plane p where this wire is the closest object. Ignoring wire ends and assuming $r_0 < \min\{\frac{w}{2}, \frac{d}{2}\}$, we thus get a contribution of

$$\begin{aligned} c_{\text{open}}(w,d) &:= 2\xi \int_0^{\frac{d+w}{2}} \int_{x+\frac{w}{2}}^\infty \frac{1}{r^3}\,\mathrm{d}r\,\mathrm{d}x \\ &= 2\xi \int_0^{\frac{d+w}{2}} \frac{2}{(2x+w)^2}\,\mathrm{d}x \\ &= 2\xi \left(\frac{1}{w} - \frac{1}{d+2w}\right) \end{aligned}$$

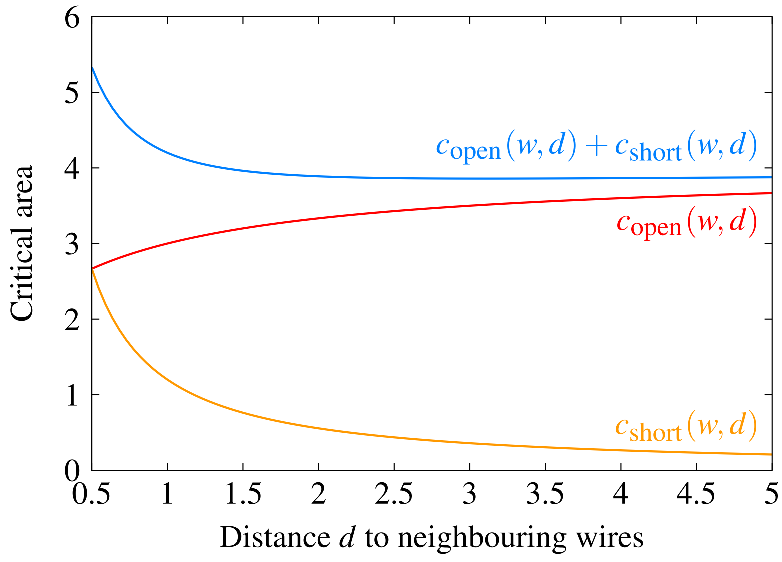


Figure 4.3: Critical area contribution of a unit-length piece of wire of width $w := 0.5$ for unit weights κ_{open} and κ_{short} (scaled by ξ^{-1}).

to open critical area for a unit-length piece of wire if $\kappa^p_{\text{open}} = 1$. Similarly, for $\kappa^p_{\text{short}} = 1$ the contribution to short critical area is

$$c_{\text{short}}(w,d) \quad := \quad 2\xi\left(\frac{1}{d} - \frac{1}{w+2d}\right).$$

Fig. 4.3 shows $c_{\text{open}}(w,d)$, $c_{\text{short}}(w,d)$ and $c_{\text{open}}(w,d)+c_{\text{short}}(w,d)$ for $w = 0.5$ length units and d from 0.5 to 5 length units (assuming a required minimum space equal to the wire width). Apparently, if short critical area is not weighted considerably higher than open critical area, not much can be gained in the sum of both by having more than 3 or 4 times the wire width of extra space in addition to minimum spacing.

Based on the above calculations, we define the spacing dependent costs $\gamma^{\text{obj}}_{N,e} : \mathbb{R}_+ \to \mathbb{R}_+$ as follows for net $N \in \mathcal{N}$ and $e \in E(G)$. Let $l(e)$ be the length of edge e, p the (wiring or via) plane of e, and $d_{N,e}$ the minimum required spacing of a wire of net N crossing edge e. Then we set

$$\begin{aligned}\gamma^{\text{obj}}_{N,e}(s) := l(e) \;\; \big(&\kappa^p_{\text{open}} c_{\text{open}}(w(N,e) - d_{N,e},\; d_{N,e} + s) \\ + \;\; &\kappa^p_{\text{short}} c_{\text{short}}(w(N,e) - d_{N,e},\; d_{N,e} + s)\big)\end{aligned}$$

(note that $w(N,e)$ includes the minimum required spacing for net $N \in \mathcal{N}$ on edge $e \in E(G)$). In our previous approach to yield optimization based on linear programming (cf. Müller [2006a]), $\gamma^{\text{obj}}_{N,e}$ had to be linearized; this is not necessary any more. Note that realizing the global routing solution in detailed routing requires that all wires which have been assigned extra spacing are put next to each other, which might not always be possible.

4.3 Global Routing Graph

For most parts of the global routing module in BonnRoute® it does not matter whether the graph has a grid-like structure as in definition 4.2 below. There are two areas however where this does play a role, or can potentially play a role, respectively:

1. In capacity estimation (see section 4.3.2), BonnRoute® relies on a graph structure based on an axis-parallel tiling. More elaborate methods might have to be applied in case of a less regular graph structure to map between global routing edge capacities and the underlying detailed routing space.

2. The block solver might rely on a regular graph structure. The standard block solver employed in BonnRoute® (see section 4.7.3) does not, but in section 4.8 we propose a data structure which relies on a grid-like graph structure and can be used to implement a fast heuristic block solver.

BonnRoute® defines and uses a *global routing grid graph* with the following structure:

Definition 4.2. *A* global routing grid graph *is a graph $G := (V,E)$ with vertex set*

$$V := \{0,\dots,x_{\max}\} \times \{0,\dots,y_{\max}\} \times \{0,\dots,z_{\max}\}$$

$(x_{\max}, y_{\max}, z_{\max} \in \mathbb{N})$ and edges

$$E := E_{via} \cup \left(\bigcup_{0 \le z \le z_{\max}} E_z \right),$$

where either $E_z = \{\{(x,y,z),(x+1,y,z)\} : (x,y,z),(x+1,y,z) \in V\}$ (called horizontal wiring edges*) or $E_z = \{\{(x,y,z),(x,y+1,z)\} : (x,y,z),(x,y+1,z) \in V\}$ (called* vertical wiring edges*) for $0 \le z \le z_{\max}$, and $E_{via} := \{\{(x,y,z),(x,y,z+1)\} : (x,y,z),\ (x,y,z+1) \in V\}$ (called* via edges*).*

The numbers $0 \le z \le z_{\max}$ are called layers*. If E_z contains horizontal edges ($0 \le z \le z_{\max}$), we say that layer z has* horizontal preferred direction *or that it is a* horizontal layer*, otherwise we say it has* vertical preferred direction *or that it is a* vertical layer*.*

A vertex $(x,y,z) \in V$ is called a vertex on layer z, and $\{(x,y,z),(x',y',z)\} \in E$ is called an edge on layer z ($0 \le x,x' \le x_{\max}$, $0 \le y,y' \le y_{\max}$ and $0 \le z \le z_{\max}$).

Some parts of BonnRoute® can however also work on general graphs. We shall make use of this later. Of course we need at least one horizontal and one vertical layer. As mentioned in chapter 2, horizontal and vertical layers alternate in practice to avoid electrical crosstalk problems between wires running on neighbouring layers, but this is not important in the following. Note that a global routing graph contains only edges running in preferred direction within each layer because jog edges in the global routing graph would be fairly long and thus block too many wires running in preferred direction.

Remark. For an efficient implementation of the block solver, lower bounds on the cost for completing a partial routing to a feasible routing for a net are important, see section 4.7.3. Often, reasonably good lower bounds can be obtained fast from L_1-distances between points in $\mathbb{R}^3$ associated with graph vertices. A global routing grid graph provides a natural such mapping by taking the center of a tile as the position associated with the corresponding vertex.

4.3.1 Construction of the Global Routing Grid Graph

A global routing grid graph can be imagined to result from the track graph used for detailed routing (definition 2.17 on page 20) by deleting jog edges and then contracting sets of vertices in the same region and on the same layer, where regions are defined by an axis-parallel rectangular tiling of the chip area $\mathcal{A}$:

Definition 4.3. *Let* $G_T := (V,E)$ *be a track graph for a chip with area* $\mathcal{A}$*, and* $X := \{x_1,\dots,x_{|X|}\}$ *and* $Y = \{y_1,\dots,y_{|Y|}\}$ cutlines *with* $\min x(\mathcal{A}) = x_1 < \dots < x_{|X|} = \max x(\mathcal{A})$ *and* $\min y(\mathcal{A}) = y_1 < \dots < y_{|Y|} = \max y(\mathcal{A})$*. Let* $p_{\max} := \max\{p : (x,y,p) \in V\}$ *and* $z_{\max} := p_{\max}/2$ *(note that wiring planes in* G_T *have even numbers).*

Then for $1 \le i < |X|$*,* $1 \le j < |Y|$ *and* $0 \le l \le z_{\max}$ *the vertex set*

$$B_{i,j,l} := \bigcup_{\substack{(x,y,2l)\in V \\ x_i \le x < x_{i+1} \\ y_j \le y < y_{j+1}}} (x,y,2l)$$

is called global routing tile *with index* (i,j,l).

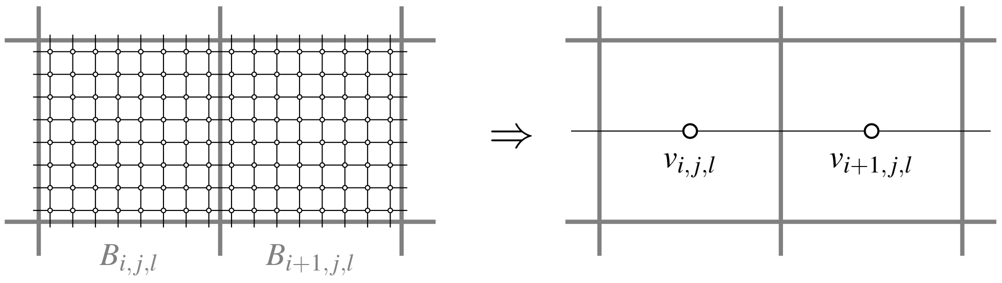


Figure 4.4: Contraction of vertex sets defined by global routing tiles in the track graph to vertices of a global routing grid graph.

A global routing grid graph is then constructed from a track graph by contracting the vertex sets defined by global routing tiles, as illustrated in figure 4.4.

4.3.2 Capacity Estimation

Each edge in the global routing graph must be assigned a capacity value that specifies the available space for routing wires between the tiles corresponding to the vertices it connects. If there are no obstacles, e.g. blockages, pins or wires existing already in the input, the tile width orthogonal to the preferred direction can be used as capacity value, and each wire consumes an amount corresponding to its *effective width* of the available space. The effective width includes the metal width and minimum distance requirements of the wire. In case of uniform track-to-track distances it is advantageous to round up the effective width to the next integer multiple of this distance on the corresponding plane.

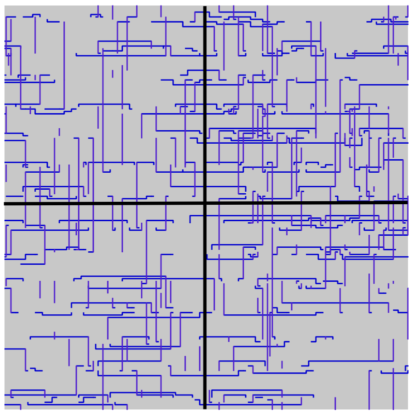


Figure 4.5: A window of 2×2 tiles showing a wiring of some “short” nets.

On a typical chip, there are many connections among pins of a net intersecting the same tile. The routing resources required for these connections are not considered in global routing because these pins result in intersecting global routing pins (see section 4.4.1). Figure 4.5 shows such short connections (including some short connections crossing tile boundaries). The amount of routing resources required for these connections is hard to estimate a priori, so it is natural to preroute these connections and then ask for routing capacities which remain for all other connections.

However, with obstacles that do not block whole tracks as e.g. power rails usually do, determining good edge capacities that do not over- or underestimate the value that can be realized by detailed routing is a more difficult task: As the number of wires routed between two neighbouring global routing tiles can be increased by using other layers, it is natural to formulate the capacity estimation problem as an integer multi-commodity flow problem, as figure 4.6 illustrates.

As even the integer two-commodity flow problem with planar supply graphs is NP-hard (see Müller [2006b] and Naves [2009], and Sebő and Naves [2008] for a survey), there is no hope to solve millions of problem instances optimally in short time, even though only a few hundred paths are to be found in each instance. Müller [2002] shows how to find good approximations very efficiently in practice by a fast bit-pattern based path augmentation procedure.

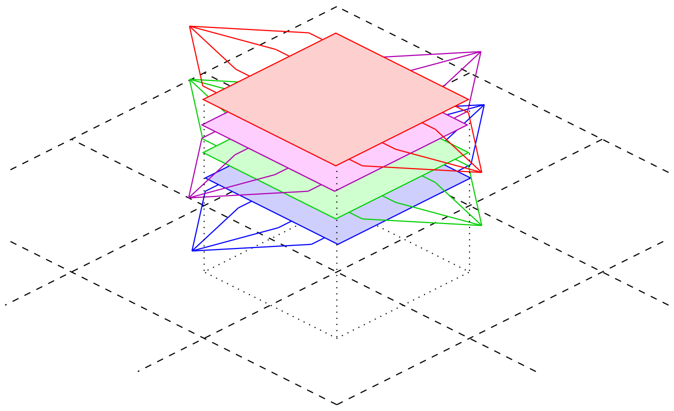


Figure 4.6: Capacity estimation by (approximately) solving a maximum multicommodity flow problem. Each layer makes up a commodity, and the flow value obtained for a commodity is distributed to two neighbouring edges in the corresponding layer.

4.4 Global Routing Net Model

Definition 4.4. *Let $G := (V,E)$ be an undirected graph. A* global routing pin *in G is an element of $2^V \times \{hard, soft\} \times \mathbb{N}$, i.e. a set of vertices in G with an attribute "hard" or "soft", and a number used to identify the pin. For convenience, we write $V(p) := A$ for $p = (A, \alpha, n)$ and say that p* is hard *or* soft *if $\alpha = hard$ or $\alpha = soft$, respectively. A* global routing net *N in G is a set of global routing pins in G.*

In this chapter, if not otherwise stated, a *pin* means a global routing pin and not a real physical pin. A global routing pin is an abstraction of one or more physical pins, as will be explained in detail below.

We require in the following that two global routing pins have different identifier numbers, so a *set* of global routing nets can have two elements that would be identical if stripping the identifiers off the pins. Also, sometimes we will consider sets of pins belonging to different nets and then have to distinguish pins covering the same vertex sets. In most places however we shall not need the pin identifier.

In the SIMPLIFIED GLOBAL ROUTING PROBLEM, feasible solutions for a net $N \in \mathcal{N}$ are defined by feasible Steiner forests:

Definition 4.5. *Let G be an undirected graph, and N a global routing net in G. A* feasible Steiner forest *for N in G is a subgraph $F := (V', E')$ of G without cycles such that*

$$\left(V', E' \cup \{\{v_1, v_2\} : v_1, v_2 \in p \in N, p \text{ is hard}\}\right)$$

is connected, and V' intersects each pin of N.

In the general GLOBAL ROUTING PROBLEM formulation, we allow topology restrictions (defined below), and therefore need a more general connectivity model. In the following, a tree T in an undirected graph with some distinguished root vertex $\text{root}(T) \in V(T)$ is called a rooted tree, and $\text{subtree}(T,v)$ denotes the subtree of T which contains all vertices w for which the unique w-$\text{root}(T)$-path in T contains v, and $\text{root}(\text{subtree}(T,v)) := v$. For $w \in V(T) \setminus \{\text{root}(T)\}$, $\text{father}(w,T)$ is the unique vertex adjacent to w in a w-$\text{root}(T)$-path in T.

Because of topology restrictions, *feasible routings* will be collections of trees whose unions are not necessarily trees:

Definition 4.6. *Let G be an undirected graph. A* tree hierarchy *in G is a pair $(\mathcal{T},H)$, where $\mathcal{T}$ is a collection of rooted trees in G numbered by an injective function $idx : \mathcal{T} \to \mathbb{N}$, and H is a rooted tree on $\mathcal{T}$ which spans $\mathcal{T}$, such that $root(T) \in V(T')$ if $father(T,H) = T'$. $idx(\mathcal{T})$ is the image of idx, and we write $\mathcal{T}[i]$ to denote the tree $T \in \mathcal{T}$ with $idx(T) = i$.*

We define subhierarchies in a natural way:

Definition 4.7. *Let $(\mathcal{T},H)$ be a tree hierarchy, $i \in idx(\mathcal{T})$ and $v \in V(\mathcal{T}[i])$. Let $J \subseteq idx(V(subtree(H,\mathcal{T}[i]))) \setminus \{i\}$ maximal with $idx(father(\mathcal{T}[j],H)) \in \{i\} \cup J$ for all $j \in J$, and $root(\mathcal{T}[j]) \in V(subtree(\mathcal{T}[i],v))$ for all $j \in J$ with $father(\mathcal{T}[j],H) = \mathcal{T}[i]$.*

The subhierarchy *of $(\mathcal{T},H)$* rooted at (i,v) *is the tree hierarchy $(\mathcal{T}',H')$ with*

$$\mathcal{T}' = \{subtree(\mathcal{T}[i],v)\} \cup \bigcup_{j \in J} \mathcal{T}[j]$$

and the numbering $idx' : \mathcal{T}' \to \mathbb{N}$ inherited from $(\mathcal{T},H)$, i. e. $idx'(\mathcal{T}'[j]) = j$ for $j \in J$ and $idx'(\{subtree(\mathcal{T}[i],v)\}) = i$, and H' is the subtree of H induced by $\{\mathcal{T}[j] : j \in J \cup \{i\}\}$ with $\mathcal{T}[i]$ replaced by $subtree(\mathcal{T}[i],v)$.

See fig. 4.7 for an example of a tree hierarchy and subhierarchies of it. Feasible routings for global routing nets N with $|V(p)| = 1$ for all hard pins $p \in N$ are defined as follows:

Definition 4.8. *Let G be an undirected graph, N a set of global routing pins in G with $|V(p)| = 1$ for each hard pin $p \in N$, and $\tau : V(G) \to 2^{(2^N)}$* topology restrictions *for N with $N'' \in \tau(v)$ for each $N'' \subseteq N' \in \tau(v)$ and $v \in V(G)$. A tree hierarchy $(\mathcal{T},H)$ is a* feasible routing *for (N,τ) iff there is a "connected-by" mapping $f : N \to idx(\mathcal{T})$ such that the following conditions are satisfied:*

i) $V(p) \cap V(\mathcal{T}[f(p)]) \neq \emptyset$ for all $p \in N$.

ii) For any $i \in idx(\mathcal{T})$ and $v \in V(\mathcal{T}[i])$, let $(\mathcal{T}',H')$ be the subhierarchy of $(\mathcal{T},H)$ rooted at (i,v) with $idx' : \mathcal{T}' \to idx(\mathcal{T})$ as in definition 4.7, and

$$D(\mathcal{T}',H',f) := \{p \in N : f(p) \in idx'(\mathcal{T}') \text{ and } V(p) \cap V(\mathcal{T}'[f(p)]) \neq \emptyset\}$$

be the set of global routing pins connected by $(\mathcal{T}',H')$ according to f (see fig. 4.8 on page 103 for an illustration). Then $D(\mathcal{T}',H',f) \subseteq \tau(v)$.

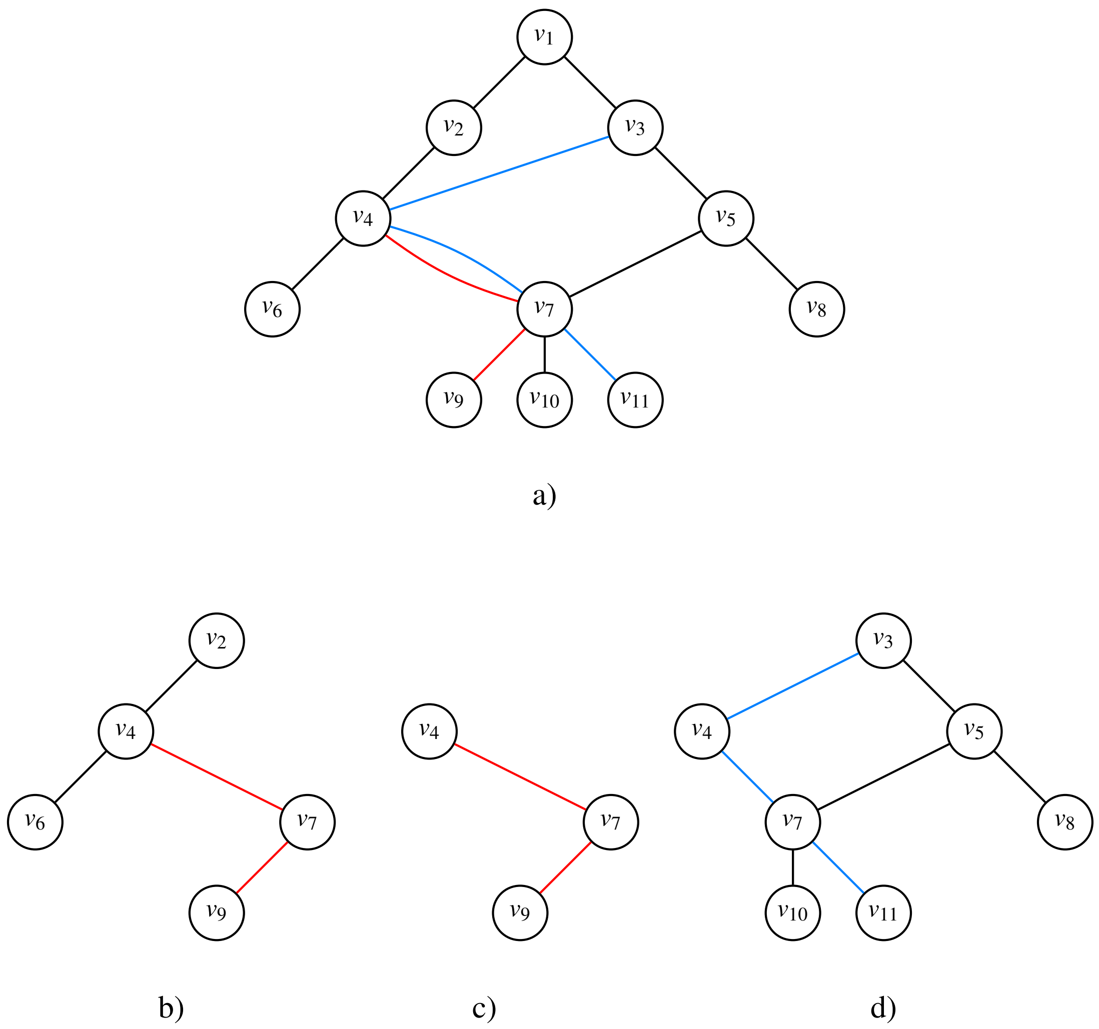
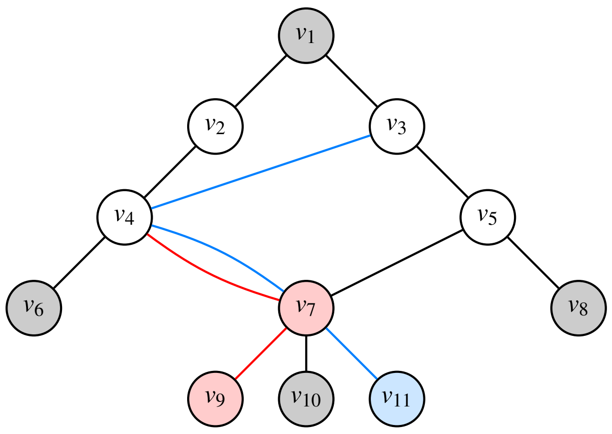
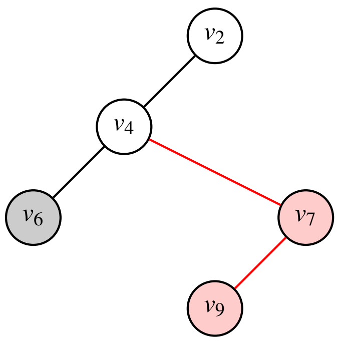


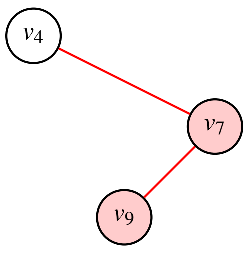
Figure 4.7: a) shows a tree hierarchy $(\mathcal{T},H)$ with three trees $\mathcal{T}[1]$ (black edges), $\mathcal{T}[2]$ (red edges) and $\mathcal{T}[3]$ (blue edges), where the highest vertex in each tree is its root, and $\mathcal{T}[1]$ is the father of $\mathcal{T}[2]$ and $\mathcal{T}[3]$ in H. Figures b), c), and d) show the subhierarchies of $(\mathcal{T},H)$ rooted at $(1,v_2)$, $(2,v_4)$ and $(1,v_3)$, respectively. Note for example that in d), $\mathcal{T}[2]$ does not appear in the subhierarchy rooted at $(1,v_3)$ because father$(\mathcal{T}[2],H) = \mathcal{T}[1]$, root$(\mathcal{T}[2]) = v_4$ and $v_4 \notin V(\text{subtree}(\mathcal{T}[1],v_3))$.



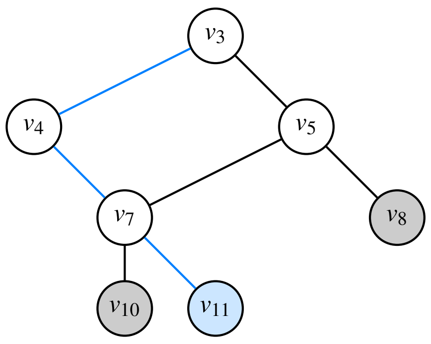
a)



b)



c)



d)

Figure 4.8: a) shows seven global routing pins p_i (drawn shaded) which are connected by the same tree hierarchy as in fig. 4.7 a), with $V(p_i) = \{v_i\}$, $i \in I := \{1,6,7,8,9,10,11\}$. For simplicity, we identify each pin with the vertex it covers. The shading color of each pin denotes a "connected-by"-mapping $f : \{p_i : i \in I\} \to \{1,2,3\}$ as in definition 4.8, i.e. p_1, p_6, p_8 and p_{10} are connected by $\mathcal{T}[1]$, p_7 and p_9 by $\mathcal{T}[2]$, and p_{11} by $\mathcal{T}[3]$. To illustrate the set D in ii) of definition 4.8, figures b), c), and d) show the subhierarchies of $(\mathcal{T}, H)$ rooted at $(1, v_2)$, $(2, v_4)$ and $(1, v_3)$, respectively, and the global routing pins connected by them (again drawn shaded) w.r.t. the "connected-by"-mapping f. Note for example that in d), p_7 is not connected because $f(p_7) = 2$, but $\mathcal{T}[2]$ does not appear in the subhierarchy rooted at $(1, v_3)$.

Feasible routings for the general case with hard pins covering more than one vertex are defined by a simple reduction as follows. Let $N^{\text{hard}} \subseteq N$ be the set of hard pins in N. If there are $p \in N^{\text{hard}}$ with $|V(p)| > 1$, we require that there is a mapping $\hat{\tau} : N^{\text{hard}} \to 2^{(2^N)}$ such that

$$\tau(v) = \bigcup_{p \in N^{\text{hard}} : v \in V(p)} \hat{\tau}(p)$$

for each $v \in V(p) : p \in N^{\text{hard}}$. Feasible routings are then defined by the following reduction (note that contracting vertices $V(p)$ for each $p \in N^{\text{hard}}$ is not equivalent because hard pins belonging to different "parts" of a net might intersect):

Definition 4.9. *Let G be an undirected graph and N a set of global routing pins in G. Let $\tau : V(G) \to 2^{(2^N)}$ be topology restrictions with $N'' \in \tau(v)$ for each $N'' \subseteq N' \in \tau(v)$ and $v \in V(G)$, and with $\tau(v) = \bigcup_{p \in N^{hard} : v \in V(p)} \hat{\tau}(p)$ for each $v \in \bigcup_{p \in N^{hard}} V(p)$, for some $\hat{\tau} : N^{hard} \to 2^{(2^N)}$.*

Let G' be the undirected graph with

$$V(G') = V(G) \,\dot{\cup}\, \{v_p : p \in N^{hard}\},$$

where $\{v_p : p \in N^{hard}\}$ are new vertices, and

$$E(G') = E(G) \cup \bigcup_{p \in N^{hard}} \{\{v_p, v\} : v \in V(p)\}.$$

Further, let $p' := p$ for each $p \in N \setminus N^{hard}$, and $p' := (\{v_p\}, \alpha, n)$ for each $p = (X, \alpha, n) \in N^{hard}$. Define $N' := \{p' : p \in N\}$, and $\tau' : V(G') \to 2^{(2^{N'})}$ with $\tau'(v) := \{\{p' : p \in I\} : I \in \tau(v)\}$ for each $v \in V(G)$, and $\tau'(v_p) := \hat{\tau}(p)$ for $v_p \in V(G') \setminus V(G)$.

Then the feasible routings for N in G are the feasible routings for N' in G' with all edges in $E(G') \setminus E(G)$ removed.

This allows us to assume w.l.o.g. that hard pins consist of only one vertex, which simplifies notation in some places (note that the graph G' from the reduction in 4.9 can be implicit in the block solver for the corresponding net).

We finally remark that tree hierarchies constructed in global routing are to be realized by *trees* in detailed routing such that the detailed routing realizations of two trees $\mathcal{T}[i]$ and $\mathcal{T}[j]$ in a tree hierarchy $(\mathcal{T}, H)$, $i, j \in \text{idx}(\mathcal{T})$, intersect if and only if $\mathcal{T}[i]$ and $\mathcal{T}[j]$ are adjacent in H, and (w. l. o. g. assuming $\mathcal{T}[j]$ is the father of $\mathcal{T}[i]$) an intersection is possible only in the area of the global routing tile corresponding to root$(\mathcal{T}[i])$. Therefore the resource consumption of a tree hierarchy $(\mathcal{T}, H)$ is given by the *multiset* $\dot{\bigcup}_{i \in \text{idx}(\mathcal{T})} E(\mathcal{T}[i])$.

4.4.1 Construction of Standard Global Routing Nets

From each detailed routing pin p_{dr} a global routing pin p_{gr} is created that covers all vertices of the global routing graph corresponding to tiles intersected by the shapes of

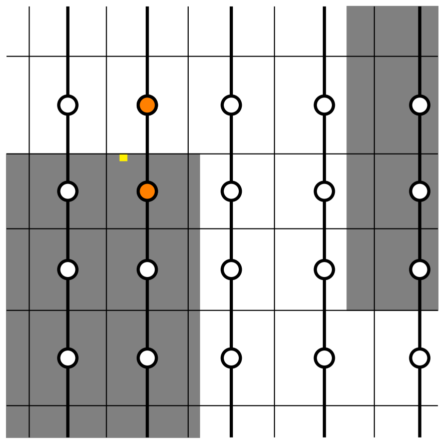


Figure 4.9: Detailed routing pin p_{dr} (yellow) at the border of a macro on a layer with vertical preferred direction. The global routing pin constructed from p_{dr} (orange vertices) is extended to cover also the vertex of the neighbouring global routing tile outside the macro.

p_{dr}. p_{gr} is defined to be a hard pin if and only if p_{dr} is hard, and a global routing net is created from each detailed routing net by replacing detailed routing pins by their corresponding global routing pins. Standard global routing nets do not have topology restrictions, but we shall see an application of topology restrictions in the next section where we define nets of "subnets" whose routes may meet only in a subset of vertices of the global routing graph.

It may be desirable to merge subsets of global routing pins of the same net into single global routing pins if a feasible routing exists for each respective subset which does not contain any edges, e.g. if two global routing pins originate from detailed routing pins lying within the same tile. The set of vertices at which such a merged group can be connected however is not necessarily the union of all vertices covered by pins of the group if it contains soft pins or there are topology constraints. Grouping of global routing pins already connected among each other is therefore optional on a per-net basis in BonnRoute®, and should not be done if a net containing soft pins is to be routed optimally by the algorithm presented in section 4.6.

Because in detailed routing all corresponding physical pin shapes have to be connected with each other and this is not reflected in global routing resource usage, edge capacities are reduced slightly depending on an estimated amount of detailed routing connections required among physical pin shapes corresponding to the same global routing pin.

Global routing pins constructed from detailed routing pins at the border of macros are often accessible only using zero capacity edges in the global routing graph even though the pin can be accessed legally in detailed routing. Figure 4.9 illustrates this problem: Here the edge connecting the two orange vertices has zero capacity because the global routing tile containing the yellow detailed routing pin is entirely covered by blockage. Therefore the global routing pin constructed from the yellow pin is extended such that it covers both orange vertices in the global routing graph. This technique solves most false unroutability issues, i.e. cases in which

$$\max_{T \in \mathcal{T}_N} \min_{e \in E(T)} (u(e) - w(N,e)) < 0$$

for a global routing net $N \in \mathcal{N}$, but the corresponding detailed routing net can be legally routed. The remaining problems of this type involve more complex blockage structures, very wide wires used by a small number of nets $N \in \mathcal{N}$ (i.e. large $w(N,e)$ for all or some edges $e \in E(G)$), and the fact that a wire in our global routing formulation cannot share capacities of two neighbouring global routing edges running in parallel in the same layer. Most nets are not affected by these problems in practice because the standard wire type has a small wire width and tile borders can usually be chosen such that there are only very few edges with extremely small non-zero edge capacities.

As by far most remaining false unroutability issues involve zero capacity edges in practice and are due to local modeling inaccuracies, restricting feasible routings to those with a minimum number of zero capacity edges yields corridors within which a detailed routing solution can be found, if they are extended by a small amount. Minimizing the number of zero capacity edges in a routing for a net is (approximately) achieved by high costs on such edges; see also the discussion in sections 4.7.1 and 4.7.3.

Wire widths $w(N,e_1)$ and $w(N,e_2)$ for a global routing net $N \in \mathcal{N}$ and edges $e_1, e_2 \in E(G)$ in practice are equal if e_1 and e_2 are wiring edges within the same wiring layer, or via edges between the same layers. This is due to the fact that with very few exceptions, the same detailed routing wire type is used for all wires of the corresponding detailed routing net. Therefore only few numbers have to be stored to encode wire widths for a global routing net. For two nets $N_1, N_2 \in \mathcal{N}$ whose corresponding detailed routing nets use the same detailed routing wire type, these numbers are grouped into a *global routing wire type*. A global routing net then stores only a pointer to a global routing wire type instead of a list of wire widths. The maximum width ratio between wiring edges in different wiring layers used by the same global routing net of course depends on the technology. Typical values are in the range of 4 to 10.

By far most pins are hard in current technologies because their internal resistance between all pairs of access points is comparable to the resistance of a wire connecting these points. This is the case e.g. for pins with only one shape that consists of metal (usually copper), as opposed to pin shapes consisting of polysilicon, which has a far lower conductance than copper. While the distinction between hard and soft pins is not essential for standard cell pins which are much smaller than the tile size used in the construction of the global routing graph, supporting soft pins can be useful for solving a special case of the port assignment problem which is discussed in the next section.

4.5 Port Assignment in Hierarchical Design

Hierarchical design means grouping disjoint sets of circuits on a chip together and working on each group more or less independently. Each group, called *random-logic macro* or *RLM* for short, is assigned a subarea of the chip area within which all its elements have to be placed and connections among them have to be routed. Subareas assigned to different groups are disjoint, and all circuits not in one of the RLMs define a special group called the *top level* which can use those parts of the chip not assigned to one of the RLMs. This grouping can be done recursively, i.e. circuits within an RLM might again be grouped to RLMs. A design with one hierarchy level, i.e. without RLMs, is called *flat*.

Of course nets can contain pins of circuits that belong to different RLMs (or to the top level), therefore RLMs and the top-level are not fully independent from each other. In order to allow to work on an RLM or on the top level independently from other parts, an *interface* is defined, both in terms of geometry and timing. For a net that has to connect pins both inside and outside of an RLM, a *port* consisting of a set of interconnection points is defined where the wiring of the net may cross the boundary of the RLM area. To make the netlist data of RLMs and the top level independent from each other, nets are splitted then: For each net having a pin in some RLM or the top level, a subnet is created containing the pins that belong to circuits assigned to this RLM or the top level, respectively. Additionally, port locations on the boundary of the RLM or top level, respectively, are added to the subnet. The reverse operation, i.e. merging all subnets of the same net and deleting the pins which are defined by port locations, is called *flattening* of a hierarchical netlist.

For simplicity we assume to have two levels of hierarchy, i.e. RLMs contain no other RLMs, and that RLM areas keep a certain minimum distance from each other such that a port always defines interconnection points between an RLM and the top level. If a port consists of more than one point, the routing on the top level can choose a point to connect to, and inside the RLM all interconnection points must be connected with each other such that they are guaranteed to be electrically equivalent.

Similarly to the geometric interface defined by the ports, the interface must define timing requirements, i.e. signal arrival times or required arrival times, respectively, of signals at the ports. Of course hierarchical design in general limits the possibilities for optimization compared to the results that can be obtained on a flat design. Hierarchy however is required sometimes because of tool limitations (e.g. running times becoming inacceptably high on large problem instances) or organizational needs. In these cases defining a good interface, i.e. port locations and timinig assertions, between RLMs and the top level is essential for hierarchical solutions to be not too far away from a global optimum that could (approximately) be reached with a flat design approach.

In this section we focus on how global routing can help in finding good port locations, assuming a flat placement is given. After the interface definition, RLM and top-level design can be refined independently from each other as long as designers adhere to the defined interface.

Often several copies of an RLM, called an *RLM group*, are instantiated on a chip, where each copy is required to have an identical physical design up to mirroring and rotation. Given a placement which is identical within each member of an RLM group, we can use the algorithm presented in section 4.6 as block solver in a global routing of the flat netlist in order to define ports such that congestion and objective costs are globally minimized, and the global routing inside each member of an RLM group is identical. This optimizes also over ports that do not consist of a single point, and yields a least-cost connection among the port locations inside the RLM. This is important because sometimes not all copies of an RLM can be entered at the same point because the neighbourhood (top level congestion and blockages) is different for each copy of the RLM.

Consider an RLM group with k members, $k \in \mathbb{N}$. We assume for simplicity that RLMs have axis-parallel rectangular areas and are not mirrored or rotated, and require that the sets of nodes they cover are pairwise disjoint and identical up to translation, i.e. there are $n_x, n_y, n_z \in \mathbb{N}$ such that

$$M_i := \left\{(x,y,z) : a_x^i \le x \le a_x^i + n_x, a_y^i \le y \le a_y^i + n_y, 0 \le z \le n_z\right\} \subseteq V(G)$$

is the set of vertices of the global routing graph G covered by the area assigned to member i of the group for some anchor $(a_x^i, a_y^i) \in \mathbb{N}^2$, and $1 \le i \le k$. Further, we require that edge capacities are identical in each copy of the RLM, i.e. $u(e) = u(e')$ for each $e = \{(x_1,y_1,z_1),(x_2,y_2,z_2)\} \in E(G[M_j])$ and

$$e' = \{(x_1 + a_x^i - a_x^j, y_1 + a_y^i - a_y^j, z_1)\,,\,(x_2 + a_x^i - a_x^j, y_2 + a_y^i - a_y^j, z_2)\},$$

$1 \le i < j \le k$.

We shall see later that mirroring and rotation is easy to take into acccount. Requiring each copy to cover an identical set of nodes up to translation, and identical edge capacities is easiest achieved in practice by using a regular tiling for defining the global routing graph, and then restrict the placement to snap RLM boundaries to this raster (of course also intersections of top-level blockages like the power grid with RLM areas have to match each other).

Now consider k nets $N_1, \ldots, N_k$, and assume that each net N_i has $t \in \mathbb{N}$ pins inside RLM i, and all others outside, i.e. $N_i' := \{p_i^1, \ldots, p_i^t\} \subseteq N_i$, $|N_i'| = t$, $V(p) \subseteq M_i$ for each $p \in N_i'$, and $V(p) \subseteq V(G) \setminus M_i$ for each $p \in N_i \setminus N_i'$. Further we require that N_i' and $N_{i'}'$ are identical up to translation according to the RLM anchors and pin identifiers, i.e.

$$\mathrm{proj}(p_i^j, i) = \mathrm{proj}(p_{i'}^j, i')$$

for $1 \le i < i' \le k$ and $1 \le j \le t$, where

$$\mathrm{proj}((V,\alpha,n),i) := (\{(l_x - a_x^i, l_y - a_y^i, l_z) : (l_x,l_y,l_z) \in V\}, \alpha)$$

for $(V,\alpha,n) \in 2^V \times \{\text{hard}, \text{soft}\} \times \mathbb{N}$ and $1 \le i \le k$. Analogously to edge capacities, of course we require wire widths of all k nets to be identical on edges in different RLM copies that correspond to each other. This is the case in practice if all k nets have the same wire type.

For simplicity, let us assume that the pins in $N_i \setminus N_i'$ are in the top level and not inside other RLMs ($1 \le i \le k$). We create a graph $G' = (V', E')$ from G by contracting the vertex sets

$$\left\{(a_x^i + d_x, a_y^i + d_y, d_z) : 1 \le i \le k\right\}$$

for each $0 \le d_x \le n_x$, $0 \le d_y \le n_y$ and $0 \le d_z \le n_z$ and denote the set of contracted vertices $V_c' \subseteq V'$. Parallel edges $e_1, \dots, e_k$ between vertices in V_c' are replaced by a single edge e with objective cost

$$\gamma_{N,e}^{\text{obj}}(s) := \sum_{i=1}^{k} \gamma_{N,e_i}^{\text{obj}}(s) \quad \text{for all} \quad N \in \mathcal{N} \quad \text{and} \quad s \ge 0,$$

as using edges inside the RLM means adding a wire segment to all k copies of the RLM, and capacity $u(e) := u(e_1)$.

We merge the nets $N_1, \dots, N_k$ into a single net N with pins

$$N_1'' \quad \cup \quad \bigcup_{1 \le i \le k} \left(N_i \setminus N_i'\right),$$

where N_1'' are the pins in N_1' mapped to the corresponding contracted vertices.

Informally, we look for routes that connect the pins $N_i \setminus N_i'$ for $1 \le i \le k$ independently from each other, but merge inside the RLM, i.e. within the vertex set V_c'. In other words, the original nets N_i are to be treated as different nets outside V_c', and as parts of the same net within V_c'. More precisely, given *topology restrictions* $\tau : V(G') \to 2^{(2^N)}$ with

$$\tau(v) := \begin{cases} 2^N & : \quad v \in V_c' \\ \bigcup_{1 \le i \le k} 2^{\left(N_i \setminus N_i'\right)} & : \quad \text{otherwise,} \end{cases}$$

we seek a feasible routing for (N, τ) in G'. The next section shows how this can be done.

Remark. Sometimes copies of RLMs are mirrored. This can be dealt with by contracting different sets of vertices. For example, if the i-th copy of the RLM group above ($1 \le i \le k$) is mirrored with a vertical symmetry axis, simply exchange vertices $(a_x^i + j, y, z)$ and $(a_x^i + n_x - j, y, z)$ in the vertex sets to be contracted, for each $0 \le j \le \lfloor n_x/2 \rfloor$, $a_y^i \le y \le a_y^i + n_y$ and $0 \le z \le n_z$.

Of course, if there are nets with pins in different RLMs, a larger set of nets has to be grouped into a single net to be routed at once. We also remark that the topology restrictions defined above allow the routing for N inside the RLM to have more than one connected component. This can happen if for one of the original nets N_i ($1 \le i \le k$) paying k times the objective cost plus (once) the congestion cost inside the RLM is cheaper than paying for global congestion somewhere outside the RLM, so the wiring for N_i could "run through" a part of the RLM to evade extreme top level congestion and then enter the RLM somewhere else to join the routes of the other nets.

We finally discuss some special cases of the port assignment problem. First we consider port assignment in early design stages where the RLM design (placement or even logic design) is very incomplete, but a rough *floorplanning* of the top level has to be done. We only assume that each copy of the RLM has been assigned an area identical up to translation as above, and there are nets that contain pins "inside" the RLM, but with unspecified locations. In this case we replace these pins for a net by a single large pin covering the RLM area, declaring it as *hard* if the RLM designer should connect all port locations among each other inside the RLM (so that the top level designer can pick one of them to connect to in each copy), or as *soft* otherwise. In the latter case, edges in the subgraph induced by the RLM must be deleted, which enforces a single port location in an optimum routing for the net combined from the nets that connect the same logical pin in each RLM copy. In the first (hard pin) case, the new edges added in the reduction given in definition 4.9 should get a high weight corresponding to half of the expected cost for connecting port locations identically in each RLM copy.

The simplest case is that there is only one copy of each RLM and the physical design of the RLM is available. In this case a flat global routing without topology restrictions suffices to determine port locations.

We finally remark that at some point in physical design port locations must also be *legalized*, i.e. placed disjointly on the RLM boundary within the global routing corridor. This has to be done as soon as the RLM or top level designer wants to compute a detailed routing. If there is only one copy of each RLM, legalization can be done by a normal detailed routing run. Otherwise some procedure that ensures that legalized port locations are identical in each RLM copy is necessary, e.g. an enhanced pin access procedure that for each RLM pin (coarsely defined as the intersection of the RLM area with the global routing area) finds a set of pin access stubs such that at least one element of this set can be used to access the pin from the top level in each RLM copy, and the sets for different pins are conflict-free.

4.6 An Optimum Algorithm for Routing a Single Net

Let G be an undirected graph, N a set of global routing pins in G, and $\tau : V(G) \to 2^{(2^N)}$ topology restrictions with $I \in \tau(v)$ for each $I \subseteq I' \in \tau(v)$ and $v \in V(G)$. A minimum-cost feasible routing for (N,τ) w.r.t. edge costs $c : E \to \mathbb{R}_+$ can be found by a dynamic programming approach shown on page 112. In the returned routing $(\mathcal{T},H)$, for convenience H is a tree on the indices of the trees in $\mathcal{T}$, instead of the trees themselves. We can safely assume that $|V(p)| = 1$ for each hard pin $p \in N$ because of definition 4.9.

Vygen [2001] presented a dynamic programming algorithm for the minimum weight Steiner tree problem in graphs and suggested an extension to the group Steiner tree problem by changing the initialization step (treating all groups like soft pins in our terminology). We explicitly allow soft pins, but as we can assume hard pins to cover only one vertex, the main difference of our algorithm is that it can handle topology restrictions, as needed in the last section for hierarchical routing. The correctness proof of our algorithm is similar to the one in Vygen [2001]:

Theorem 4.10. *The* GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM *works correctly.*

Proof. For any $I \subseteq N$ and $v \in V(G)$, let $\mathrm{opt}_I(c,v)$ be the minimum total cost $\sum_{T\in\mathcal{T}}\sum_{e\in E(T)} c(e)$ of a feasible routing $(\mathcal{T},H)$ for (I,τ) with $\mathrm{root}(\mathrm{root}(H)) = v$, if such a routing exists, or ∞ otherwise.

We prove that the following invariants hold for each non-empty subset $I \subseteq N$ each time when line 4 is executed:

a) For each $v \in V(G)$ and $I' \supseteq I$: $l_I(v) \le l_{I'}(v)$ and $\mathcal{R}_{I'} \subseteq \mathcal{R}_I$.

b) For each $v \in V(G)$ with $0 < l_I(v) < \infty$: Let $\mathcal{V}_I(v) = \{(w_1,I_1),\dots,(w_j,I_j)\}$, then $I\setminus\{p\in N : v\in V(p)\} = I_1 \dot\cup \dots \dot\cup I_j$, and for each $1 \le i \le j$ we have $w_i \in \mathcal{R}_{I_i}$, and $l_I(v) = \sum_{i=1}^{j}(l_{I_i}(w_i) + c(\{v,w_i\}))$.

c) For each $v \in \mathcal{R}_I$: $l_I(v) = \mathrm{opt}_I(c,v)$.

d) For each $v \in V(G)\setminus\mathcal{R}_I$: $l_I(v) \ge \mathrm{opt}_I(c,v)$. If $I \in \tau(v)$ and $I \subseteq \{p\in N : v\in V(p)\}$, we have $l_I(v) = 0$, and $l_I(v) = \infty$ if $I \notin \tau(v)$. Otherwise $l_I(v)$ is the minimum of $\sum_{i=1}^{j}(l_{I_i}(w_i) + c(\{v,w_i\}))$ over all partitions $I\setminus\{p\in N : v\in V(p)\} = I_1 \dot\cup \dots \dot\cup I_j$ and all neighbours $w_1,\dots,w_j$ of v with $w_i \in \mathcal{R}_{I_i}$ for $i = 1,\dots,j$.

All four statements are satisfied after initialization (lines 1 and 2) because $I \in \tau(v)$ for each $I \subseteq I' \in \tau(v)$, $v \in V(G)$. We show that they are preserved by lines 5 to 11. In the following, let I and v be the set and the vertex chosen in line 4 in some particular iteration.

Because a) was true in the last iteration, by the properties of τ and the choice of I in line 4 it continues to hold after lines 5 to 11 have been executed in this iteration. Clearly, b) is still true after line 5. By a), (v,I) is not added to $\mathcal{V}_{I'}(w)$ for $I' \supseteq I$ in lines 8 or 11 if $I \cap \{p\in N : w\in V(p)\} \ne \emptyset$, so b) remains true at the end of this iteration. By a) and the induction hypothesis, also d) is preserved by lines 5 to 11.

It remains to show that c) holds for the new element v added to $\mathcal{R}_I$ in line 5. Because $l_I(v) \ge \mathrm{opt}_I(c,v)$ by statement d) of the induction hypothesis, we need to prove only $l_I(v) \le \mathrm{opt}_I(c,v)$.

Let $(\mathcal{T},H)$ be a feasible routing for I with $\mathrm{root}(\mathrm{root}(H)) = v$ and indexing $\mathrm{idx} : \mathcal{T} \to \mathbb{N}$, and $f : I \to \mathrm{idx}(\mathcal{T})$ a "connected-by"-mapping as in definition 4.8. We assume w.l.o.g. that $\mathrm{root}(T) = \mathrm{root}(\mathrm{father}(T,H))$ only if $\mathrm{father}(T,H) = \mathrm{root}(H)$, for $T \in \mathcal{T}\setminus\mathrm{root}(H)$. For $i \in \mathrm{idx}(\mathcal{T})$ and $w \in V(\mathcal{T}[i])$, let $(\mathcal{T}_{(i,w)},H_{(i,w)})$ be the subhierarchy of $(\mathcal{T},H)$ rooted at (i,w), and $J_{(i,w)} := D(\mathcal{T}_{(i,w)},H_{(i,w)},f)$ the pins connected by this subhierarchy according to f. We call $(i,w) \in \mathrm{idx}(\mathcal{T}) \times V(G)$ with $w \in V(\mathcal{T}[i])$ *proper* if $w \in \mathcal{R}_{J_{(i,w)}}$ or $J_{(i,w)} = \emptyset$. The "leaves" (i,w) of the routing $(\mathcal{T},H)$, i.e. w leaf in $\mathcal{T}[i]$ for some $i \in \mathrm{idx}(\mathcal{T})$, are all proper, and $(\mathrm{idx}(\mathrm{root}(H)),v)$ was not proper before v was added to $\mathcal{R}_I$ in line 5. This implies that there is an $i \in \mathrm{idx}(\mathcal{T})$ and $w \in V(\mathcal{T}[i])$

```
GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM
Input  : An undirected graph $G$, weights $c : E(G) \to \mathbb{R}_+$, a set $N$ of global
         routing pins in $G$, and topology restrictions $\tau : V(G) \to 2^{(2^N)}$ with
         $I \in \tau(v)$ for each $I \subseteq I' \in \tau(v)$ and $v \in V(G)$.
Output: Either "infeasible", or a feasible routing $(\mathcal{T} = \{T_1, \dots, T_q\}, H)$, $q \in \mathbb{N}$,
         with minimum total cost $\sum_{i=1}^{q} \sum_{e \in E(T_i)} c(e)$.
1  Set $\mathcal{R}_I := \emptyset$, $l_I(v) := \infty$ and $\mathcal{V}_I(v) := \emptyset$ for all $I \subseteq N$ and $v \in V(G)$.
2  For all $I \subseteq N$, set $l_I(v) := 0$ for $v \in \cap_{p \in I} V(p)$ with $I \in \tau(v)$.
3  while $\mathcal{R}_N = \emptyset$ and there is an $I \subseteq N$ and $v \in V(G) \setminus \mathcal{R}_I$ with $l_I(v) \neq \infty$ do
4      Choose a nonempty $I \subseteq N$ and $v \in V(G) \setminus \mathcal{R}_I$ such that $I$ is minimal with
       $l_I(v) \leq l_{I'}(w)$ for any $I' \subseteq I$ and $w \in V(G) \setminus \mathcal{R}_{I'}$.
5      Set $\mathcal{R}_I := \mathcal{R}_I \cup \{v\}$.
6      foreach $e = \{v, w\} \in E(G)$ with $w \notin \mathcal{R}_I$ do
7          if $l_I(w) > l_I(v) + c(e)$ and $I \in \tau(w)$ then
8              Set $l_I(w) := l_I(v) + c(e)$ and $\mathcal{V}_I(w) := \{(v, I)\}$.
9          foreach $I' \supset I$ with $I' \in \tau(w)$ do
10             if $l_{I'}(w) > l_I(w) + l_{I' \setminus I}(w)$ then
11                 Set $l_{I'}(w) := l_I(w) + l_{I' \setminus I}(w)$ and $\mathcal{V}_{I'}(w) := \mathcal{V}_I(w) \cup \mathcal{V}_{I' \setminus I}(w)$.
12 if $\mathcal{R}_N \neq \emptyset$ then
13     Choose $r \in \mathcal{R}_N$ and set $q := 1$, $T_1 := (\{r\}, \emptyset)$, root$(T_1) := r$, and root$(H) := 1$.
14     Call Backtrace$(r, N)$.
15 else
16     return "infeasible".

17 Procedure Backtrace$(v \in V(G), I \subseteq N)$:
18 Set $q^* := q$.
19 foreach $(w, I') \in \mathcal{V}_I(v)$ do
20     if $E(subtree(T_{q^*}, v)) \neq \emptyset$ then
21         Set $q := q + 1$, $T_q := (\{v\}, \emptyset)$ and root$(T_q) := v$.
22         Set $H := H \cup \{\{q, q^*\}\}$.
23     Set $T_q := T_q \cup \{\{v, w\}\}$.
24     Call Backtrace$(w, I')$.
```

such that (i,w) is not proper before executing line 5 in this iteration, but all "neighbours" $(i_1,w_1),\dots,(i_k,w_k)$ in $(\mathcal{T}_{(i,w)},H_{(i,w)})$ are, where (i_j,w_j) is a neighbour of (i,w) in $(\mathcal{T}_{(i,w)},H_{(i,w)})$ iff $w_j \in V(\mathcal{T}_{(i,w)}[i_j]) \setminus \mathrm{root}(\mathcal{T}_{(i,w)}[i_j])$, $\mathrm{father}(w_j,\mathcal{T}_{(i,w)}[i_j]) = w$ and either $i_j = i$ or $\mathrm{father}(\mathcal{T}_{(i,w)}[i_j],H_{(i,w)}) = \mathcal{T}_{(i,w)}[i]$ and $\mathrm{root}(\mathcal{T}_{(i,w)}[i_j]) = w$.

Then c) holds for $(w_1,J_{(i_1,w_1)})$, …, $(w_k,J_{(i_k,w_k)})$ because (i_j,w_j) are proper for each $1 \le j \le k$. Because $J_{(i,w)} \subseteq \tau(w)$ and d) is true for $J_{(i,w)}$ and w, we have

$$\begin{aligned} l_{J_{(i,w)}}(w) &\le \textstyle\sum_{j=1}^k (l_{J_{(i_j,w_j)}}(w_j) + c(\{w_j,w\})) \\ &= \textstyle\sum_{j=1}^k (\mathrm{opt}_{J_{(i_j,w_j)}}(c,w_j) + c(\{w_j,w\})) \\ &\le \textstyle\sum_{j=1}^k (\sum_{T\in\mathcal{T}_{(i_j,w_j)}} c(T) + c(\{w_j,w\})) \\ &= \textstyle\sum_{T\in\mathcal{T}_{(i,w)}} c(T) \end{aligned} \tag{4.7}$$

By the choice of I and v in line 4 we must therefore have $l_I(v) \le \sum_{T\in\mathcal{T}} c(T)$. □

The runtime can be bounded by $O(3^{|N|}|V(G)| + 2^{|N|}|V(G)|\log|V(G)|)$. As our algorithm can only make less labels than that of Vygen [2001] due to topology restrictions, we do not reproduce the proof from that work. As in Vygen [2001], a lower bounding technique can be added to significantly reduce runtimes in practice, making the algorithm practically usable as the majority of global routing nets has a very small number of pins. If in the application of the last chapter, the number of pins outside an RLM in each of the "subnets" of N is small, and $3^{|N|}|V(M)|$ is not too large, where M is the set of RLM vertices, the GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM might therefore be of value. The number of nets for which port assignment has to be done is usually very small compared to the number of all nets, and a high-quality solution to the port assignment problem justifies an increased computational effort. However, similarly to topology-unrestricted nets, it might turn out that a series of path searches using Dijkstra's algorithm achieves reasonably good results in practice. This approach could then also be used on larger instances.

Volatility Tolerance

Unlike Dijkstra's algorithm for connecting two sets of vertices by a path (see section 3.3.2), the GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM queries the cost of an edge more than once in general. For a volatility-tolerant implementation of the GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM one therefore has to save the cost of an edge after the first query, and use this value later instead of performing a new query on the same edge.

4.7 Implementation Aspects

This section discusses some key aspects and tradeoffs that have been made in the implementation of the PARALLEL RESOURCE SHARING ALGORITHM, the block solver, randomized rounding, and ripup and reroute in the global routing module of BonnRoute®.

As in the abstract formulation, block solvers are not part of the resource sharing algorithm and can be implemented individually for different classes of nets, or even single nets. In fact, the implementation of the resource sharing algorithm is independent of global routing and is kept separate from global routing specific data structures.

For example, an implementation of the GENERALIZED DIJKSTRA-GROUP-STEINER ALGORITHM can be used as block solver for nets with topology restrictions as needed for port assignment in hierarchical design, or to find optimum Steiner forests for nets with large soft pins. For most nets, a heuristic decomposition of the routing task for multi-pin nets into a series of Dijkstra path searches achieves sufficient quality. This method is used in the *standard block solver* presented below in section 4.7.3, which currently is the only block solver implemented in BonnRoute®.

The modular implementation also makes it easier to add new types of resources. An important example is buffering space, which currently is not represented in BonnRoute®. Adding this resource type neither affects the implementation of the resource sharing algorithm, nor of the already existing standard block solver. Of course a buffering-aware block solver must be provided for those nets that require buffering, and the iterative refinement routine employed after randomized rounding to find a feasible integral solution must consider the new resources.

4.7.1 Resource Sharing Algorithm

Weighting of resource prices

The cost for using one unit of a resource $r \in \mathcal{R}$ in the RESOURCE SHARING ALGORITHM is $y_r(\zeta) := \psi_r e^{\varepsilon_2 \zeta}$, where ζ is the current utilization of resource $r \in \mathcal{R}$, ε_2 is a parameter to the RESOURCE SHARING ALGORITHM, and $\psi_r \geq 1$ is a weighting factor (see definition 3.1).

Let now $G = (V, E)$ be the global routing graph, $N \in \mathcal{N}$ a global routing net in G, and $T \in \mathcal{T}_N$ in an instance of the GLOBAL ROUTING PROBLEM. Let us assume for simplicity that there are no critical paths with delay bounds, and objective costs do not depend on spacing, i.e. $\gamma^{\text{obj}}_{N,e}(s_1) = \gamma^{\text{obj}}_{N,e}(s_2)$ for $s_1, s_2 \geq 0$. Then using edge $e \in E(T)$ consumes $w(N,e)/u(e)$ units of the resource r_e which models edge capacity of edge e, and $\gamma^{\text{obj}}_{N,e}(0)/\Gamma^{\text{obj}}$ units of the resource r_{obj} which models the optimization objective, so the contribution of using edge e in T to the total cost of the feasible routing $T \in \mathcal{T}_N$ for N is

$$\frac{\gamma^{\text{obj}}_{N,e}(0)}{\Gamma^{\text{obj}}} \psi_{r_{\text{obj}}} e^{\varepsilon_2 \zeta_{r_{\text{obj}}}} + \frac{w(N,e)}{u(e)} \psi_{r_e} e^{\varepsilon_2 \zeta_{r_e}} \tag{4.8}$$

(see section 4.2), where $\zeta_{r_{\text{obj}}}$ and ζ_{r_e} are the current resource utilizations of r_{obj} and r_e, respectively. Typically, $w(N,e)/u(e) \geq 10^{-2}$, but $\gamma^{\text{obj}}_{N,e}(0)/\Gamma^{\text{obj}} \leq 10^{-8}$ on large designs because the average size of global routing tiles is chosen such that at most 50 to 100 nets can share the same edge in the global routing graph, but all nets of course share the resource r_{obj}. The consequence of this is that for reasonable values of the parameter ε_2

and with unit weights ψ, utilization of resource r_{obj} in the first phases of the RESOURCE SHARING ALGORITHM is almost for free compared to edge congestion costs, i.e. costs for using edge capacity resources. Hence, the algorithm *spreads* the wiring very far across the chip area in the first phases, at the cost of increased wiring length, critical area and power consumption. In later phases, when $e^{\varepsilon_2 \zeta_{r_{\text{obj}}}}$ has increased sufficiently, many nets are rerouted with shorter wiring length. We observed considerably faster convergence towards an optimum solution in practice by setting $\psi_{r_{\text{obj}}}$ such that

$$\psi_{r_{\text{obj}}} \sum_{N \in \mathcal{N}} \sum_{e \in E(G)} \frac{\gamma_{N,e}^{\text{obj}}(0)}{\Gamma^{\text{obj}}} = \sum_{N \in \mathcal{N}} \sum_{e \in E(G)} \frac{w(N,e)}{\max\{u(e), w(N,e)\}},$$

and $\psi_{r_e} := 1$ for $e \in E(G)$. This can increase the worst-case runtime only by a constant factor under reasonable assumptions: First, we can restrict Γ^{obj} to be at most

$$|E(G)| U \gamma_{\max},$$

where $\gamma_{\max} := \max\{\gamma_{N,e}^{\text{obj}}(0) : e \in E(G), N \in \mathcal{N}\}$ and $U := \max\{\frac{u(e)}{w(N,e)} : e \in E(G), N \in \mathcal{N}\}$, as we are interested only in feasible solutions or a certificate of infeasibility. Then with $\gamma_{\min} := \min\{\gamma_{N,e}^{\text{obj}}(0) : e \in E(G), N \in \mathcal{N}\}$ we have

$$\begin{aligned} \psi_{r_{\text{obj}}} &\leq \frac{\sum_{N \in \mathcal{N}} |E(G)|}{\sum_{N \in \mathcal{N}} \frac{\gamma_{\min}}{U \gamma_{\max}}} \\ &\leq |\mathcal{R}| U \frac{\gamma_{\max}}{\gamma_{\min}}. \end{aligned}$$

Because $U \frac{\gamma_{\max}}{\gamma_{\min}}$ is much smaller than $|\mathcal{R}|$ in practice, we have

$$\Psi := \sum_{r \in \mathcal{R}} \psi_r = O(|\mathcal{R}|^2)$$

and thus an increase of worst-case runtime by only a small constant factor compared to unit weights (see Theorem 3.6). The practical benefit however outweighs this by far.

Numerical Precision

If numbers are not represented symbolically, but explicitely by listing their (binary) digits, precision of calculations is of course limited. While there are software packages for calculating with arbitrary precision (only limited by memory space), using hardware operations on floating point numbers as defined in the IEEE 754 standard [1985] is much faster. Our C++ implementation of the RESOURCE SHARING ALGORITHM takes a template argument specifying the data type for implementing numbers, so precision limits are not fixed by the implementation. By default the C++ `double` data type is used, corresponding to 64-bit double-precision IEEE-754 floating point numbers for which hardware instructions performing elementary calculations are provided on all important processors. Switching from double-precision to extended-precision numbers (with 80 bits instead of 64), also directly supported in hardware, already causes a significant runtime penalty, but offers almost no benefit in practice.

With double-precision IEEE-754 floating point numbers $a,b \in \mathbb{Q}_+ \setminus \{0\}$, the condition that

$$a+b > \max\{a,b\} \tag{4.9}$$

holds only if $\max\{a,b\}/\min\{a,b\} \leq 2^{52}$, as the mantissa of double-precision numbers encompasses 52 bits (11 bits are used for the exponent, and one bit for the sign). Ensuring this condition is important because otherwise resources can be used at an effective cost of zero, causing unnecessary routing detours.

Although resources are introduced only for edges with positive edge capacity, the block solver (see section 4.7.3 below) operates directly on the global routing graph. Because we want to minimize the number of zero capacity edges, i.e. edges in $E(G) \setminus E^+$, in a feasible routing generated by a call to the block solver, we have to impose costs on $e \in E(G) \setminus E^+$ which are greater than the sum of costs for using edges with positive capacity. Further, the block solver has to increment total costs a of partial routings for a net by costs b for adding an edge, so the ratio between maximum and minimum edge costs on edges with positive capacity may be limited to

$$\eta := 2^{52}/|\mathcal{R}|^2 \tag{4.10}$$

in the worst case if (4.9) is to be guaranteed under all circumstances. For simplicity, we neglect here that using an edge also consumes other resources than edge capacity, feasible routings can use an edge more than once in the general GLOBAL ROUTING PROBLEM, and that $1/k \leq w(N,e_1)/w(N,e_2) \leq k$ for a small constant k, $N \in \mathcal{N}$ and $e_1,e_2 \in E(G)$. Based on the value of η, we define an interval $[\lambda^p_{\min}, \lambda^p_{\max}]$ of congestion values for which (4.9) can be guaranteed in phase p of the fractional RESOURCE SHARING ALGORITHM, $\lambda^p_{\min}, \lambda^p_{\max} \in \mathbb{R}_+$, with $\lambda^p_{\min} = \lambda^p_{\max} - \frac{\ln \eta}{\varepsilon_2 p}$ (assuming uniform resource weights ψ for simplicity), and replace y_r from definition 3.1 by

$$y_r(\zeta) := \psi_r e^{\varepsilon_2 \max\{\min\{\zeta, p\lambda^p_{\max}\}, p\lambda^p_{\min}\}} \tag{4.11}$$

in phase p of the RESOURCE SHARING ALGORITHM.

The value of η drops below 10 on the largest of todays global routing instances, providing only a very small congestion interval. In practice, it suffices to ensure that costs for using an edge with zero capacity are 100 times higher than the highest cost for using an edge with positive capacity; this minimizes the number of zero capacity edges in a shortest path by far in most cases. Usually, not more than ten zero capacity edges are needed per path (in most cases only for accessing blocked pins). Therefore it is relatively safe in practice to replace the factor $|\mathcal{R}|^2$ above by 2^{10}, providing a ratio of 2^{42} between maximum and minimum edge congestion costs. With wire type widths

$w(N,e)$ typically differing only by a small factor for fixed $N \in \mathcal{N}$, we can choose $\lambda^p_{\min}$ and $\lambda^p_{\max}$ such that

$$\lambda^p_{\max} = \lambda^p_{\min} + \frac{\ln(2^{42})}{\varepsilon_2 p} \approx \lambda^p_{\min} + \frac{29}{\varepsilon_2 p}$$

in phase p of the algorithm. For example, with $\varepsilon_2 = 1$, the interval $[\lambda^p_{\min}, \lambda^p_{\max}]$ can be set to $[0, 29]$ in the first phase, $[0.71, 1.29]$ in the 50-th phase and $[0.855, 1.145]$ in the 100-th phase of the RESOURCE SHARING ALGORITHM. This suffices in practice because routable chips typically have $\lambda^* \in [0.9, 1]$, and on unroutable chips the value of λ^* is not of high interest if $\lambda^* > 1.1$. Also, a number t of phases with $t\varepsilon_2 = 100$ usually is sufficient to obtain good results.

As expected, if the maximum congestion exceeds $\lambda^p_{\max}$ at some time during phase p of the algorithm ($1 \le p \le t$), it often increases significantly from that point. Bounding ζ from below by $p\lambda^p_{\min}$ in (4.11) with $\lambda^p_{\min} < \lambda^*$ does not cause significantly different behaviour of the algorithm.

For these reasons, 64-bit floating point numbers can be chosen in practice as data type to represent costs in the implementation of the RESOURCE SHARING ALGORITHM applied to the GLOBAL ROUTING PROBLEM.

Ordering of Nets

Although the analysis of the RESOURCE SHARING ALGORITHM is independent from the ordering of customers within a phase, this ordering can considerably speed up or slow down convergence in practice on instances of the GLOBAL ROUTING PROBLEM. In this context, it is advantageous to process nets in the order of non-decreasing estimated wiring length (given e.g. by the half perimeter of the bounding box or an initial Steiner tree computed for each net). The reason is that with longer connections there is more flexibility for evading congested areas without a large detour. Therefore, with $\varepsilon_1 > 0$, less nets have to be rerouted in each phase. Of course, in the PARALLEL RESOURCE SHARING ALGORITHM, there is no total order in which nets are processed. In this case, nets should appear in the order of non-decreasing estimated wiring length in the task list which is processed in parallel by fetching new tasks from the beginning of the list.

Setting the Bound Γ^{obj} on Objective Costs

Let $(T_N, s_N)_{N\in\mathcal{N}}$ be a feasible solution to the GLOBAL ROUTING PROBLEM minimizing (4.1), and let γ^* be the value of (4.1) in this solution. Instead of narrowing down γ^* by binary search, we set Γ^{obj} to an empirically determined estimate $k_1\gamma_{\mathrm{lb}}$ of γ^* and adapt it after each phase of the RESOURCE SHARING ALGORITHM, where

$$\gamma_{\mathrm{lb}} := \sum_{N\in\mathcal{N}} \min_{T\in\mathcal{T}_N} \sum_{e\in E(T)} \inf_{s\ge 0} \gamma^{\mathrm{obj}}_{N,e}(s) \tag{4.12}$$

and $k_1 \in \mathbb{R}_+$. The ratio $\gamma^*/\gamma_{\mathrm{lb}}$ of course depends highly on the functions $\gamma^{\mathrm{obj}}_{N,e}$ for $N \in \mathcal{N}$ and $e \in E(G)$, and thus on the optimization objective:

Total wiring length: The solution $T \in \mathcal{T}_N$ for net $N \in \mathcal{N}$ in (4.12) uses only the lowest two wiring layers in practice if N is not restricted to run only on higher layers and has pins only in the lowest layer (this is the case for the majority of nets). However, for minimization of total wiring length, via edges are usually assigned a length value (specified as parameter) which is considerably larger than the physical length of a via, and not all nets can be routed on the lowest two layers at the same time because of edge capacity restrictions. Still, the ratio $\gamma^*/\gamma_{\mathrm{lb}}$ is small in most cases and typically within $[1.0, 1.05]$. Exceptions are chips which can be routed only with considerable detours in a large number of nets.

Critical area: The ratio $\gamma^*/\gamma_{\mathrm{lb}}$ in most cases is considerably greater for critical area minimization than for wiring length minimization. One reason for this is that wiring length, in contrast to critical area, of course does not depend on spacing. Larger wire widths on higher routing layers contribute even more to the increase of $\gamma^*/\gamma_{\mathrm{lb}}$: The contribution to open critical area of a wire on a higher routing layer can be much less than that of a wire on a lower routing layer because of increased metal width, and also the contribution to short critical area can reduce drastically, even at zero extra spacing, because the higher metal width imposes greater minimum distance requirements. Although using higher metal layers causes extra costs because more vias are required, typically the demand for routing space on the desired layers in (4.12) significantly exceeds the available space. Typically, $\gamma^*/\gamma_{\mathrm{lb}} \in [1.05, 1.5]$ for the 65 nanometer chips in our tests, for example.

Power: For power optimization, the ratio $\gamma^*/\gamma_{\mathrm{lb}}$ typically is much smaller than for critical area minimization, but larger than for wiring length minimization. The differences between routing layers are rather small (greater wire widths in high routing layers are compensated by higher distance to the substrate), but assigning extra space can reduce coupling capacitance significantly. Typically, $\gamma^*/\gamma_{\mathrm{lb}} \in [1.0, 1.2]$.

Let now r_{obj} be the resource which models the optimization objective, and let

$$\lambda^{(p)}_{\mathrm{obj}} := \frac{\alpha^{(p)}_{r_{\mathrm{obj}}}}{p} \tag{4.13}$$

be the congestion of this resource after phase p of the RESOURCE SHARING ALGORITHM, $p \geq 1$. We choose k_1 as the maximum of the above intervals depending on the optimization objective, or slightly larger, and starting with a value $\Gamma^{\mathrm{obj}} = k_1 \gamma_{\mathrm{lb}}$, multiply Γ^{obj} by $\frac{\lambda^{(p)}_{\mathrm{obj}}}{k_2}$ at the end of phase p for some $k_2 \in \mathbb{R}_+$, implicitely changing $g_c(b)_{r_{\mathrm{obj}}}$ for each customer $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$. In our experiments, we chose $k_2 := 0.95$. This keeps congestion of r_{obj} approximately at 0.95 throughout the RESOURCE SHARING ALGORITHM. In regions of a chip with very low edge congestion, reducing the objective value becomes the primary goal, and in regions with edge congestion above 0.95 the algorithm rather works towards feasibility (similarly for delay constraints). Convergence to a feasible fractional solution to the GLOBAL ROUTING PROBLEM is much faster with $k_2 := 0.95$ than with $k_2 := 1.0$ in practice.

4.7.2 Parallelization

The global router in BonnRoute® is an implementation of the PARALLEL RESOURCE SHARING ALGORITHM presented in section 3.3. An important aspect of the implementation is to reduce the probability of *collisions* in the sense that block solvers concurrently executed by two threads work on nets in the same region of a chip and simultaneously use some edge without seeing the other's resource allocation. This leads to rejections of solutions and sequential reprocessing at the end of a phase. To reduce collision probability, each thread tries to choose nets (or customers) to process such that the search spaces of different threads overlap as little as possible.

Much time can be saved by choosing clusters of nets instead of individual nets. BonnRoute® therefore creates a partition $\mathcal{J}_1 \dot\cup \ldots \dot\cup \mathcal{J}_k = \mathcal{N}$ of the global routing nets or customers, where $k \in \mathbb{N}$ and $\mathcal{J}_1, \ldots, \mathcal{J}_k$ are called *jobs*. Each job $\mathcal{J}_i$, $1 \leq i \leq k$, has a *job area* which is an extended bounding box of the pins of the nets in $\mathcal{J}_i$. The jobs are inserted into a list at the beginning of each phase of the algorithm, and each of the Π worker threads ($\Pi \in \mathbb{N}$) concurrently deletes a job from the beginning of the list and processes the customers contained in it, until the list is empty. Instead of always choosing the first job in the list, a thread determines a job $\mathcal{J}_i$ among the first l jobs in the list ($l \in \mathbb{N}$) whose job area has smallest overlap with the areas of jobs currently processed by other threads.

Although this can be done in principle without a clustering of nets, i.e. $k = |\mathcal{N}|$, a partitioning based on geometric clustering offers several advantages:

1. Nets in the same region of a chip are processed consecutively by the same processor. This improves caching efficiency, even if $\Pi = 1$, because of overlapping search space in the block solver calls for these nets.

2. The number of accesses to the job list decreases significantly, so instead of a more complicated lock-free list data structure as e.g. proposed by Sundell and Tsigas [2005], a normal linked list protected by mutex locking can be used almost without performance loss.

3. Checking overlap of job areas has to be done less often, again reducing contention on the job list (threads cannot remove items from the list while other threads scan the first l items to determine a best next job to process)

4. While a thread determines the next job to process, currently active jobs might complete. This leads to superfluous overlap checks, and probability of this situation is reduced with larger job sizes.

Because shorter connections should be routed before longer connections, global routing nets are first partitioned into *length classes* based on estimated wiring length, and then a geometrical clustering is done within each length class. With typical wiring length distributions, it is natural to choose exponentially increasing length class boundaries allowing estimated wiring length within a class to differ by some constant factor (in our experiments we used a factor of two). Geometrical clustering of nets within the same length class is done based on the *center of mass* of each net (computed from the pins of the net, or wires in initial Steiner forests). The chip area is recursively subdivided by horizontal or vertical lines until the total estimated wiring length of nets with their center of mass contained in an area falls below a certain minimum desired *job size*. Of course estimated wiring length is only a very rough measure for the computational effort needed to process the corresponding set of nets in the PARALLEL RESOURCE SHARING ALGORITHM, but to even the processing times by this simple method helps sufficiently to avoid idle times at the end of a phase (threads are *idle* if there are other threads still working on a job, but there are no more jobs to be processed in the job list). Jobs within the same length class are ordered randomly, which provides sufficient probability in practice for each thread to find a job among the first l jobs in the list with little or no overlap with jobs currently processed by other threads, for small values of l. In our experiments, we chose $l := 4$.

It turns out that with this simple strategy for collision reduction, sequential recomputations are necessary only very rarely (see experimental results in section 4.9). Therefore BonnRoute® offers a parallelization mode in which all solutions returned by block solvers are accepted, and no sequential recomputation is done at the end of a phase. In this parallelization mode, of course also the *check* if a solution is to be rejected can be skipped, so the vector α' of tentative resource usages does not need to be updated (see the PARALLEL RESOURCE SHARING ALGORITHM on page 76). The resulting reduced number of write operations on memory saves in fact much more time than only eliminating sequential recomputations. While speedups with sequential recomputations are impressive already, they improve even further in this "unsafe" parallelization mode. Experimental results show no significant differences in the quality of results.

Deterministic Parallelization

The PARALLEL RESOURCE SHARING ALGORITHM is not deterministic because resource utilizations observed by threads depend on thread execution speeds, which are impossible to control in practice. The clustering of nets described above can also be used for a *deterministic* parallelization approach. In this approach, each thread has its own vector of resource utilizations, and increments are communicated at certain synchronization points between threads. The sets of customers to be processed between two synchronization points are determined a priori and not changed before the next synchronization point. Disjoint search spaces in different threads are of course even more important compared to the non-deterministic parallelization described above because resource allocations by other threads are seen with a much higher delay. Another challenge is reducing idle times: Threads cannot be dynamically assigned new jobs if they finished their job early, but other threads take longer than expected. Good estimates of processing times are essential, but difficult to obtain. Increasing job sizes helps to even out wrong estimates of processing times, but increase the delay with which resource allocations of other threads become visible to a thread.

Determinism is of course desirable for debugging, but otherwise *stability* w.r.t. quality of the returned solution is sufficient. As there are no significant deviations between different runs of BonnRoute® with non-deterministic parallelization, we do not further pursue deterministic parallelization.

Memory Management

The functions for dynamic memory allocation provided by system libraries often do not scale well with the number of threads simultaneously using them. While there are approaches to memory allocation that provide a significantly better scaling (see e.g. Michael [2004]), we use independent memory management for each thread (only occasionally calling a system library function to obtain a large block of memory which is internally subdivided into smaller blocks to satisfy memory allocation requests). A drawback of this is that some data have to be *moved between contexts*, i.e. copied into memory allocated by the memory manager of another thread, because memory blocks have to be freed by the same memory manager that allocated them, and (most of the time) a thread must not access the memory manager of another thread. As this affects only a rather limited amount of data in our case, independent memory management in each thread is the easiest way to overcome parallelization bottlenecks in memory management.

4.7.3 The Standard Block Solver

The standard block solver used in BonnRoute® does not support topology restrictions and finds a Steiner forest for $N \in \mathcal{N}$ satisfying the connectivity requirement stated in definition 4.5. Given edge costs $c : E(G) \to \mathbb{R}_+$, it heuristically tries to find a Steiner forest $F \in \mathcal{T}_N$ for a net $N \in \mathcal{N}$ in a global routing graph G with near-optimum cost $\sum_{e \in E(F)} c(e)$ by a series of shortest-path computations in (G, c), using a variant of the algorithm of Dijkstra [1959] to find shortest paths between *vertex sets* $S, T \subseteq V(G)$ as shown in section 3.3.2. An approximation guarantee cannot be given in general, but it turns out that this approach works surprisingly well in practice. Details are given below.

Edge Costs

When routing net $N \in \mathcal{N}$, edge costs are given implicitely by

$$c(e_i) := \inf_{s \geq 0} \left(\omega_i \frac{w(N, e_i) + s}{u(e_i)} + \sum_{1 \leq j \leq q} \omega_{m+j} \frac{\gamma_{N,e_i}^{P_j}(s)}{\Gamma^{P_j}} + \omega_{m+q+1} \frac{\gamma_{N,e_i}^{\text{obj}}(s)}{\Gamma^{\text{obj}}} \right)$$

for each edge $e_i \in E^+$, where ω_i is the price for using one unit of the edge capacity resource of edge e_i, $1 \le i \le m := |E^+|$, ω_{m+j} is the unit price of delay on the critical path P_j, $1 \le j \le q$, and ω_{m+q+1} the unit price of the objective function resource (see section 4.2). Due to the modeling inaccuracies discussed in section 4.4.1, sometimes a small number of nets can only be routed using zero capacity edges, and the block solver should minimize the number of zero capacity edges used in these cases. With $c^{\max} := \max\{c(e) : e \in E^+\}$, we therefore set

$$c(e) := kc^{\max}$$

for each edge $e \in E(G) \setminus E^+$, where $k \in \mathbb{R}_+$ is a large constant. Although

$$k > \frac{1}{c^{\max}} \sum_{e \in E^+} c(e)$$

is required to minimize the number of zero capacity edges in a shortest path, we set $k := 100$ for the reasons discussed in section 4.7.1.

Decomposition into Shortest-Path Computations

In the initialization of the global router, we compute a near-optimum rectilinear (group) Steiner tree for each net $N \in \mathcal{N}$, where center points of tiles covered by each detailed routing pin (projected into the x-y-plane) define the groups to be connected. Optimum solutions can be found fast for small numbers of groups each consisting of only one point (as is the case for the majority of nets), and fast heuristics are used in the other cases. See Robins and Zelikovsky [2009] for an up-to-date survey on Steiner tree algorithms. With groups originating from hard pins, the solutions are of course Steiner forests in general.

From the resulting Steiner points, artificial global routing pins are created and added to the corresponding net. These *Steiner pins* are defined to be soft pins and extend over all routing layers. They are not required to be connected in order to avoid unnecessary congestion in areas with high utilization of edge capacity resources.

The STANDARD BLOCK SOLVER algorithm, called for a net $N \in \mathcal{N}$, iteratively connects *connected components* with each other by performing shortest-path computations, starting with single global routing pins of N and stopping as soon as all non-Steiner pins are in the same component (see below). For convenient notation, let $N^{\text{Steiner}} \subseteq N$ be the Steiner pins in N. For a set C of global routing pins, we write

$$V(C) := \begin{cases} \bigcup_{p \in C} V(p) & : \ |C| \le 1 \\ \bigcup_{p \in C : p \text{ hard}} V(p) & : \ |C| > 1 \end{cases}$$

Further, we require $V(P) \neq \emptyset$ for any path P in G. Specifically, for $S, T \subseteq V(G)$ with $S \cap T \neq \emptyset$, we need $V(P) \cap S \cap T \neq \emptyset$ for an S-T-path P in G.

Proposition 4.11. *The* STANDARD BLOCK SOLVER *works correctly.*

```
STANDARD BLOCK SOLVER
   Input  : An undirected connected graph $G$, edge costs $c : E(G) \to \mathbb{R}_+$, and a
            global routing net $N \in \mathcal{N}$ in $G$.
   Output: A feasible Steiner forest $F$ for $N \setminus N^{\text{Steiner}}$ in $G$.
 1 Set $\mathcal{C} := \{\{p\} : p \in N\}$, $V(F) := \emptyset$ and $E(F) := \emptyset$.
 2 while $\nexists C \in \mathcal{C} : (N \setminus N^{Steiner}) \subseteq C$ do
 3     Choose a $C \in \mathcal{C}$ with $C \cap (N \setminus N^{\text{Steiner}}) \neq \emptyset$ and $|V(C)|$ minimum.
 4     Let $S := V(C)$ and $T := \bigcup_{C' \in \mathcal{C} \setminus \{C\}} V(C')$.
 5     Find a shortest $S$-$T$-path $P$ in $(G, c)$.
 6     Choose $C' \in \mathcal{C} \setminus \{C\}$ with $V(P) \cap V(C') \neq \emptyset$.
 7     Choose an unused identifier $n \in \mathbb{N}$ and set $C_{\text{new}} := C \cup C' \cup \{(V(P), \text{hard}, n)\}$.
 8     Set $\mathcal{C} := (\mathcal{C} \setminus \{C, C'\}) \cup \{C_{\text{new}}\}$.
 9     Set $V(F) := V(F) \cup V(P)$ and $E(F) := E(F) \cup E(P)$.
```

Proof. Assume that for each $D \in \mathcal{C}$ with $|D| > 1$, F is a feasible Steiner forest for D at the beginning of a specific iteration of the **while**-loop. By the definition of $V(C)$ for a set C of global routing pins, $(V(F) \cup V(P), E(F) \cup E(P))$ is therefore a feasible Steiner forest for the component C_{new} constructed in this iteration. As $|D| = 1$ for each $D \in \mathcal{C}$ in the first iteration, the statement hence follows by induction. □

We do not choose Steiner pins as source components in line 3 because this would enforce to connect them. Choosing $C \in \mathcal{C}$ with $|V(C)|$ minimum is not required for correctness, but increasing the number of target vertices T often reduces the number of label operations in Dijkstra's algorithm.

Although an optimality guarantee cannot be given in general for the STANDARD BLOCK SOLVER, experiments indicated that total objective cost and edge congestion in a solution to the SIMPLIFIED GLOBAL ROUTING PROBLEM do not improve significantly when using an exact graph Steiner tree algorithm instead of the STANDARD BLOCK SOLVER for nets with ≤ 5 terminals. The STANDARD BLOCK SOLVER is considerably faster however in practice.

Goal-Oriented Path Search

Let $S \subseteq V(G)$ be a set of *source* vertices, and $T \subseteq V(G)$ *target* vertices. A standard approach to speed up the search for a shortest path from S to T with Dijkstra's algorithm is to compute lower bounds $\pi : V(G) \to \mathbb{R}_+$ on the length $\text{dist}_c(v, T) := \min\{\sum_{e \in E(P)} : t \in T, P \text{ shortest } v\text{-}t\text{-path in } (G, c)\}$ of a shortest path from v to T for each $v \in V(G)$, and to use *reduced edge costs*

$$c_\pi((v, w)) := c(e) - \pi(v) + \pi(w)$$

if the undirected edge $e = \{v, w\} \in E(G)$ is traversed from v to w, and $c_\pi((w, v))$ if traversed in the other direction (formally, each undirected edge $e \in E(G)$ is replaced by two oppositely directed edges $\overleftarrow{e}$ and $\overrightarrow{e}$ which share the same edge capacity resource). π is required to be a *feasible potential*, which means that $c_\pi((v,w)) \geq 0$ for each $\{v,w\} \in E(G)$ to ensure that shortest paths w.r.t. c_π are shortest paths w.r.t. c and vice versa.

Many global routing nets have the same wire widths and objective cost functions on all edges of the global routing graph because the corresponding detailed routing nets have the same wire type. Denote this set of nets $\mathcal{N}' \subseteq \mathcal{N}$ and assume for simplicity that we have no delay bounds and the sequential RESOURCE SHARING ALGORITHM is run. Then edge costs in subsequent block solver calls for nets in $\mathcal{N}'$ do not decrease because ω does not decrease, and hence a feasible potential computed at some point in time for routing a net $N_1 \in \mathcal{N}'$ remains feasible when routing another net $N_2 \in \mathcal{N}'$ at a later time.

Of course the target vertex set T changes in different shortest-path computations, so instead of storing lower bounds on the distance to a vertex set T for each vertex $v \in V(G)$, we compute and store distances $d_c^i(v) := \mathrm{dist}_c(v, \{l_i\})$ for each $v \in V(G)$, edge costs $c : E(G) \to \mathbb{R}_+$ and a small set of *landmark vertices* $l_i \in V(G)$, $1 \leq i \leq h$, $h \in \mathbb{N}$, and use the triangle inequality to obtain a feasible potential

$$\pi(v) := \max_{1 \leq i \leq h} \max \left\{ d_c^i(v) - \max_{t \in T} d_c^i(t),\ \min_{t \in T} d_c^i(t) - d_c^i(v) \right\}$$

for a shortest-path computation with target vertices $T \subseteq V(G)$ and edge costs c, or edge costs $c' : E(G) \to \mathbb{R}_+$ with $c'(e) \geq c(e)$ for each $e \in E(G)$. The values $\min_{t \in T} d_c^i(t)$ and $\max_{t \in T} d_c^i(t)$ for $1 \leq i \leq h$ can be determined at the beginning of the path search, so in each labeling step only $2h$ subtractions and max-operations have to be performed to obtain a lower bound on $\mathrm{dist}_{c'}(v, T)$ for some $v \in V(G)$. This technique of goal-oriented path search using precomputed distances to a small set of landmark vertices is due to Goldberg and Harrelson [2005]. In our case, landmark distances $\mathrm{dist}_c(v, \{l_i\})$ have to be recomputed from time to time as edge costs c increase and lower bounds gradually become weaker. In BonnRoute®, we use 4 landmark vertices in the "corners" of the global routing graph.

An advantage of the landmark-based lower bounding technique is that it takes edge congestion costs into account. Often, a lower bound ignoring edge congestion can be computed very simply from geometric distance between vertices w.r.t. a natural embedding of the graph vertices into $\mathbb{R}^3$ (provided by global routing tile centerpoints in case of a global routing grid graph). Taking edge congestion into account turns out to be very effective if critical area is to be minimized because in this case the demand for routing on the highest layers exceeds the available routing space by far because of large wire widths, resulting in considerable congestion also on easily routable chips.

Volatility Tolerance

The block solver presented in this section is not volatility-tolerant because volatility tolerance requires a block solver to return a $(1 + \varepsilon_0)$-optimal solution w.r.t. observed edge costs (see definition 3.16 on page 72), but we do not know ε_0 even for constant edge costs. Nonetheless, our standard block solver returns the sum of observed path costs, so assuming that these costs are $(1+\varepsilon_0)$-optimal w.r.t. observed edge costs for a given $\varepsilon_0 \geq 0$, the PARALLEL RESOURCE SHARING ALGORITHM can check if resource costs have increased too much until termination of the call to the block solver. The experimental results in section 4.9 indicate that this happens only very rarely.

4.7.4 Randomized Rounding and Iterative Refinement

After a fractional solution to the GLOBAL ROUTING PROBLEM has been found by running the RESOURCE SHARING ALGORITHM, we generate an integral solution from it by randomized rounding as described in section 3.4. If $t \in \mathbb{N}$ phases were performed in the RESOURCE SHARING ALGORITHM, we randomly and independently for each net $N \in \mathcal{N}$ choose a phase index $1 \leq p \leq t$, selecting the solution used in phase p of the fractional algorithm for net N. Each phase index is chosen with probability $1/t$. We repeat this procedure several times and pick one of the integral solutions to the GLOBAL ROUTING PROBLEM obtained in this way. This solution of course violates some of the constraints in general, so an iterative refinement procedure (discussed below) tries to achieve feasibility changing the routing for nets involved in constraint violations. Instead of picking a randomized rounding solution with smallest maximum congestion, in practice picking a solution which minimizes the sum

$$\sum_{r\in\mathcal{R}} \max\{0, \lambda_r - 1\} \tag{4.14}$$

of constraint violations results in slightly less runtime spent in iterative refinement, where λ_r denotes the congestion of resource $r \in \mathcal{R}$ in the corresponding solution. As the variance of (4.14) over different randomly rounded solutions is low in practice, and reducing $\max_{r\in\mathcal{R}} \lambda_r$ is achieved faster by iterative refinement than by increasing the number of randomized rounding steps, only a few such steps are performed. In our experiments, we did eight randomized rounding steps.

Iterative refinement of a randomly rounded solution is done by a combination of choosing alternative routings from the fractional solution for nets involved in constraint violations, and computing new routings if constraint violations cannot be reduced by rechoosing from the fractional solution.

A basic subprocedure IMPROVE(r) is called iteratively, getting a resource $r \in \mathcal{R}$ with congestion $\lambda_r > 1$ as input and for one net N in the set $\mathcal{N}' \subseteq \mathcal{N}$ of nets with positive utilization of this resource tries to replace the current routing of N by another routing such that

$$\left(\max_{r\in\mathcal{R}} \lambda_r,\ \sum_{r\in\mathcal{R}} \max\{0, \lambda_r - 1\},\ \hat{\gamma} \right) \tag{4.15}$$

becomes lexicographically smaller, where $\hat{\gamma}$ is the objective value of the current global routing solution. The number of zero capacity edges is not considered here; each net is required to be routed with a minimum number of zero capacity edges. Three steps with increasing computational cost are performed to find an alternative routing for a net $N \in \mathcal{N}'$ such that (4.15) is improved:

1. First a small number of nets $N \in \mathcal{N}'$ is selected randomly, and for each of them an alternative routing from the fractional solution is chosen at random. If exchanging the current routing by the alternative improves on (4.15) for some of them, the best of these exchange operations is selected.

2. If the first step did not improve (4.15), all alternative routings in the fractional solution are evaluated for each $N \in \mathcal{N}'$.

3. If the first two steps did not improve (4.15), the block solver is called for each $N \in \mathcal{N}'$ to generate a new routing, using resource prices which depend exponentially on resource utilization as in the fractional RESOURCE SHARING ALGORITHM. As soon as a new routing is found for some $N \in \mathcal{N}'$ which improves (4.15), the procedure is stopped.

If step 3) did not improve (4.15), IMPROVE(r) *fails*. Otherwise it returns an alternative routing for some $N \in \mathcal{N}'$.

The overall iterative refinement procedure works multi-threaded as follows. A master thread assigns pairwise disjoint sets $R_i \subseteq \mathcal{R}$ of overcongested resources (i.e. resources with congestion greater than 1) to slave threads $1 \leq i \leq \Pi$, $\Pi \in \mathbb{N}$. Each thread $1 \leq i \leq \Pi$ executes IMPROVE(r) for each $r \in R_i$, but does not actually change the routing of any net in the current solution. After termination of the slave threads, the master thread evaluates the resulting candidate exchange operations in the order of non-increasing improvements which were observed by the slave threads, and applies each of them which still improves (4.15). All other candidate exchange operations are rejected. To keep the number of rejections small, not all overcongested resources are assigned to slave threads at once. We limit the sets R_i to contain at most a few hundred resources and distribute only the most congested resources to slave threads. This process is repeated iteratively until either a feasible solution to the GLOBAL ROUTING PROBLEM is found, or the observed improvement stays below some threshold over several iterations, or a time bound is exceeded.

Although the described procedure is a simple heuristic approach, it works reasonably well on state-of-the-art industrial chips.

4.7.5 Technical Aspects

We finally discuss some technical issues which we observed to impact performance significantly:

1. First of all, storing frequently accessed data memory-efficiently can improve caching performance considerably. For example, resource prices, represented by 64-bit floating numbers, are not stored explicitely, but computed on-the-fly from

resource utilizations which need only 32 bits per resource. In the global routing grid graph implementation, the graph structure is used to implicitely derive the neighbours of a vertex from an index assigned to it, instead of explicitly storing adjacency lists. Both techniques together reduce runtime by about 30 percent.

2. For obtaining good parallelization speedups, it is very important to eliminate concurrent accesses to the same *cache line* as far as possible. A cache line is an interval of k bytes in memory at an address divisible by k, where $k \in \mathbb{N}$ depends on the processor. For example, on current Intel processors $k = 64$. Elementary data types in most cases require less than k bytes (this is true for all elementary data types on Intel processors). For example, storing eight 32-bit counter variables accessed by different threads in successive entries of an array will most likely cause all of them to lie within the same cache line and thus significantly degrade speedup if these variables are accessed frequently. In this example, the problem can be easily eliminated by filling the array with unused entries.

4.8 Fast Tree Enumeration

In this section we present a data structure which, building on a global routing grid graph, allows extremely fast queries on *edge segments*:

Definition 4.12. *A* segment *is a connected set of edges contained in one of the sets E_z $(1 \le z \le z_{\max})$ or E_{via}. A maximal segment is called a* line.

This data structure allows to enumerate and determine the cost of a large number of trees in short time, providing a fast alternative to maze routing based on Dijkstra's algorithm as in the standard block solver presented in section 4.7.3. See section 4.8.1 below for further details.

Let $L = \{e_1, \dots, e_{|L|}\}$ be a line in G, with edge numbers such that e_j and e_{j+1} are incident for each $1 \le j < |L|$. Let $\omega \in \mathbb{R}^L$ be a vector of edge costs, and $S \subseteq L$ be a segment in L specified by numbers $0 \le j_{\min} \le j_{\max} \le |L|$ such that $S = \{e_j : j_{\min} < j \le j_{\max}\}$ ($S = \emptyset$ if $j_{\min} = j_{\max}$).

We present a data structure which allows to compute $\sum_{e \in S} \omega_e$ in $O(\log|S|)$ time (assuming $S \neq \emptyset$), and which can be updated in $O(|S| + \log|L|)$ time if ω is changed on edges in S.

Since the number of layers in G is very small in practice (up to 11 in current technologies), we will use this data structure only for segments of horizontal or vertical edges, but not for via edges.

Definition 4.13. *For $0 \le j \le |L|$, let $p^0(j) := j$ and*

$$p^k(j) := p^{k-1}(j) \otimes (p^{k-1}(j) - 1)$$

for $k \in \mathbb{N}$, where the $\otimes$-operator denotes the bitwise and operation.
For $0 \le j_1 \le j_2 \le |L|$, let $p^*(j_1, j_2) := \max\{p^k(j_2) : k \ge 0$ and $p^k(j_2) \le j_1\}$.

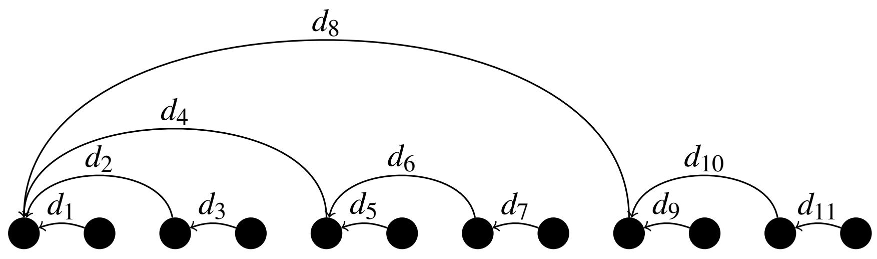


Figure 4.10: The underlying structure of the segment cost lookup table. Each arc spans the set of edges whose weights are summed up in the corresponding table entry.

We assume the standard binary representation of natural numbers. The representation of negative numbers does not matter since the only operation in which a negative number occurs is $0 \otimes -1 = 0$.

First observe some simple facts:

Lemma 4.14. *Let* $0 \leq j \leq |L|$.

(i) $p^k(j) \geq 0$ *for all* $k >= 0$.

(ii) $p^{k_1}(j) > p^{k_2}(j)$ *for all* $0 \leq k_1 < k_2$ *with* $p^{k_1}(j) > 0$.

(iii) $p^k(j) - p^{k+1}(j) \geq 2(p^{k-1}(j) - p^k(j))$ *for each* $k \geq 1$ *with* $p^k(j) > 0$.

(iv) If $p^0(j) - p^1(j) > 1$, *then* $j > 1$ *and* $p^0(j-1) - p^1(j-1) = 1$, *and furthermore* $p^k(j-1) - p^{k+1}(j-1) = 2(p^{k-1}(j-1) - p^k(j-1))$ *for all* $k \geq 1$ *with* $p^{k+1}(j-1) \geq p^1(j)$.

(v) For $0 \leq j_1 \leq j_2 \leq |L|$, *there is a number* $k \geq 0$ *such that* $p^k(j_1) = p^*(j_1, j_2)$

Definition 4.15. *Let L be a line in G and* $\omega \in \mathbb{R}^L$. *The* segment cost lookup table $T(L, \omega)$ *for L and* ω *is*

$$T(L, \omega) := \left\{d_0, \ldots, d_{|L|}\right\}$$

with

$$d_j := \sum_{p^1(j) < i \leq j} \omega_i$$

for each $0 \leq j \leq |L|$.

Fig. 4.10 visualizes the underlying structure of the segment cost lookup table. Obviously,

$$\sum_{e \in S} \omega_e = \sum_{0 \leq i \leq \lceil \log_2(j_{\max}) \rceil} d_{p^i(j_{\max})} - \sum_{0 \leq i \leq \lceil (\log_2 j_{\min}) \rceil} d_{p^i(j_{\min})},$$

and this sum can be computed in $O(\log |L|)$ time (actually, $\lceil \log_2(j_{\max}) \rceil$ and $\lceil (\log_2 j_{\min}) \rceil$ can be replaced by the number of one's in the binary representation of $j_{\max}$ and $j_{\min}$, respectively, but this is no improvement in the worst case). However, we can compute $\sum_{e\in S}\omega_e$ also in $O(\log|S|)$ time:

Theorem 4.16. *Using the segment cost lookup table, the total cost $\sum_{e\in S}\omega_e$ of segment S can be computed in $O(\log|S|)$ time.*

Proof. Let $k_{\max}$ be the smallest number such that $p^{k_{\max}}(j_{\max}) = p^*(j_{\min}, j_{\max})$. Then $k_{\max} \le \lfloor\log_2(|S|)\rfloor + 1$. If $j_{\min} - p^*(j_{\min}, j_{\max}) \le |S|$, then there is also a number $k_{\min} \le \lfloor\log_2(|S|)\rfloor + 1$ such that $p^{k_{\min}}(j_{\min}) = p^*(j_{\min}, j_{\max})$, and

$$\sum_{e\in S}\omega_e = \sum_{0\le i<k_{\max}} d_{p^i(j_{\max})} - \sum_{0\le i<k_{\min}} d_{p^i(j_{\min})}$$

can be computed in $O(\log|S|)$ time.

Otherwise, let $j_{\text{split}} := p^{k_{\max}-1}(j_{\max})$. It is easy to observe that $j_{\min} < j_{\text{split}} \le j_{\max}$. We split S into three segments $S_1 := \{e_{j_{\min}+1}, \dots, e_{j_{\text{split}}-1}\}$, $S_2 := \{e_{j_{\text{split}}}\}$ and $S_3 := \{e_{j_{\text{split}}+1}, \dots e_{j_{\max}}\}$, where S_1 and S_3 might be empty. Obviously, $\sum_{e\in S_3}\omega_e$ can be computed in $O(\log|S_3|)$ time. To see that $\sum_{e\in S_1}\omega_e$ can be computed in $O(\log|S_1|)$ time, assume that S_1 is not empty and observe that $p^0(j_{\text{split}}) - p^1(j_{\text{split}}) = p^{k_{\max}-1}(j_{\max}) - p^{k_{\max}}(j_{\max}) > |S| \ge 1$, so lemma 4.14 (iv) implies that $j_{\min} - p^*(j_{\min}, j_{\text{split}} - 1) \le |S_1|$, and the claim follows. □

The improvement from $O(\log|L|)$ to $O(\log|S|)$ is significant in practice since most nets are very short and thus require only very short segments.

We now show how to update the segment cost lookup table.

Theorem 4.17. *If $T(L,\omega)$ is the segment cost lookup table for a line L in G and $\omega \in \mathbb{R}^L$, and $\Delta \in \mathbb{R}^L$ with $\Delta_e = 0$ for $e \in L\setminus S$ is given for a segment $S \subseteq L$, the segment cost lookup table $T(L, \omega+\Delta)$ for L and $\omega+\Delta$ can be computed in $O(|S| + \log|L|)$ time.*

In particular, $T(L,\omega)$ can be computed from ω in $O(|L|)$ time.

Proof. Let again $0 \le j_{\min} \le j_{\max} \le |L|$ such that $S = \{e_{j_{\min}+1}, \dots, e_{j_{\max}}\}$. Let $d_0, \dots, d_{|L|}$ be the entries of $T(L,\omega)$. We show how the entries $d'_0, \dots, d'_{|L|}$ of $T(L, \omega+\Delta)$ can be computed.

Initially, set $d'_j := 0$ for each $0 \le j \le |L|$ (of course this does not have to be done explicitly). Then, for each $j_{\min} < j \le j_{\max}$ in increasing order do the following:

1) Set $d'_j := d'_j + \Delta_j$.

2) If $2j - p^1(j) \le |L|$, set $d'_{2j-p^1(j)} := d'_{2j-p^1(j)} + d'_j$.

For each $1 \le q \le |L|$, let $I_q := \{p^0(q-1), \dots, p^{k-1}(q-1)\}$, where k is the smallest number such that $p^k(q-1) = p^1(q)$ (so $I_q = \emptyset$ if q is an odd number). Inductively, the following two invariants are fulfilled immediately after each iteration $j_{\min} < j \le j_{\max}$ of the two steps above:

i) $d'_q = \sum_{p^1(q)<i\le q}\Delta_i = \Delta_q + \sum_{i\in I_q} d'_i$ for each $1 \le q \le j$

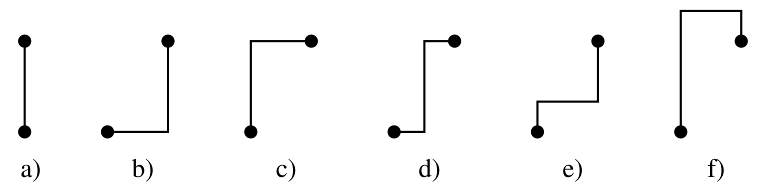


Figure 4.11: Pattern routing by at most three non-via segments for two-terminal connections: a) shows an I-pattern, b) and c) L-patterns, d) and e) Z-patterns and f) an U-pattern

ii) $d'_q = \sum_{i \in I_q \cap \{1,\dots,j\}} d'_i$ for each $j < q \le |L|$.

After all $j_{\min} < j \le j_{\max}$ have been processed, set $j := 2j_{\max} - p^1(j_{\max})$ and repeat the following steps as long as $j \le |L|$:

1) If $2j - p^1(j) \le |L|$, set $d'_{2j-p^1(j)} := d'_{2j-p^1(j)} + d'_j$.

2) Set $j := 2j - p^1(j)$.

Let j^t be the value of j in step 1) of the t-th iteration of this loop and assume $t \ge 2$. If a $(t+1)$-st iteration is performed, then $j^{(t+1)} - j^t \ge 2(j^t - j^{(t-1)})$, so at most $O(\log |L|)$ iterations are performed.

Finally, set $d'_j := d'_j + d_j$ for each $0 \le j \le |L|$. $\square$

4.8.1 Application

Not surprisingly, runtime of the standard block solver based on Dijkstra's algorithm depends very much on how long the connections are that have to be made. Of course also short connections can require a large number of labeling operations in presence of high edge congestion, but in most cases global routing runtime in BonnRoute® is dominated by block solver calls that have to find long paths. In practice, long connections still offer much flexibility to avoid congested regions with only small relative detour length if they are restricted to a small number of segments, i.e. a limit is imposed on the number of bends. E.g., restricting two-terminal nets to be routed by a path with at most three non-via segments (called I/L/Z/U pattern routing, see fig. 4.11) is a popular and successful approach in placement congestion estimation (see Shelar and Saxena [2009] or Menge [2008] for details and further references) and is also used in some global routers, see section 4.1.1.

Figure 4.12 shows the three-segment Steiner trees for a set of three terminal points (ignoring via segments). If the limit on the number of segments is raised to four or five in this example, the number of solutions increases drastically, of course. An alternative to full enumeration is to iteratively shift segments with high edge congestion costs such that a tree structure is preserved, as depicted in fig. 4.13. As segment costs have a length-dependent component because of the resource which models the optimization

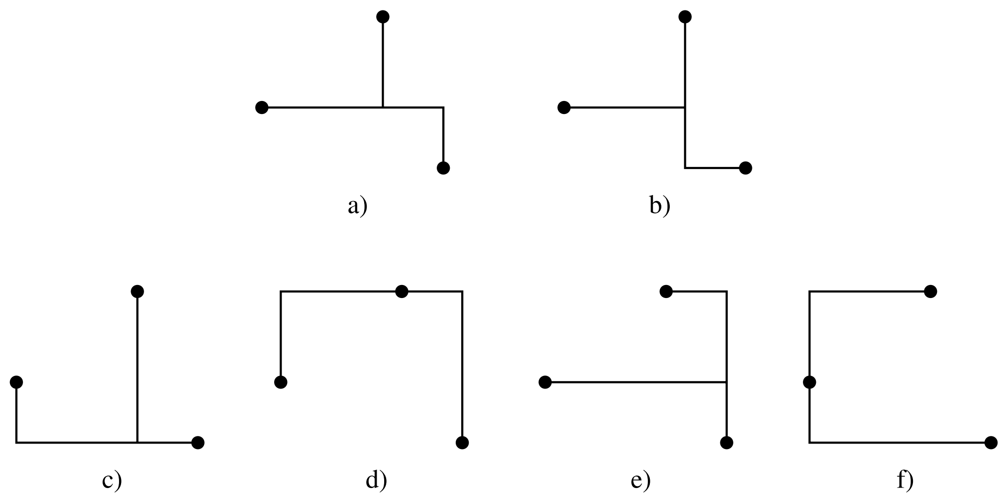


Figure 4.12: For a given set of three terminal points, a) and b) show the two shortest Steiner trees w.r.t. L_1-distance, each consisting of three axis-parallel line segments. c)-f) show all other Steiner trees with three axis-parallel line segments: c) and d) originate from a) by shifting the horizontal line segment, and e) and f) originate from b) by shifting the vertical line segment.

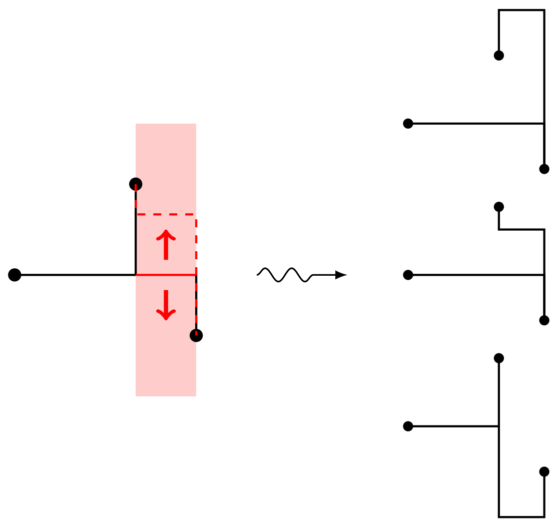


Figure 4.13: Three possible results of a segment shift operation. Length-dependent costs allow to limit the scope to the red-shaded area in this example.

objective, the detour incurred by a shift can be bounded. With a constant limit on the number of segments, each shift operation can be evaluated in $O(\log |S|)$ time using the the segment cost lookup table, where S is the longest segment that changes by a segment shift operation.

4.9 Experimental Results

In this section we present experimental results on recent industrial chips designed at IBM with the help of BonnTools. The problem sizes range from about hundred thousand to several million nets and edges in the global routing graph, see table 4.1. The table also shows process technology, die size, number of routing layers, number of nets, and total Steiner length, i.e. the sum of wiring lengths of a shortest Steiner tree for each net, for all chips in our testbed. In our experiments we do not consider delay bounds, so we have one resource for each edge in the global routing graph, and one resource for modeling the objective function.

Comparison with the Previous Version of BonnRoute®

We first compare our new global router based on the RESOURCE SHARING ALGORITHM presented in chapter 3 with the previous version of BonnRoute®, demonstrating that it converges to a near-optimum solution in considerably shorter time. Table 4.2 shows a comparison between the old and new version. The old implementation is based on the work of Albrecht [2001a,b] and neither supports power and yield optimization, nor 3D global routing. For this reason, the optimization objective in this comparison is wiring length, and both versions are run on a global routing grid graph in which all horizontal and vertical layers, respectively, are merged into a single layer. We only consider the computation of a fractional solution in this comparison. The bound Γ^{obj} on objective costs has been set to $1.03\gamma_{\text{lb}}$ initially for these experiments, and 20 phases of the RESOURCE SHARING ALGORITHM were performed with $\varepsilon_2 := 5.0$. In the table, "% Gap" denotes the percentaged deviation of the objective value γ_{fract} of the fractional solution achieved in the corresponding run from the lower bound γ_{lb}, i.e. the value

$$100\frac{\gamma_{\text{fract}} - \gamma_{\text{lb}}}{\gamma_{\text{lb}}}. \tag{4.16}$$

λ_{fract} denotes the maximum congestion in the fractional solution. It is the maximum of the congestion in the resource which models the objective function, and the maximum over edge congestion values. Both values are given additionally in parentheses.

The old and new version of BonnRoute® show a remarkable difference in (4.16); in most cases, the new version is better. On the other hand, maximum edge congestion values are considerably lower with the old version in most cases, leaving room for reducing wiring length. The reason for this difference is the weighting of resource prices as discussed in chapter 3 and section 4.7.1 of this chapter. Without this weighting, the

Chip	Technology	Die size [mm^2]	Routing layers	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{C}\rvert = \lvert\mathcal{N}\rvert$	Steiner length [m]
Lucius	65 nm	2.3 × 1.7	10	337,330	78,426	5.08
Heijo	65 nm	1.1 × 1.8	10	168,319	105,257	5.17
Ruediger	65 nm	2.1 × 0.8	10	164,740	103,899	5.61
Timo	65 nm	1.4 × 2.0	10	239,960	276,552	8.57
Jo	65 nm	3.1 × 1.6	10	419,005	515,044	26.70
Renate	65 nm	3.2 × 3.0	10	827,965	529,004	26.79
Rosemarie	65 nm	2.5 × 2.4	10	594,084	437,406	28.19
Tuula	65 nm	4.4 × 4.9	10	1,807,436	321,770	38.20
Georg	65 nm	2.7 × 4.1	10	1,132,940	783,685	43.28
Sigurd	65 nm	3.3 × 2.6	10	848,818	515,054	43.77
Wighart	65 nm	5.2 × 4.5	10	2,338,526	936,396	72.55
Claus	65 nm	4.9 × 3.3	10	1,590,875	709,007	78.31
Paula	65 nm	14.0 × 14.0	10	16,533,335	327,194	78.66
Dorothea	65 nm	6.1 × 6.1	10	3,647,013	679,820	79.26
Richard	65 nm	16.4 × 16.4	10	26,083,860	1,513,967	254.41
Laci	65 nm	18.0 × 18.8	10	32,652,236	1,300,577	271.12
Vasek	65 nm	18.8 × 18.8	10	36,519,998	434,945	280.97
Camilla	65 nm	10.0 × 10.0	10	10,255,304	3,582,559	316.65
Chiara	65 nm	14.8 × 14.0	10	20,757,260	3,293,378	401.79
Tomoko	65 nm	14.8 × 15.6	10	23,680,235	5,340,088	469.71
Andre	65 nm	15.6 × 14.8	10	23,923,565	7,039,094	589.83
Ludwig	65 nm	14.8 × 15.6	10	23,577,166	7,677,972	869.84
Raphaelo	45 nm	0.5 × 1.0	11	104,765	124,376	3.14
Nelu	45 nm	0.8 × 1.4	11	214,355	189,543	4.58
Gerhard	45 nm	2.3 × 1.7	11	749,359	348,860	11.13
Victor	45 nm	2.5 × 1.2	11	589,481	345,087	12.44
Henrik	45 nm	1.0 × 15.9	11	3,142,548	476,397	32.63
Emilia	45 nm	6.3 × 6.3	11	7,947,101	430,551	34.43
Milena	45 nm	3.7 × 2.9	11	2,101,489	1,314,074	41.58
Simona	45 nm	3.4 × 1.9	11	1,271,069	1,410,540	42.92
Angela	45 nm	6.0 × 6.0	11	7,040,649	1,008,736	59.35
Guido	45 nm	4.0 × 4.0	11	3,127,653	1,797,719	68.23
Dirk	45 nm	3.8 × 5.6	11	4,201,325	1,603,290	83.11
Thilo	45 nm	3.9 × 3.7	11	2,816,277	2,785,100	97.26
Martina	45 nm	5.1 × 5.3	11	5,329,900	3,788,281	151.46

Table 4.1: The testbed.

Chip	Old BonnRoute®			New BonnRoute®		
	λ_{fract} (obj./edges)	% Gap	Runtime	λ_{fract} (obj./edges)	% Gap	Runtime
Lucius	0.991 (0.991/0.932)	2.10	0:00:34	0.993 (0.993/0.961)	2.25	0:00:12
Ruediger	0.971 (0.971/0.889)	0.04	0:00:27	0.972 (0.972/0.925)	0.07	0:00:04
Timo	0.973 (0.973/0.902)	0.22	0:00:41	0.974 (0.974/0.937)	0.33	0:00:09
Jo	0.983 (0.983/0.885)	1.25	0:04:41	0.980 (0.980/0.949)	0.89	0:00:52
Renate	0.971 (0.971/0.845)	0.02	0:02:05	0.971 (0.971/0.925)	0.05	0:00:19
Rosemarie	0.974 (0.974/0.873)	0.29	0:03:27	0.973 (0.973/0.927)	0.24	0:00:21
Tuula	0.977 (0.977/0.887)	0.65	0:04:43	0.976 (0.976/0.941)	0.51	0:00:26
Georg	0.972 (0.972/0.836)	0.16	0:04:13	0.973 (0.973/0.931)	0.17	0:00:38
Sigurd	1.013 (1.013/0.923)	4.34	0:11:46	0.997 (0.997/0.978)	2.72	0:03:14
Claus	0.982 (0.982/0.865)	1.15	0:25:36	0.982 (0.982/0.971)	1.14	0:03:12
Dorothea	1.003 (1.003/0.884)	3.30	0:42:03	0.996 (0.996/0.967)	2.62	0:04:14
Laci	0.973 (0.973/0.840)	0.11	9:06:19	0.972 (0.972/0.968)	0.14	0:03:32
Vasek	0.980 (0.980/0.878)	0.92	7:50:39	0.996 (0.980/0.996)	0.96	0:16:00
Camilla	1.017 (1.017/0.821)	4.91	1:37:34	0.981 (0.981/0.968)	1.05	0:09:53
Chiara	0.987 (0.987/0.824)	1.74	2:42:07	0.981 (0.981/0.976)	1.07	0:13:49
Andre	1.039 (1.039/0.755)	7.11	4:45:33	0.980 (0.980/0.948)	0.91	0:13:59
Ludwig	1.167 (1.167/0.944)	20.60	16:13:09	0.989 (0.989/0.988)	1.84	0:51:06

Table 4.2: Comparison between the old and new BonnRoute® global router on 2D global routing instances. Runtimes (hh:mm:ss) are sequential on an Intel Xeon E7220 processor at 2.93 GHz and only include computation of the fractional solution.

cost of detours is marginal in the first phases of the algorithm, in particular on large chips where the number of nets is higher which share the resource for modeling the objective function.

This is also the most important reason for the better runtimes with the new version: A higher weight of the objective function resource yields better lower bounds for goal-oriented path search in the block solver and thus speeds up path search considerably.

Parallelization Speedup

Because of different layer characteristics, global routing has to be done without merging layers as in 2D global routing if power and yield, or a weighted sum of wiring length and number of vias is to be optimized. Also feasibility is affected because not all wire types have the same width ratios over different layers, and some nets usually are allowed to be routed only on a subset of layers.

Full 3D routing has a considerable impact on runtime, however: First, the number of vertices and edges in the global routing graph is 5-6 times higher than in a 2D global routing graph if global routing tiles have the same size. In addition to that, restrictions on usable wiring layers for some nets can cause congestion on some layers which cannot be distributed to other layers, and higher congestion not only causes more nets to be rerouted in each phase of the RESOURCE SHARING ALGORITHM, but also increases the runtime of the STANDARD BLOCK SOLVER based on Dijkstra's algorithm because the lower bounds for guiding path searches become weaker.

We show in tables 4.3 and 4.4 that the PARALLEL RESOURCE SHARING ALGORITHM scales very well with the number of processors and thus can reduce runtime considerably. All experiments have been performed on the original (3D) global routing grid graph, i.e. layers are not merged as above for the comparison with the old version of BonnRoute®, and wiring layer restrictions are respected for all nets.

The results shown in table 4.3 have been obtained on a machine with 8 Intel Xeon E7220 processors running at 2.93 GHz, and table 4.4 shows results on a machine with 16 slightly slower AMD Opteron 8384 processors, running at 2.7 GHz. As above, ε_2 is set to 5.0, and the number of phases is 20.

All runtimes in these tables are runtimes of the fractional (PARALLEL) RESOURCE SHARING ALGORITHM only; runtimes for randomized rounding and iterative refinement are not included here. The sequential runtimes reported in the second column are without parallelization overhead. In particular, normal addition operations are used instead of atomic addition. The third column shows runtimes with 8 threads in table 4.3 and 16 threads in table 4.4, and the corresponding parallelization speedups compared to sequential runtimes. In these runs, atomic addition is used, but no sequential recomputation of rejected solutions is performed. Consequently, also the checking if a solution is to be rejected is not done. The last three columns show results for a strict implementation of the PARALLEL RESOURCE SHARING ALGORITHM, including sequential recomputation of rejected solutions. Column 4 again shows runtimes and corresponding parallelization speedups. Column 5 shows the average percentage of nets over all

Chip	1 thread	8 threads, no seq. recomp.		8 threads,		seq. recomp.: % of nets	seq. recomp.: seconds
Lucius	0:00:50	0:00:07	(6.68x)	0:00:08	(6.21x)	0.0135	0.1
Heijo	0:00:07	0:00:02	(3.58x)	0:00:03	(2.29x)	0.0456	0.1
Ruediger	0:00:06	0:00:01	(3.55x)	0:00:03	(2.18x)	0.0505	0.0
Timo	0:00:12	0:00:04	(2.93x)	0:00:07	(1.66x)	0.0157	0.0
Jo	0:02:57	0:00:27	(6.43x)	0:00:32	(5.46x)	0.0054	0.2
Renate	0:00:26	0:00:08	(3.26x)	0:00:13	(1.97x)	0.0130	0.0
Rosemarie	0:00:49	0:00:09	(5.14x)	0:00:13	(3.58x)	0.0106	0.1
Tuula	0:01:20	0:00:13	(5.94x)	0:00:16	(4.83x)	0.0076	0.1
Georg	0:01:05	0:00:14	(4.59x)	0:00:22	(2.93x)	0.0068	0.1
Sigurd	0:21:43	0:02:49	(7.70x)	0:02:52	(7.53x)	0.0013	0.2
Wighart	0:03:44	0:00:36	(6.18x)	0:00:46	(4.84x)	0.0031	0.3
Claus	0:17:14	0:02:13	(7.76x)	0:02:25	(7.10x)	0.0041	3.7
Paul	0:12:29	0:03:28	(3.60x)	0:03:42	(3.38x)	0.0037	9.9
Dorothea	0:23:17	0:03:11	(7.29x)	0:03:25	(6.80x)	0.0038	6.6
Richard	0:30:28	0:04:34	(6.65x)	0:04:54	(6.21x)	0.0012	2.3
Laci	0:36:35	0:04:57	(7.38x)	0:05:21	(6.84x)	0.0018	2.8
Vasek	7:16:06	0:57:39	(7.56x)	1:00:07	(7.25x)	0.0026	44.6
Camilla	0:56:50	0:07:10	(7.92x)	0:07:37	(7.45x)	0.0004	0.4
Chiara	0:27:00	0:03:53	(6.93x)	0:04:28	(6.03x)	0.0007	0.4
Tomoko	0:12:29	0:02:19	(5.36x)	0:03:18	(3.78x)	0.0008	0.1
Andre	1:04:37	0:09:14	(7.00x)	0:10:17	(6.28x)	0.0003	0.4
Ludwig	7:58:22	0:57:28	(8.32x)	0:59:08	(8.09x)	0.0000	0.6

Table 4.3: Running times (hh:mm:ss) of the fractional resource sharing algorithm, and parallelization speedups on a machine with 8 Intel Xeon E7220 processors at 2.93 GHz.

Chip	1 thread	16 threads, no seq. recomp.		16 threads,		seq. recomp.: % of nets	seconds
Lucius	0:01:08	0:00:06	(10.97x)	0:00:07	(9.45x)	0.0331	00.4
Heijo	0:00:10	0:00:02	(3.89x)	0:00:04	(2.69x)	0.1023	00.2
Ruediger	0:00:09	0:00:02	(4.39x)	0:00:03	(3.01x)	0.0976	00.1
Timo	0:00:16	0:00:04	(3.35x)	0:00:07	(2.19x)	0.0369	00.2
Jo	0:04:05	0:00:29	(8.27x)	0:00:33	(7.38x)	0.0119	00.7
Renate	0:00:37	0:00:07	(4.80x)	0:00:12	(3.14x)	0.0296	00.1
Rosemarie	0:01:06	0:00:09	(7.31x)	0:00:12	(5.33x)	0.0241	00.3
Tuula	0:01:59	0:00:13	(9.00x)	0:00:16	(7.28x)	0.0164	00.7
Georg	0:01:28	0:00:13	(6.69x)	0:00:19	(4.51x)	0.0144	00.2
Sigurd	0:30:34	0:02:29	(12.30x)	0:02:35	(11.81x)	0.0031	01.2
Wighart	0:05:26	0:00:31	(10.34x)	0:00:40	(8.11x)	0.0069	01.3
Claus	0:25:31	0:01:44	(14.69x)	0:02:14	(11.42x)	0.0086	15.5
Paul	0:21:34	0:05:57	(3.62x)	0:06:44	(3.20x)	0.0103	28.8
Dorothea	0:34:44	0:02:32	(13.69x)	0:03:12	(10.84x)	0.0087	29.5
Richard	0:44:41	0:04:03	(11.03x)	0:04:39	(9.59x)	0.0034	20.2
Laci	0:55:04	0:03:58	(13.85x)	0:04:36	(11.94x)	0.0045	23.0
Vasek	11:23:42	0:44:08	(15.49x)	0:51:31	(13.27x)	0.0090	209.0
Camilla	1:21:15	0:05:34	(14.55x)	0:06:01	(13.48x)	0.0009	01.1
Chiara	0:40:16	0:03:15	(12.38x)	0:03:40	(10.94x)	0.0016	02.0
Tomoko	0:17:44	0:02:01	(8.75x)	0:02:42	(6.57x)	0.0015	00.3
Andre	1:35:17	0:07:30	(12.69x)	0:08:27	(11.27x)	0.0006	01.6
Ludwig	10:25:43	0:44:21	(14.11x)	0:46:04	(13.58x)	0.0001	03.3

Table 4.4: Running times (hh:mm:ss) of the fractional resource sharing algorithm, and parallelization speedups on a machine with 16 AMD Opteron 8384 processors at 2.7 GHz.

phases which have been rerouted sequentially, and column 6 shows the runtime used for sequential rerouting. Note that the difference between columns 3 and 4 is considerably higher than the time needed for sequential rerouting. This difference is caused by the checking if a solution has to be rejected.

The tables show that the PARALLEL RESOURCE SHARING ALGORITHM scales very well with the number of processors on many chips. On chips with a bad speedup, sequential runtime is already low in relation to wiring length (cf. table 4.1). On such instances, only few nets are rerouted on average in each phase, and a significant amount of runtime is spent on write operations to update resource utilizations. This of course has to be done for all nets, including those that are not rerouted. Write operations on memory thus seem to cause a bottleneck for parallelization speedup. In addition, unparallelized parts of the program code gain a larger fraction of runtime on instances which run fast already sequentially.

On the other hand, speedup on instances with high sequential runtime in relation to wiring length comes always close to the number of processors in table 4.3 if no sequential recomputation of rejected solutions is done at the end of a phase. Even with sequential recomputation, speedup is not much worse. Parallelization speedup on the chip Ludwig even exceeds the number of processors in table 4.3. There are two possible reasons for this:

1. In the PARALLEL RESOURCE SHARING ALGORITHM, nets are not routed in the same order as in the sequential RESOURCE SHARING ALGORITHM (in fact, there is no total order), but the ordering can have a significant impact on runtime in practice (see the discussion in section 4.7.1).

2. In spite of geometrical clustering, the search space for nets routed on different processors may overlap. In such cases, a processor may save a memory access by reading a variable from a shared cache which another processor previously fetched from memory. This benefit may compensate part of the disadvantage due to increased probability of write access collisions when search spaces overlap.

The results with 16 processors in table 4.4 show a slightly worse ratio between parallelization speedup and number of processors, but using 16 processors still improves speedup considerably compared to 8 processors.

We finally remark that table 4.3 shows that the new BonnRoute® global router already with 1 thread is faster on almost all 3D instances than the old global router on the corresponding 2D instances, despite the increased graph size.

Quality of Results and the Approximation Parameter

Tables 4.5 and 4.6 show the quality of results in wiring length optimization obtained for different settings of the approximation parameter ε_2. By the standard parameter settings used in the last years with BonnRoute®, vias are weighted relative to wiring length by assigning them a length corresponding to 7-11 routing tracks, depending on the layer.

The number t of phases is always chosen such that $\varepsilon_2 t = 100$. All runs in these tables were done multi-threaded using 8 Intel Xeon E7220 processors running at 2.93 GHz. As for the parallelization speedup tests, all experiments have been performed on the original (3D) global routing grid graph. The bound Γ^{obj} on objective costs has been set to $1.1\gamma_{\text{lb}}$ initially. Because considerably more vias are needed in 3D global routing instances compared to 2D instances, a bound of $1.03\gamma_{\text{lb}}$ as in the 2D comparisons above would sometimes be exceeded.

As above, λ_{fract} in column 4 is the maximum congestion over all resources in the fractional solution, and the breakdown on the maximum over edge capacity resources and the objective function resource is given in parentheses. λ_{lb} is a lower bound on the maximum congestion in an optimum solution obtained by applying the weak duality Lemma 3.2 on the resource prices returned by the RESOURCE SHARING ALGORITHM. λ_{rounded} is the maximum congestion after randomized rounding, and λ_{final} after iterative refinement by rechoosing and ripup-and-reroute. Column 8 shows the *total overload* of the final integral solution (abbreviated TOL), which is the sum

$$\sum_{r\in\mathcal{R}_{\text{edges}}} \max\{0, \lambda_r - 1\},$$

where λ_r is the congestion of resource $r \in \mathcal{R}$, and $\mathcal{R}_{\text{edges}}$ is the set of resources corresponding to global routing edge capacities.

Columns 9 and 10 show the percentaged deviation of the objective value γ_{fract} of the fractional solution and the final integral solution, respectively, from the lower bound γ_{lb} on the objective value. Finally, the last column shows the runtime needed altogether for computing initial Steiner trees, running the PARALLEL RESOURCE SHARING ALGORITHM, randomized rounding and iterative refinement.

Not surprisingly, runtime increases considerably when running the PARALLEL RESOURCE SHARING ALGORITHM with a smaller value of ε_2 and more phases. With smaller values of ε_2,

- the maximum congestion improves or remains approximately equal on all chips
- with the exception of the chip Gerhard (see below), the lower bound λ_{lb} on the maximum congestion in an optimum solution improves
- total overload is reduced or stays equal in most cases; where this is not the case, maximum congestion is reduced
- the objective value achieved, both in the fractional and in the final integral solution, improves by far on most chips; where this is not the case, it becomes only marginally worse

On the chip Gerhard, the lower bound λ_{lb} obtained with $\varepsilon_2 := 0.5$ and 200 phases performed in the PARALLEL RESOURCE SHARING ALGORITHM is only 0.443 compared to 0.93 with $\varepsilon_2 := 1.0$ and 100 phases. Of course the number of phases required

Chip	ε_2	phases	λ_{fract} (obj./edges)	λ_{lb}	λ_{rounded}	λ_{final}	TOL	% Gap (fract.)	% Gap (final)	Runtime
Lucius	0.5	200	0.963 (0.963/0.961)	0.946	1.500	1.000	0.0	4.36	4.42	0:00:58.2
	1.0	100	0.960 (0.960/0.949)	0.938	1.636	0.971	0.0	4.55	4.67	0:00:37.5
	5.0	20	0.961 (0.961/0.932)	0.915	1.532	1.000	0.0	5.77	5.85	0:00:11.7
Heijo	0.5	200	1.008 (0.950/1.008)	0.992	1.333	1.000	0.0	1.42	1.39	0:00:23.1
	1.0	100	1.007 (0.950/1.007)	0.809	1.333	1.000	0.0	1.46	1.41	0:00:14.5
	5.0	20	1.035 (0.950/1.035)	0.716	1.333	1.000	0.0	1.84	1.91	0:00:06.7
Ruediger	0.5	200	0.949 (0.949/0.944)	0.943	1.167	0.969	0.0	0.89	0.94	0:00:20.1
	1.0	100	0.948 (0.948/0.930)	0.942	1.138	0.969	0.0	0.96	1.00	0:00:12.9
	5.0	20	0.949 (0.949/0.946)	0.938	1.122	0.971	0.0	1.27	1.27	0:00:06.6
Timo	0.5	200	0.950 (0.950/0.950)	0.943	1.333	0.971	0.0	1.31	1.38	0:00:48.7
	1.0	100	0.950 (0.950/0.950)	0.942	1.222	0.971	0.0	1.34	1.39	0:00:29.8
	5.0	20	0.949 (0.949/0.939)	0.934	1.145	0.970	0.0	1.82	1.79	0:00:15.2
Jo	0.5	200	0.950 (0.950/0.950)	0.938	1.517	0.971	0.0	1.65	1.64	0:03:47.8
	1.0	100	0.950 (0.950/0.950)	0.937	1.517	0.971	0.0	1.67	1.69	0:02:38.6
	5.0	20	0.950 (0.950/0.948)	0.924	1.517	0.970	0.0	2.24	2.17	0:00:48.7
Renate	0.5	200	1.000 (0.949/1.000)	0.962	1.400	1.000	0.0	0.82	0.81	0:01:27.2
	1.0	100	1.000 (0.950/1.000)	0.961	1.333	1.000	0.0	0.86	0.85	0:00:56.5
	5.0	20	1.000 (0.934/1.000)	0.961	1.178	1.000	0.0	0.93	0.94	0:00:29.3
Rosemarie	0.5	200	0.950 (0.950/0.949)	0.941	2.000	1.000	0.0	1.91	1.96	0:01:45.5
	1.0	100	0.950 (0.950/0.947)	0.939	2.000	1.000	0.0	1.97	2.01	0:01:08.4
	5.0	20	0.949 (0.949/0.944)	0.930	2.000	1.000	0.0	2.41	2.43	0:00:28.4
Tuula	0.5	200	0.950 (0.950/0.950)	0.941	2.000	0.971	0.0	1.27	1.30	0:02:23.9
	1.0	100	0.950 (0.950/0.948)	0.940	2.000	1.000	0.0	1.26	1.27	0:01:21.9
	5.0	20	0.949 (0.949/0.944)	0.937	2.000	0.971	0.0	1.50	1.50	0:00:30.4
Georg	0.5	200	0.950 (0.950/0.949)	0.942	2.000	1.000	0.0	1.53	1.56	0:02:38.5
	1.0	100	0.950 (0.950/0.945)	0.940	3.000	1.000	0.0	1.61	1.63	0:01:40.2
	5.0	20	0.949 (0.949/0.941)	0.934	2.000	1.000	0.0	1.90	1.91	0:00:49.7
Sigurd	0.5	200	0.968 (0.968/0.967)	0.945	2.286	1.000	0.0	4.71	4.79	0:15:47.8
	1.0	100	0.964 (0.964/0.964)	0.936	2.545	1.000	0.0	4.95	4.99	0:12:45.1
	5.0	20	0.970 (0.970/0.968)	0.885	2.500	1.000	0.0	6.75	6.94	0:03:36.8
Wighart	0.5	200	0.950 (0.950/0.950)	0.939	3.000	1.000	0.0	1.92	2.00	0:05:48.7
	1.0	100	0.950 (0.950/0.948)	0.937	2.333	1.000	0.0	1.99	2.05	0:03:40.2
	5.0	20	0.949 (0.949/0.941)	0.929	2.000	1.000	0.0	2.39	2.43	0:01:32.7
Claus	0.5	200	0.972 (0.950/0.972)	0.932	3.000	1.030	0.1	1.81	1.96	0:12:36.9
	1.0	100	0.972 (0.950/0.972)	0.923	3.000	1.029	0.1	1.91	2.04	0:08:16.2
	5.0	20	0.977 (0.950/0.977)	0.844	2.222	1.034	1.0	2.83	2.87	0:03:04.9
Paul	0.5	200	9.476 (0.947/9.476)	4.465	20.286	9.091	771.6	1.45	1.30	0:34:21.3
	1.0	100	10.571 (0.932/10.571)	3.973	20.857	9.091	778.8	1.47	1.28	0:21:48.6
	5.0	20	20.143 (0.928/20.143)	3.524	21.857	9.429	745.5	1.80	1.57	0:05:24.7
Dorothea	0.5	200	0.951 (0.951/0.937)	0.935	3.000	1.000	0.0	3.11	3.27	0:24:02.4
	1.0	100	0.950 (0.950/0.946)	0.930	3.000	1.000	0.0	3.26	3.40	0:14:49.8
	5.0	20	0.950 (0.949/0.950)	0.913	3.000	1.000	0.0	3.84	3.94	0:04:48.6
Richard	0.5	200	3.000 (0.944/3.000)	1.205	3.000	3.000	49.5	1.27	1.31	0:58:04.5
	1.0	100	3.000 (0.922/3.000)	1.122	3.000	3.000	52.4	1.31	1.32	0:30:35.1
	5.0	20	3.000 (0.923/3.000)	0.379	3.000	3.000	70.8	1.66	1.62	0:11:30.7
Laci	0.5	200	15.000 (0.933/15.000)	2.706	15.000	15.000	211.1	1.90	1.93	0:59:35.5
	1.0	100	15.000 (0.928/15.000)	2.682	15.000	15.000	217.1	1.98	1.97	0:46:01.3
	5.0	20	15.000 (0.929/15.000)	2.581	15.000	15.000	220.0	2.27	2.23	0:12:00.4
Vasek	0.5	200	3.500 (0.955/3.500)	1.177	4.667	4.000	294.7	2.80	2.95	8:09:04.7
	1.0	100	3.500 (0.959/3.500)	1.106	5.000	4.000	338.1	2.96	2.99	4:23:15.3
	5.0	20	3.500 (0.948/3.500)	0.670	4.000	4.000	440.8	5.30	4.77	1:27:40.8
Camilla	0.5	200	0.950 (0.950/0.946)	0.937	4.000	1.000	0.0	2.52	2.56	0:51:09.6
	1.0	100	0.950 (0.950/0.950)	0.933	4.000	1.000	0.0	2.59	2.61	0:33:05.5
	5.0	20	0.950 (0.950/0.947)	0.918	3.000	1.000	0.0	3.34	3.33	0:11:23.6
Chiara	0.5	200	0.950 (0.950/0.949)	0.939	3.000	1.000	0.0	1.41	1.48	0:36:32.8
	1.0	100	0.950 (0.950/0.950)	0.938	3.500	1.000	0.0	1.47	1.52	0:22:12.9
	5.0	20	0.950 (0.950/0.948)	0.932	3.000	1.000	0.0	1.86	1.88	0:08:30.1
Tomoko	0.5	200	1.629 (0.950/1.629)	1.449	3.000	1.600	0.8	1.12	1.14	0:23:43.0
	1.0	100	1.625 (0.950/1.625)	1.449	3.000	1.600	0.8	1.18	1.20	0:15:21.2
	5.0	20	1.667 (0.949/1.667)	1.369	2.500	1.600	0.9	1.36	1.36	0:07:09.0
Andre	0.5	200	0.950 (0.950/0.948)	0.938	3.000	1.000	0.0	2.25	2.38	1:18:54.8
	1.0	100	0.950 (0.950/0.950)	0.935	3.000	1.000	0.0	2.38	2.48	0:50:43.3
	5.0	20	0.950 (0.950/0.949)	0.923	3.000	1.000	0.0	2.92	2.96	0:17:32.3
Ludwig	0.5	200	0.971 (0.957/0.971)	0.937	4.000	1.000	0.0	3.00	3.04	5:58:18.7
	1.0	100	0.971 (0.955/0.971)	0.930	4.000	1.000	0.0	3.14	3.17	4:24:53.3
	5.0	20	0.984 (0.957/0.984)	0.875	3.000	1.000	0.0	4.35	4.44	1:09:30.0

Table 4.5: Results of the BonnRoute® global router on the 65 nm chips in our testbed, optimizing wiring length and vias (8 threads)

Chip	ε_2	Phases	λ_{fract} (obj./edges)	λ_{lb}	λ_{rounded}	λ_{final}	TOL	% Gap	% Gap	Runtime
								(fract.)	(final)	
Raphaelo	0.5	200	0.949 (0.949/0.865)	0.944	1.200	0.971	0.0	2.78	2.74	0:00:21.5
	1.0	100	0.948 (0.948/0.938)	0.942	1.211	0.971	0.0	2.84	2.77	0:00:13.3
	5.0	20	0.947 (0.947/0.889)	0.926	1.233	0.971	0.0	3.43	3.39	0:00:06.4
Nelu	0.5	200	0.949 (0.949/0.935)	0.943	2.000	0.971	0.0	2.99	3.11	0:00:32.4
	1.0	100	0.949 (0.949/0.935)	0.940	2.000	0.971	0.0	3.10	3.22	0:00:20.1
	5.0	20	0.948 (0.948/0.910)	0.925	1.852	0.971	0.0	3.66	3.78	0:00:09.3
Gerhard	0.5	200	0.954 (0.954/0.946)	0.443	2.778	1.177	0.0	4.75	4.91	0:01:49.7
	1.0	100	0.972 (0.953/0.972)	0.930	3.000	1.293	0.3	4.82	4.95	0:01:18.1
	5.0	20	1.021 (0.960/1.021)	0.712	3.000	1.158	0.3	5.60	5.65	0:00:48.2
Victor	0.5	200	0.971 (0.971/0.971)	0.946	1.719	0.985	0.0	6.95	7.15	0:11:27.5
	1.0	100	0.973 (0.973/0.973)	0.942	1.637	1.004	0.0	7.21	7.27	0:08:51.0
	5.0	20	0.989 (0.989/0.987)	0.918	1.619	0.997	0.0	8.97	9.23	0:01:56.0
Henrik	0.5	200	1.646 (0.950/1.646)	1.409	3.000	1.630	29.0	2.33	2.58	0:03:46.3
	1.0	100	1.647 (0.949/1.647)	1.400	3.000	1.647	28.8	2.44	2.64	0:02:23.8
	5.0	20	1.651 (0.949/1.651)	1.372	3.000	1.647	28.7	3.37	3.46	0:00:57.9
Emilia	0.5	200	0.950 (0.950/0.946)	0.943	2.456	0.971	0.0	1.56	1.59	0:02:20.5
	1.0	100	0.950 (0.950/0.950)	0.942	2.000	0.971	0.0	1.55	1.55	0:01:27.7
	5.0	20	0.950 (0.950/0.945)	0.941	2.000	0.971	0.0	1.66	1.65	0:00:39.4
Milena	0.5	200	0.991 (0.966/0.991)	0.949	3.666	0.996	0.0	6.22	6.36	0:07:10.6
	1.0	100	0.991 (0.967/0.991)	0.944	3.000	1.004	0.0	6.36	6.48	0:04:40.9
	5.0	20	1.007 (0.984/1.007)	0.919	3.000	1.007	0.0	8.24	8.26	0:01:58.3
Simona	0.5	200	0.973 (0.973/0.972)	0.949	3.333	1.004	0.0	7.02	7.25	0:24:55.8
	1.0	100	0.975 (0.975/0.974)	0.942	3.333	1.000	0.0	7.22	7.43	0:18:30.6
	5.0	20	0.994 (0.994/0.987)	0.904	3.486	1.011	0.1	9.30	9.36	0:03:44.8
Angela	0.5	200	0.956 (0.956/0.955)	0.949	3.000	1.000	0.0	2.59	2.59	0:06:48.1
	1.0	100	0.953 (0.953/0.953)	0.944	3.000	1.000	0.0	2.58	2.57	0:04:20.9
	5.0	20	0.950 (0.950/0.946)	0.931	3.000	1.000	0.0	2.89	2.86	0:01:54.6
Guido	0.5	200	0.950 (0.950/0.947)	0.938	3.055	0.990	0.0	3.66	3.79	0:07:12.2
	1.0	100	0.950 (0.950/0.943)	0.935	3.055	1.000	0.0	3.74	3.84	0:04:33.0
	5.0	20	0.950 (0.950/0.927)	0.918	3.000	1.000	0.0	4.45	4.50	0:01:51.6
Dirk	0.5	200	1.893 (0.982/1.893)	1.102	4.122	1.872	3727.0	12.88	12.48	1:09:19.0
	1.0	100	1.894 (0.982/1.894)	1.069	4.036	1.872	4023.5	14.41	13.93	0:40:51.4
	5.0	20	1.967 (1.015/1.967)	0.838	6.778	1.872	6269.3	21.84	22.26	0:11:59.3
Thilo	0.5	200	1.402 (0.971/1.402)	1.088	3.000	1.397	786.9	7.53	7.33	0:24:14.3
	1.0	100	1.406 (0.974/1.406)	1.048	3.000	1.397	806.7	7.91	7.58	0:15:06.2
	5.0	20	1.478 (0.996/1.478)	0.817	3.027	1.414	1188.6	10.78	9.89	0:06:48.4
Martina	0.5	200	0.954 (0.954/0.953)	0.939	4.000	1.000	0.0	4.98	5.14	0:26:55.6
	1.0	100	0.955 (0.955/0.954)	0.935	4.000	1.000	0.0	5.09	5.20	0:17:57.7
	5.0	20	0.967 (0.967/0.959)	0.919	3.889	1.011	0.0	6.34	6.39	0:07:35.2

Table 4.6: Results of the BonnRoute® global router on the 45 nm chips in our testbed, optimizing wiring length and vias (8 threads)

by Theorem 3.6 increases quadratically in $1/\varepsilon$. With 400 phases instead of 200, λ_{lb} increased from 0.443 to 0.928.

On all chips for which none of the three parameter settings produced a feasible fractional or integral solution, λ_{lb} provided a certificate of infeasibility in the runs with $\varepsilon_2 = 0.5$ or $\varepsilon_2 = 1$. For most chips, a feasible integral solution is found if a feasible fractional solution has been found. Claus is the only 65nm chip for which this is not the case; on the 45nm testcases, this happens on five out of 13 chips, depending on the approximation parameter ε_2. It is not clear if this is due to the integrality gap or the simple iterative refinement heuristic. In all these cases however, the maximum congestion in the final integral solution is not much above 1, and total overload of this solution is small.

On the chip Heijo, the fractional solution is not feasible, but iterative refinement manages to find a feasible integral solution in the end. This does not contradict Theorem 3.6, and indeed the lower bound λ_{lb} does not prove that no feasible solution exists.

On many chips the maximum congestion in the fractional solution is very close to 0.95. This is due to the fact that the bound Γ^{obj} on objective costs is tightened only such that congestion in the objective function resource does not exceed 0.95 (see section 4.7.1). On the other hand, the maximum edge congestion tends to increase as long as it has not reached this value if this allows to reduce wire length and the number of vias.

Yield- and Power Optimization

Tables 4.7 and 4.8 show results on wiring length-, yield- and power optimization obtained with $\varepsilon_2 := 5.0$ and 20 phases, and with $\varepsilon_2 := 1.0$ and 100 phases, respectively, in each run of the PARALLEL RESOURCE SHARING ALGORITHM. The experiments were done using 8 Intel Xeon E7220 processors running at 2.93 GHz. For each chip there are three lines in the tables, one for each optimization objective. For *wiring length optimization*, as above vias are assigned a length which corresponds to 7-11 routing tracks (depending on the layer) by the standard parameters of BonnRoute®, and the bound Γ^{obj} on objective costs is set to $1.1\gamma_{\text{lb}}$ initially. For *yield optimization*, objective costs are weighted by factors obtained from manufacturing. 95 percent of vias are assumed to be realizable as so-called *redundant vias*, i.e. pairs of two neighboured vias, which drastically reduces contribution of vias to manufacturing yield loss. Often single vias are replaced by redundant vias in a post-processing step after routing, resulting in a lower percentage of redundant vias, but this figure can be raised significantly by making redundant vias default and switch to single vias only in case of routability problems. Also, *local loops* can be added to the wiring to increase via redundancy, see Bickford et al. [2006]. For yield optimization, we set $\Gamma^{\text{obj}} := 2.0\gamma_{\text{lb}}$ initially. For *power optimization*, we use extracted capacitances for closest possible spacing and one track of extra spacing. For simplicity, the extraction values of the standard wire type are used for all nets, although a small fraction of nets is routed with other wire types, and maximum allocated extra space is limited to one track. We set the bound $\Gamma^{\text{obj}} := 1.2\gamma_{\text{lb}}$ initially for power optimization.

As earlier, λ_{lb} is the lower bound obtained on the maximum congestion in an optimum solution, and runtimes include construction of initial Steiner trees, the PARALLEL RESOURCE SHARING ALGORITHM, randomized rounding and iterative refinement. Columns 4 to 7 refer to the final integral solution: Column 4 is the percentaged deviation of the achieved objective value from the lower bound γ_{lb}, and Columns 5 to 7 relate the results to each other by evaluating the global routing solution obtained in each line w.r.t. the objective costs of each of the three objectives and dividing the resulting value by the value achieved when actually optimizing this objective. The diagonal entries in each of the 3×3-matrices therefore are all 1. For a fair comparison, before evaluating a global routing solution $((T_N, s_N))_{N \in \mathcal{N}}$ w.r.t. one of the objectives, remaining unused capacity

$$\max\left\{0, u(e) - \sum_{N \in \mathcal{N}: e \in E(T_N)} (w(N,e) + s_N(e))\right\}$$

for each edge $e \in E(G)$ is distributed in discrete amounts equal to the wiring pitch on the layer of edge e in an optimal way among the nets $N \in \mathcal{N}$ with $e \in E(T_N)$. This obviously can be done in a greedy fashion on each edge.

The results clearly show that when optimizing one of the objectives, results improve considerably compared to a routing optimized for a different objective. This becomes apparent most distinctly in critical area, which reduces by up to 30 percent compared to a wiring length- or power-optimized routing. The reason for this is that both for wiring length and power optimization, using higher routing layers causes high via costs, but costs of wire segments are the same on all layers for length optimization, and almost the same for power optimization for wires with the same spacing. Hence optimizing a weighted sum of wiring length and number of vias, as is done in most classical routing benchmarks, can significantly degrade manufacturing yield. The difference between wiring length and power optimization is much smaller. This is due to the fact that power consumption is reduced mainly by assigning extra space to wires. In a routing optimized for wiring length, a large part of wires however runs through regions with low or medium edge congestion, so assigning extra space in a post-processing step as sketched above comes already close to the objective value achieved when actually optimizing power. Note that in the runs with $\varepsilon_2 = 5.0$, power consumption of the solution generated by wiring length optimization on the chips Sigurd and Claus is even better than the result obtained by actually optimizing power. In table 4.8 (with $\varepsilon_2 = 1.0$) this does not happen any more.

Runtimes for yield and power optimization are considerably higher than for wiring length minimization. This is due to the weaker lower bounds which not only increase the runtime of a single block solver call, but also cause a higher number of block solver calls, as lower bounds on the achievable objective costs for a net are used also in the recomputation criterion in the RESOURCE SHARING ALGORITHM (see chapter 3). Nonetheless, critical area optimization takes less than an hour on most of our testcases, and only a few hours on the largest chips (depending on the approximation parame-

Chip	Optimization objective	λ_{lb}	% Gap	Relative obj. value			Runtime
				wiring length	critical area	power	
Lucius	wiring length	0.914	5.75	1.000	1.115	1.007	0:00:15.2
	critical area	0.860	20.72	1.060	1.000	1.067	0:00:53.8
	power	0.881	15.92	1.050	1.196	1.000	0:00:36.6
Heijo	wiring length	0.364	1.84	1.000	1.163	1.014	0:00:07.3
	critical area	0.797	6.21	1.032	1.000	1.072	0:00:23.5
	power	0.532	6.20	1.049	1.302	1.000	0:00:19.9
Ruediger	wiring length	0.939	1.27	1.000	1.186	1.012	0:00:07.1
	critical area	0.898	4.48	1.034	1.000	1.058	0:00:20.9
	power	0.913	3.98	1.047	1.327	1.000	0:00:17.6
Timo	wiring length	0.934	1.82	1.000	1.132	1.017	0:00:16.1
	critical area	0.899	4.40	1.034	1.000	1.068	0:00:34.5
	power	0.913	5.19	1.055	1.286	1.000	0:00:35.3
Jo	wiring length	0.924	2.21	1.000	1.145	1.013	0:00:53.0
	critical area	0.855	13.12	1.028	1.000	1.058	0:04:38.0
	power	0.892	8.56	1.051	1.291	1.000	0:02:59.6
Renate	wiring length	0.961	0.93	1.000	1.198	1.005	0:00:31.5
	critical area	0.940	3.33	1.027	1.000	1.065	0:01:35.5
	power	0.949	3.61	1.036	1.353	1.000	0:01:07.3
Rosemarie	wiring length	0.930	2.43	1.000	1.216	1.010	0:00:31.9
	critical area	0.777	34.18	1.054	1.000	1.070	0:05:24.3
	power	0.896	9.23	1.058	1.286	1.000	0:02:59.0
Tuula	wiring length	0.937	1.50	1.000	1.246	1.005	0:00:31.8
	critical area	0.866	10.15	1.039	1.000	1.086	0:04:31.0
	power	0.903	7.52	1.037	1.389	1.000	0:02:27.3
Georg	wiring length	0.934	1.91	1.000	1.302	1.011	0:00:53.7
	critical area	0.794	23.79	1.061	1.000	1.088	0:05:45.9
	power	0.909	6.04	1.057	1.424	1.000	0:02:53.3
Sigurd	wiring length	0.884	6.92	1.000	1.122	0.980	0:04:15.6
	critical area	0.701	60.03	1.053	1.000	1.024	0:19:35.5
	power	0.779	21.64	1.083	1.218	1.000	0:11:49.3
Wighart	wiring length	0.929	2.42	1.000	1.274	1.006	0:01:42.9
	critical area	0.762	31.81	1.073	1.000	1.082	0:15:25.2
	power	0.450	8.42	1.053	1.344	1.000	0:07:55.3
Claus	wiring length	0.850	2.85	1.000	1.192	0.997	0:03:38.9
	critical area	0.671	50.52	1.050	1.000	1.067	0:28:58.6
	power	0.733	10.02	1.049	1.269	1.000	0:26:46.5
Dorothea	wiring length	0.913	3.95	1.000	1.281	1.004	0:05:29.9
	critical area	0.754	37.68	1.064	1.000	1.064	0:43:26.7
	power	0.870	11.08	1.050	1.396	1.000	1:07:15.6
Camilla	wiring length	0.918	3.33	1.000	1.255	1.011	0:13:57.4
	critical area	0.719	42.38	1.055	1.000	1.073	1:25:15.9
	power	0.883	10.75	1.047	1.338	1.000	0:58:13.6
Chiara	wiring length	0.932	1.88	1.000	1.284	1.002	0:08:27.7
	critical area	0.778	41.17	1.061	1.000	1.079	2:03:34.0
	power	0.897	8.48	1.042	1.360	1.000	1:30:20.3
Tomoko	wiring length	1.369	1.36	1.000	1.387	1.009	0:07:40.4
	critical area	1.220	26.74	1.061	1.000	1.086	1:18:53.0
	power	1.373	5.23	1.043	1.496	1.000	0:38:21.0
Andre	wiring length	0.923	2.96	1.000	1.255	1.004	0:17:36.3
	critical area	0.746	34.80	1.064	1.000	1.070	3:04:24.2
	power	0.893	8.47	1.048	1.332	1.000	2:10:58.5
Ludwig	wiring length	0.875	4.44	1.000	1.304	1.007	1:21:47.3
	critical area	0.679	51.76	1.058	1.000	1.081	6:15:17.6
	power	0.799	12.87	1.049	1.399	1.000	3:04:10.6

Table 4.7: Relative objective values obtained for wiring length, yield and power optimization with $\varepsilon_2 := 5.0$ and 20 phases performed in the PARALLEL RESOURCE SHARING ALGORITHM. Runtimes are for the complete global routing runs, using 8 threads.

Chip	Optimization objective	λ_{lb}	% Gap	Relative obj. value			Runtime
				wiring length	critical area	power	
Lucius	wiring length	0.938	4.65	1.000	1.106	1.013	0:00:50.5
	critical area	0.904	20.19	1.083	1.000	1.097	0:03:46.5
	power	0.919	14.07	1.030	1.161	1.000	0:02:00.1
Heijo	wiring length	0.809	1.47	1.000	1.196	1.027	0:00:16.6
	critical area	0.925	5.31	1.035	1.000	1.078	0:01:23.3
	power	0.927	5.75	1.043	1.299	1.000	0:01:06.2
Ruediger	wiring length	0.942	0.98	1.000	1.200	1.025	0:00:13.1
	critical area	0.912	3.42	1.037	1.000	1.066	0:01:01.8
	power	0.917	3.51	1.041	1.340	1.000	0:00:58.7
Timo	wiring length	0.942	1.36	1.000	1.161	1.031	0:00:32.4
	critical area	0.916	3.33	1.039	1.000	1.076	0:01:37.0
	power	0.923	4.66	1.049	1.288	1.000	0:01:44.8
Jo	wiring length	0.937	1.67	1.000	1.166	1.027	0:03:00.4
	critical area	0.897	12.51	1.033	1.000	1.065	0:17:45.9
	power	0.921	7.75	1.036	1.268	1.000	0:08:33.8
Renate	wiring length	0.961	0.86	1.000	1.244	1.017	0:00:50.6
	critical area	0.944	2.66	1.032	1.000	1.074	0:04:26.5
	power	0.950	3.16	1.031	1.363	1.000	0:03:07.9
Rosemarie	wiring length	0.939	2.01	1.000	1.274	1.030	0:01:19.1
	critical area	0.877	31.73	1.050	1.000	1.074	0:19:13.9
	power	0.919	8.41	1.043	1.281	1.000	0:07:43.8
Tuula	wiring length	0.940	1.28	1.000	1.290	1.020	0:01:35.7
	critical area	0.902	8.83	1.041	1.000	1.096	0:18:34.2
	power	0.921	6.47	1.024	1.393	1.000	0:07:22.9
Georg	wiring length	0.940	1.62	1.000	1.354	1.027	0:01:45.3
	critical area	0.882	21.57	1.055	1.000	1.092	0:22:22.8
	power	0.919	5.32	1.047	1.432	1.000	0:07:56.7
Sigurd	wiring length	0.936	4.99	1.000	1.117	1.012	0:14:53.6
	critical area	0.853	57.15	1.061	1.000	1.064	0:58:33.6
	power	0.909	15.84	1.032	1.149	1.000	0:34:07.7
Wighart	wiring length	0.937	2.04	1.000	1.320	1.021	0:04:03.2
	critical area	0.869	28.63	1.068	1.000	1.091	1:00:47.9
	power	0.920	7.25	1.039	1.360	1.000	0:17:58.5
Claus	wiring length	0.925	2.05	1.000	1.228	1.016	0:10:05.1
	critical area	0.846	45.34	1.036	1.000	1.070	1:40:35.5
	power	0.907	7.75	1.028	1.283	1.000	0:47:53.7
Dorothea	wiring length	0.930	3.40	1.000	1.338	1.019	0:17:28.8
	critical area	0.866	34.18	1.062	1.000	1.079	3:21:38.0
	power	0.913	9.15	1.027	1.398	1.000	1:41:38.2
Camilla	wiring length	0.933	2.60	1.000	1.288	1.023	0:38:38.3
	critical area	0.852	40.10	1.053	1.000	1.079	5:13:32.1
	power	0.918	9.66	1.032	1.325	1.000	2:14:00.2
Chiara	wiring length	0.937	1.52	1.000	1.332	1.018	0:26:04.9
	critical area	0.878	37.88	1.054	1.000	1.084	8:18:08.2
	power	0.920	7.13	1.027	1.370	1.000	2:30:23.2
Tomoko	wiring length	1.449	1.20	1.000	1.434	1.026	0:16:55.1
	critical area	1.342	24.35	1.054	1.000	1.091	5:01:36.5
	power	1.449	4.46	1.033	1.513	1.000	1:27:42.2
Andre	wiring length	0.935	2.47	1.000	1.282	1.018	0:58:14.6
	critical area	0.861	31.57	1.060	1.000	1.083	12:22:40.8
	power	0.917	6.86	1.029	1.337	1.000	4:12:02.3
Ludwig	wiring length	0.930	3.17	1.000	1.316	1.021	4:58:45.7
	critical area	0.841	50.12	1.060	1.000	1.092	20:15:24.1
	power	0.908	10.42	1.028	1.368	1.000	8:52:50.8

Table 4.8: Relative objective values obtained for wiring length, yield and power optimization with $\varepsilon_2 := 1.0$ and 100 phases performed in the PARALLEL RESOURCE SHARING ALGORITHM. Runtimes are for the complete global routing runs, using 8 threads.

ter), making it very valuable in practice because a 30 percent reduction in critical area increases manufacturing yield correspondingly.

4.10 Discussion and Outlook

In this chapter, we showed how a fractional relaxation of the GLOBAL ROUTING PROBLEM in VLSI design can be formulated as a RESOURCE SHARING PROBLEM and efficiently solved by the PARALLEL RESOURCE SHARING ALGORITHM presented in chapter 3. A near-optimum integral solution can be obtained fast in practice from the fractional solution by randomized rounding and an iterative refinement procedure.

We showed experimental results which demonstrate that the algorithm can be used to optimize not only wiring length very efficiently, but also the practically relevant objectives, manufacturing yield and power consumption. Moreover, the algorithm scales very well with the number of processors used. This makes it possible to optimize even the largest chips in a few minutes to a few hours, depending on the optimization objective. Critical area in routing can be reduced by up to 30 percent, and manufacturing yield increased correspondingly.

Nonetheless, runtime can be reduced further when lower bounds (for guiding path search or for deciding which nets to reroute in a phase of the RESOURCE SHARING ALGORITHM) are weak. This is the case in yield optimization because of the large differences in layer characteristics, in power optimization because of the significant influence of spacing, and in general in presence of high congestion. To drastically reduce runtime spent on routing very long connections, a fast tree enumeration approach as proposed in section 4.8 is likely to help to achieve near-optimum solutions (or indicate infeasibility of the global routing instance) much faster. Although a strict *proof* of infeasibility requires the call to an exact block solver at least once for each net on the resource prices obtained from the RESOURCE SHARING ALGORITHM and applying the weak duality Lemma 3.2, using such heuristic block solvers in most cases — if an instance is not on the edge of infeasibility — should suffice in practice to distinct between feasible and infeasible instances. Ideally, a global router based on the RESOURCE SHARING ALGORITHM with very fast block solvers could be used for congestion estimation in placement, replacing probabilistic approaches (see e.g. Shelar and Saxena [2009]).

A further challenge is imposed by the continued shrinking of feature sizes. Electrical characteristics of wires scale significantly worse than transistors across successive process generations, and thus the relative contribution of wiring delay to overall latch-to-latch delay continuously grows in each new technology generation. Because wiring delay is roughly quadratic in electrical capacitance of the wiring, bridging long distances with the shortest possible signal delay requires insertion of *repeater circuits*, i.e. buffers or inverters, to refresh signals. Saxena et al. [2004] investigate the consequences of continuously shrinking feature sizes and predict a dramatic increase of *buffering* needed in forthcoming technologies. Hence there is a growing need of doing buffering during global routing in order not to cut down flexibility by a fixed placement of repeaters, thus coupling routing, timing optimization and placement more closely with each other. As buffering space is a limited resource, it is natural to understand buffers as parts of global routing wires and to incorporate them into the RESOURCE SHARING PROBLEM formulation of the global routing problem. The RESOURCE SHARING ALGORITHM does not have to be changed to accomodate this, but it is an important challenge to write a block solver which in short time finds a near-optimum buffered routing for a global routing net w.r.t. given prices for delay, buffering space and edge capacity consumption.

Finally, integration of global routing and timing optimization must be investigated further. Imposing constraints on total delay of critical paths as proposed in this chapter should improve timing after routing considerably, but this is only a first step. A more accurate computation of delay which takes resistance into account is important for timing critical nets, and delay-optimal solutions for a single net require it to be routed with a certain topology. In particular, block solvers cannot rely on the assumption that the contribution of using some edge in the global routing graph to signal delay is independent of the routing topology. An important task is to develop a block solver which finds a (near-)optimum routing for a single net w.r.t. a combination of topology-aware delay costs and costs for routing space consumption. Recomputing timing slacks during global routing might give more accurate results, and integration with other timing optimization steps, such as gate sizing, can increase the potential for optimization.

Bibliography

Adel'son-Vel'skiĭ, G. M., and Landis, E. M. [1962]: An Algorithm for the Organization of Information. *Soviet Mathematics Doklady, Vol. 3 (1962)*, pp. 1259–1262.

Adir, A., Attiya, H., and Shurek, G. [2003]: Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 5*, pp. 502–515.

Albrecht, C. [2001]: Global Routing by New Approximation Algorithms for Multicommodity Flow. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 20, no. 5 (2001)*, pp. 622–632.

Albrecht, C. [2001]: Zwei kombinatorische Optimierungsprobleme im VLSI-Design: Optimierung der Zykluszeit und der Slackverteilung und globale Verdrahtung. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 2001.

Alglave, J., Fox, A., Ishtiaq, S., Myreen, M. O., Sarkar, S., Sewell, P., and Nardelli, F. Z. [2009]: The Semantics of Power and ARM Multiprocessor Machine Code. *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming (DAMP 2009)*, pp. 13–24.

Alpert, C. J., Mehta, D. P., and Sapatnekar, S. S. (Eds.) [2009]: Handbook of Algorithms for VLSI Physical Design Automation. Taylor and Francis, Boca Raton 2009.

AMD [2007]: AMD64 Architecture Programmer's Manual (rev. 3.14). Advanced Micro Devices, Sept. 2007. Available online from `http://www.amd.com`.

Brenner, U., and Rohe, A. [2002]: An Effective Congestion Driven Placement Framework. *Proceedings of the ACM International Symposium on Physical Design (ISPD 2002)*, pp. 6–11.

Brenner, U., Struzyna, M., and Vygen, J. [2008]: BonnPlace: Placement of Leading-Edge Chips by Advanced Combinatorial Algorithms. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 27 (2008)*, pp. 1607–1620.

Burstein, M., and Pelavin, R. [1983]: Hierarchical Wire Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 2 (1983)*, pp. 223–234.

C++ Standards Committee, Pete Becker, ed. [2009]: Working Draft, Standard for Programming Language C++. C++ Standards Committee paper WG21/N2857=09-0047, available online from `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2857.pdf`.

Bienstock, D., and Iyengar, G. [2006]: Approximating Fractional Packings and Coverings in $O(1/\varepsilon)$ Iterations. *SIAM Journal on Computing, Vol. 35, No. 4 (2006)*, pp. 825–854.

Bickford, J., Bühler, M., Hibbeler, J., Koehl, J., Müller, D., Peyer, S., and Schulte, C. [2006]: Yield Improvement by Local Wiring Redundancy. In *Proceedings of the 7th International Symposium on Quality Electronic Design (2006)*, pp. 473–478.

Boehm, H. J. [2005]: Threads Cannot Be Implemented As a Library. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, pp. 261–268.

Boehm, H. J., and Adve, A. V. [2009]: Foundations of the C++ Concurrency Memory Model. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pp. 68–78.

Carden IV, R.C., Li, J., and Cheng, C.-K. [1996]: A Global Router with a Theoretical Bound on the Optimum Solution. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15 (1996), pp. 208–216.

Chang, Y.-J., Lee, Y.-T., and Wang, T.-C. [2008]: NTHU-Route 2.0: A Fast and Stable Global Router. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'08)*, pp. 338–343.

Charikar, M., Chekuri, C., Goel, A., Guha, S., and Plotkin, S. [1998]: Approximating a Finite Metric by a Small Number of Tree Metrics. In *Proceedings of the 39th Annual IEEE Symposium on the Foundations of Computer Science (1998)*, pp. 379–388.

Chen, H., Yao, B., Zhou, F., and Cheng, C. K. [2003]: The Y-Architecture: Yet Another On-Chip Interconnect Solution. *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 840–846, 2003.

Chen, H., Cheng, C.-K., Kahng, A., Mandoiu, I. .I., Wang, Q., and Yao, B. [2003]: The Y-Architecture for On-Chip Interconnect: Analysis and Methodology. *Proceedings of the 40th Conference on Design Automation (DAC'03)*, pp. 13–19, 2003.

Chernoff, H. [1952]: A Measure of Asymptotic Efficiency for Tests based on the Sum of Observations. *Annals of Mathematical Statistics 23 (1952)*, pp. 493–509.

Chiang, C., and Sarrafzadeh, M. [1991]: Wirability of Knock-Knee Layouts with 45°Wires. *IEEE Transactions on Circuits and Systems* 38, pp. 613–624, 1991.

Cho, M., Xiang, H., Puri, R., and Pan, D. Z. [2007]: TROY: Track Router with Yield-driven Wire Planning. *Proceedings of the 44th Conference on Design Automation (DAC'07)*, pp. 55–58.

Cho, M., Lu, K., Yuan, K., and Pan, D. Z. [2009]: BoxRouter 2.0: A Hybrid and Robust Global Router with Layer Assignment for Routability. *ACM Transactions on Design Automation of Electronic Systems, vol. 14, no. 2*, article 32.

Cho, M., Mitra, J., and Pan, D. Z. [2009]: Manufacturability Aware Routing. *In: Handbook of Algorithms for VLSI Physical Automation (C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds.)*, CRC Press, 2009.

Dai, V., Yang, J., Rodriguez, N., and Capodieci, L. [2009]: Developing DRC plus Rules through 2D Pattern Extraction and Clustering Techniques. *Proceedings of SPIE, Vol. 7275 (2009).*

Dijkstra, E. W. [1959]: A Note on two Problems in Connexion with Graphs. *Numerische Mathematik 1 (1959)*, pp. 269–271.

Erdős, P., and Lovász, L.: Problems and Results on 3-Chromatic Hypergraphs and some Related Questions. *In: Infinite and Finite Sets (A. Hajnal, R. Rado, V. T. Sos, eds.), North-Holland, Amsterdam (1975)*, pp. 609–628.

Fraser, K. and Harris, T. [2007]: Concurrent Programming Without Locks. *ACM Transactions on Computer Systems, vol. 25, issue 2 (2007)*, pp. 1–61.

Garg, N., and Könemann, J. [2008]: Faster and Simpler Algorithms for Multicommodity Flow and other Fractional Packing Problems. *SIAM Journal on Computing, Vol. 37, No. 2 (2007)*, pp. 630–652.

Goldberg, A. V., and Harrelson, C. [2005]: Computing the Shortest Path: A^* Search Meets Graph Theory. *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2005)*, pp. 156–165.

Grigoriadis, M. D., and Khachiyan, L. D. [1994]: Fast Approximation Schemes for Convex Programs with Many Blocks and Coupling Constraints. *SIAM Journal on Optimization, Vol. 4, No. 1 (1994)*, pp. 86–107.

Grigoriadis, M. D., and Khachiyan, L. D. [1996]: Coordination Complexity of Parallel Price-Directive Decomposition. *Mathematics of Operations Research, Vol. 21, No. 2 (1996)*, pp. 321–340.

Grötschel, M., Martin, A., and Weismantel, R. [1996b]: Packing Steiner Trees: A Cutting Plane Algorithm and Computational Results. Mathematical Programming 72 (1996), pp. 125–145.

Heiss, R. [1968]: A Path Connection Algorithm for Multi-Layer Boards. *Proceedings of the 5th Conference on Design Automation (DAC'68)*, pp. 6.1–6.14, 1968.

Held, S. [2008]: Timing Closure in Chip Design. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 2008.

Hennessy, J. L., and Patterson, D. A. [2006]: Computer Architecture: A Quantitative Approach (4th Edition). ISBN 0123704901, Morgan Kaufmann.

Herlihy, M. [1990]: A Methodology for Implementing Highly Concurrent Data Structures. *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 90)*, pp. 197–206.

Herlihy, M. [1991]: Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 13, issue 1*, pp. 124–149.

Herlihy, M., and Moss, J. E. [1993]: Transactional Memory: Architectural Support for Lock-Free Data Structures *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA 93)*, pp. 289–300.

Hetzel, A. [1995]: Verdrahtung im VLSI-Design: Spezielle Teilprobleme und ein sequentielles Lösungsverfahren. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 1995.

Hetzel, A. [1998]: A sequential detailed router for huge grid graphs. In *Proceedings of Design, Automation and Test in Europe (DATE 1998)*, pp. 332–339.

Ho, T.-Y., Chang, C.-F., Cheang, Y.-W. and Chen, S.-J. [2005]: Multilevel Full-chip Routing for the X-based Architecture *Proceedings of the 42nd Conference on Design Automation (DAC'05)*, pp. 597–602, 2005.

Hong, X., Xue, T., Huang, J., Cheng, C.-K., and Kuh, E. S. [1997]: TIGER: An Efficient Timing-Driven Global Router for Gate Array and Standard Cell Layout Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, no. 11*, pp. 1323–1331.

Hu, J., and Sapatnekar, S.S. [2001]: A Survey on Multi-Net Global Routing for Intergated Circuits. *Integration, the VLSI Journal 31 (2001)*, pp. 1–49.

Hu, J., Robins, G., and Sze, C. C. N. [2009]: Timing-Driven Interconnect Synthesis. *In: Handbook of Algorithms for VLSI Physical Automation (C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds.)*, CRC Press, 2009.

Humpola, J. [2009]: Schneller Algorithmus für kürzeste Wege in irregulären Gittergraphen. Diploma thesis, Research Institute for Discrete Mathematics, University of Bonn, 2009.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, IEEE, 1985.

Intel [2009]: Intel®64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 031, order number 253668-031US, June 2009. Available online from `http://www.intel.com`.

ISPD Global Routing Contest [2007]: `http://www.sigda.org/ispd2007/contest.html`.

ISPD Global Routing Contest [2008]: `http://www.sigda.org/ispd2008/contests/ispd08rc.html`.

Jansen, K., and Zhang, H. [2008]: Approximation Algorithms for General Packing Problems and their Application to the Multicast Congestion Problem. *Mathematical Programming 114, Ser. A (2008)*, pp. 183–206.

Khandekar, R. [2004]: Lagrangian Relaxation based Algorithms for Convex Programming Problems. PhD thesis, Indian Institute of Technology Delhi, 2004.

Klein, P., and Young, N. E. [1999]: On the Number of Iterations for Dantzig-Wolfe Optimization and Packing-Covering Approximation Algorithms. *Integer Programming and Combinatorial Optimization; Proceedings of the 7th International IPCO Conference; LNCS 1610 (G. Cornuéjols, R. E. Burkard, G. J. Woeginger, eds.)*, pp. 320–327, Springer, Berlin 1999.

Koren, I., and Koren, Z. [1998]: Defect Tolerance in VLSI Circuits: Techniques and Yield Analysis. *Proceedings of the IEEE 86 (1998)*, pp. 1819–1837.

Korte, B., Rautenbach, D. and Vygen, J. [2007]: BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip. *Proceedings of the IEEE 95 (2007)*, pp. 555–572.

Korte, B., and Vygen, J. [2008]: Combinatorial Optimization: Theory and Algorithms. Fourth edition. Springer, Berlin 2008.

LEF/DEF Language Reference Version 5.7, available online from `http://www.si2.org`, 2007.

Lamport, L. [1979]: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers, vol. C-28, no. 9 (1979)*, pp. 690–691.

Lodi, E. [1988]: Routing Multiterminal Nets in a Diagonal Model. *Proceedings of the 1988 Conference on Information Sciences and Systems*, pp. 899–902, 1988.

Lee, T.-H., and Wang, T.-C. [2008]: Congestion-Constrained Layer Assignment for Via Minimization in Global Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 9*, pp. 1543–1656.

Lee, T.-H., and Wang, T.-C. [2009]: Robust Layer Assignment for Via Optimization in Multi-layer Global Routing. *Proceedings of the 2009 international symposium on Physical design (ISPD 09)*, pp. 159–166.

Leighton, T., Lu, C.-J., Rao, S., and Srinivasan, A. [2001]: New Algorithmic Aspects of the Local Lemma with Applications to Routing and Partitioning. *SIAM Journal on Computing 31 (2001)*, pp. 626–641.

Maßberg, J. [2009]: Facility Location and Clocktree Synthesis. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 2009.

Menge, A. [2008]: Weiterentwicklung der probabilistischen Congestion-Abschätzung. Diploma thesis, Research Institute for Discrete Mathematics, University of Bonn, 2008.

Michael, M. M. [2004]: Scalable Lock-Free Dynamic Memory Allocation. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 2004)*, pp. 35–46.

Moffitt, M. D. [2008]: MaizeRouter: Engineering an Effective Global Router. *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASP-DAC 08)*, pp. 226–231.

Moffitt, M. D., Roy, J. A., and Markov, I. L. [2008]: The Coming of Age of (Academic) Global Routing. *Proceedings of the 2008 International Symposium on Physical Design (ISPD 08)*, pp. 148–155.

Müller, D. [2002]: Bestimmung der Verdrahtungskapazitäten im Global Routing von VLSI-Chips. Diploma thesis, Research Institute for Discrete Mathematics, University of Bonn, 2002.

Müller, D. [2006a]: Optimizing Yield in Global Routing. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD'06)*, pp. 480–486.

Müller, D. [2006b]: On the Complexity of the Planar Directed Edge-Disjoint Paths Problem. *Mathematical Programming 105(2-3), Ser. B (2006)*, pp. 275–288.

Müller, D., and Vygen, J. [2008]: Faster min-max Resource Sharing and Applications. Technical Report No. 08987, Research Institute for Discrete Mathematics, University of Bonn, 2008.

Natarajan, S., Sherwani, N., Holmes, N. D., and Sarrafzadeh, M. [1992]: Over-the-Cell Channel Routing For High Performance Circuits. *Proceedings of the 29th Conference on Design Automation (DAC'92)*, pp. 600–603, 1992.

Naves, G. [2009]: The Hardness of Routing two Pairs on one Face. Available online from `http://hal.archives-ouvertes.fr/hal-00313944/fr`, 2008.

Nichols, B., Buttlar, D., and Proulx Farrell, J. [1996]: Pthreads Programming. ISBN 1-56592-125-1, O'reilly, Bonn, Cambridge, Paris, 1996.

Orlin, J.B. [1993]: A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Operations Research 41 (1993)*, pp. 338–350.

Owens, S., Sarkar, S., and Sewell, P. [2009]: A Better x86 Memory Model: x86-TSO. *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLS 2009)*, to appear.

Ozdal, M. M., and Wong, M. D. F. [2009]: Archer: A History-Based Global Routing Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 4*, pp. 528–540.

Pan, D. Z., Yu, P., Cho, M., Ramalingam, A., Kim, K., Rajaram, A., and Shi, S. X. [2008]: Design for Manufacturing Meets Advanced Process Control: A Survey. *Journal of Process Control, Volume 18, Issue 10*, pp. 975–984, 2008.

Panten, C. [2005]: Paralleles Verdrahten von VLSI-Chips auf Shared-Memory-Basis. Diploma thesis, Research Institute for Discrete Mathematics, University of Bonn, 2005.

Peyer, S., Rautenbach, D., and Vygen, J. [2006]: A Generalization of Dijkstra's Shortest Path Algorithm with Applications to VLSI Routing. Report No. 06964, Research Institute for Discrete Mathematics, University of Bonn, 2006. To appear in Journal of Discrete Algorithms.

Peyer, S. [2007]: Shortest Paths and Steiner Trees in VLSI Routing. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 2007.

Plotkin, S. A., Shmoys, D. B., and Tardos, É. [1995]: Fast Approximation Algorithms for Fractional Packing and Covering Problems. *Mathematics of Operations Research 2 (1995)*, pp. 257–301

Power ISA Version 2.06, January 2009. Available online from `http://www.power.org/resources/reading`.

Preparata, F. P. and Shamos, M. I. [1988]: Computational Geometry. Springer, 1988.

Raghavan, P. [1988]: Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs. *Journal of Computer and System Sciences 37 (1988)*, pp. 130–143.

Raghavan, P., and Thompson, C. D. [1987]: Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs. *Combinatorica 7 (1987)*, pp. 365–374.

Raghavan, P., and Thompson, C. D. [1991]: Multiterminal Global Routing: A Deterministic Approximation. *Algorithmica 6 (1991)*, pp. 73–82.

Robins, G., and Zelikovsky, A. [2009]: Minimum Steiner Tree Construction. *In: Handbook of Algorithms for VLSI Physical Automation (C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds.)*, CRC Press, 2009.

Rohe, A. [2001]: Sequential and Parallel Algorithms for Local Routing. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 2001.

Roy, J. A. , and Markov, I. L. [2008]: High-Performance Routing at the Nanometer Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, issue 6*, pp. 1066–1077.

Salowe, J. S. [2009]: Rip-Up and Reroute. *In: Handbook of Algorithms for VLSI Physical Automation (C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds.)*, CRC Press, 2009.

Sarkar, S., Sewell, P., Nardelli, F. Z., Owens, S., Ridge, T., Braibant, T., Myreen, M. O., and Alglave, J. [2009]: The Semantics of x86-CC Multiprocessor Machine Code. *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pp. 379–391.

Saxena, P., Menezes, N., Cocchini, P., and Kirkpatrick, D. A. [2004]: Repeater Scaling and Its Impact on CAD. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 4*, pp. 451–463.

Saxena, P., Shelar, R. S. and Sapatnekar, S. S. [2007]: Routing Congestion in VLSI Circuits: Estimation and Optimization. Springer, New York, 2007.

Schulte, C. [2006]: Yield-Optimierung im Detailed Routing. Diploma thesis, Research Institute for Discrete Mathematics, University of Bonn, 2006.

Sebő, A., and Naves, G. [2008]: Multiflow Feasibility: An Annotated Tableau. *Research Trends in Combinatorial Optimization (W. J. Cook, L. Lovász, J. Vygen, eds.)*, pp. 261–283, Springer, Berlin 2008.

Shahrokhi, F., and Matula, D. W. [1990]: The Maximum Concurrent Flow Problem. Journal of the ACM 37 (1990), pp. 318–334.

Shelar, R. S., and Saxena, P. [2009]: Estimation of Routing Congestion. *In: Handbook of Algorithms for VLSI Physical Automation (C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds.)*, CRC Press, 2009.

Shragowitz, E., and Keel, S. [1987]: A Global Router Based on a Multicommodity Flow Model. *Integration, the VLSI Journal, vol. 5, no. 1 (1987)*, pp. 3–16.

SPARC [2000]: The SPARC Architecture Manual, version 9. SPARC International, Inc., 2000. Available online from `http://developers.sun.com`.

Stroustrup, B. [2000]: The C++ Programming Language. ISBN 0201700735, Addison-Wesley, 2000.

Sundell, H. [2004]: Efficient and Practical Non-Blocking Data Structures. PhD thesis, technical report 30D, ISSN 1651-4971, ISBN 91-7291-514-5, Department of Computing Science, Chalmers University of Technology and Göteborg University.

Sundell, H., and Tsigas, P. [2005]: Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS 2004); LNCS 3544*, pp. 240–255, Springer, Berlin 2005.

Teig, S. [2002]: The X Architecture: Not Your Father's Diagonal Wiring. *Proceedings of the 2002 international workshop on system-level interconnect prediction*, pp. 33–37, 2002.

Teig, S., Hetzel, A., Ganley, J., Frankle, J., and Fujimura, A. [2009]: X Architecture Place and Route: Physical Design for the X Interconnect Architecture. *In: Handbook of Algorithms for VLSI Physical Automation (C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds.)*, CRC Press, 2009.

Ting, B. N., and Tien, B. N. [1983]: Routing Techniques for Gate Array. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 2, no. 4 (1983)*, pp. 301–312.

Valois, J. D. [1995]: Lock-free Linked Lists using Compare-And-Swap. *Proceedings of the 14th annual ACM symposium on Principles of Distributed Computing (PODC 95)*, pp. 214–222.

Vygen, J. [2001]: Theory of VLSI Layout. Habilitation thesis, Research Institute for Discrete Mathematics, University of Bonn, 2001.

Vygen, J. [2004]: Near-Optimum Global Routing with Coupling, Delay Bounds, and Power Consumption. *Integer Programming and Combinatorial Optimization; Proceedings of the 10th International IPCO Conference; LNCS 3064 (G. Nemhauser, D. Bienstock, eds.)*, pp. 308–324, Springer, Berlin 2004.

Werber, J. [2007]: Logic Restructuring for Timing Optimization in VLSI Design. PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, 2007.

Wu, T.-H., Davoodi, A., and Linderoth, J. T. [2009]: GRIP: Scalable 3D Global Routing Using Integer Programming. *Proceedings of the 46th Conference on Design Automation (DAC'09)*, 2009.

Xu, Y., Zhang, Y., and Chu, C. [2009]: FastRoute 4.0: Global Router with Efficient Via Minimization. *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC'09)*, pp. 576–581.

Young, N. E. [1995]: Randomized Rounding Without Solving the Linear Program. *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms (1995)*, pp. 170–178.

Zühlke, J. [2008]: Verfahren zur Beschleunigung der Chipverdrahtung. Diploma thesis, Research Institute for Discrete Mathematics, University of Bonn, 2008.

Summary

Routing is the last major step in chip design. It naturally splits into *global* and *detailed routing*: In global routing, an approximate layout of electrical connections is computed, subject to constraints on space and signal propagation delays. Among the feasible solutions, a solution with (near-)optimum power consumption, manufacturing yield or a combination of both is to be found. Detailed routing generates an exact layout subject to constraints on the geometrical arrangement of metal shapes (so-called *design rules*), restricting search space to corridors defined by the approximate solution obtained in global routing. BonnRoute®, the routing tool developed at the Reasearch Institute for Discrete Mathematics at the University of Bonn, follows this partitioning and contains a global and a detailed routing module.

In detailed routing, BonnRoute® discretizes routing space by a *track graph* in order to simplify the problem. *Routing tracks* can be left if necessary, e.g. for pin access, but most wires are supposed to run *on-track* in order to reduce search space. In chapter 2 we show how track positions can be found such that routing space is optimally used in this discretized representation. In general, track-to-track distances are not uniform with optimum track positions. We present data structures for representing the track graph and gridless routing space, respectively, which are fast and memory efficient also with irregular track-to-track distances. To achieve this, both data structures have to be aligned with each other in a certain way. With our *fast grid* data structure for representing the track graph, we obtain a runtime improvement of more than a factor 5 in the core path search algorithm of the BonnRoute® detailed router.

At global routing scale, most design rules are irrelevant; only minimum spacing requirements are translated into resource consumptions of edge capacities in a global routing graph which is used as a much coarser representation of routing space than in detailed routing. Signal delays, power consumption and manufacturing yield depend non-linearly on the spacing between wires. Because all these dependencies are given by convex functions, we can show that a fractional relaxation of the global routing problem can be formulated as *min-max resource sharing problem*, which is defined as follows: Given finite sets $\mathcal{R}$ of resources and $\mathcal{C}$ of customers, a convex set $\mathcal{B}_c$, called *block*, and a continuous convex function $g_c : \mathcal{B}_c \to \mathbb{R}_+^{\mathcal{R}}$ for $c \in \mathcal{C}$, the task is to find $b_c \in \mathcal{B}_c$ $(c \in \mathcal{C})$

approximately attaining

$$\lambda^* := \inf\left\{ \max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(b_c))_r \mid b_c \in \mathcal{B}_c (c \in \mathcal{C}) \right\}.$$

Given a constant $\varepsilon_0 \geq 0$ and oracle functions $f_c : \mathbb{R}_+^{\mathcal{R}} \to \mathcal{B}_c$, called *block solvers*, which for $c \in \mathcal{C}$ and $\omega \in \mathbb{R}_+^{\mathcal{R}}$ return an element $b_c \in \mathcal{B}_c$ with

$$\omega^\top g_c(b_c) \leq (1+\varepsilon_0) \inf_{b\in\mathcal{B}_c} \omega^\top g_c(b),$$

we describe a simple algorithm in chapter 3 which solves this problem with an approximation guarantee $1+\varepsilon_0+\varepsilon$ for any $\varepsilon > 0$, and whose running time is

$$O(|\mathcal{C}|\theta\rho \log|\mathcal{R}|(\log|\mathcal{R}|+\varepsilon^{-2}))$$

for any fixed $\varepsilon_0 \geq 0$, where θ is the time for an oracle call and

$$\rho := \max\left\{1, \sup\left\{ \frac{(g_c(b))_r}{\lambda^*} \mid r \in \mathcal{R}, c \in \mathcal{C}, b \in \mathcal{B}_c \right\}\right\}.$$

While the fastest previously known algorithms have a linear runtime dependency on $|\mathcal{R}|$, the runtime of our algorithm grows only logarithmically with $|\mathcal{R}|$. In contrast to many previous results which require *strong block solvers*, i.e. $\varepsilon_0 = 0$ or $\varepsilon_0 > 0$ can be chosen arbitrarily small, our result applies to both strong and weak block solvers, i.e. there is no constraint on the choice of ε_0.

In the context of global routing, customers are sets of *pins* (called *nets*) to be connected with each other, blocks are convex hulls of incidence vectors of Steiner forests, and block solvers are implemented by an algorithm for the (group) Steiner tree algorithm in graphs. Our result is of immediate significance for global routing: Here, as in many other practical applications, ρ is bounded by a small constant (in most cases even $\rho = 1$), and weak block solvers are used to achieve fast runtimes. With our algorithm, it is possible for the first time to find solutions to large instances of the global routing problem with provably near-optimum power consumption or manufacturing yield. In the largest instances today, up to 10 million customers must be coordinated to concurrently use 40 million resources in a (near-)optimum way.

In order to further reduce runtime, we present an efficient parallelized version of our algorithm designed for shared-memory parallel computers. It allows block solvers to work on outdated data, making use of the approximation parameter ε to accomodate for communication delays between processors in a shared-memory multi-processor computer. The algorithm detects cases in which solutions returned by a block solver have been computed based on too old data and in these cases schedules the corresponding block solver for sequential recomputation. This makes it possible to prove correctness of the algorithm despite block solvers working with outdated information. It turns out that the time spent on sequential recomputations is sufficiently low to obtain speedups of up to 8x on 8 processors and 14x on 16 processors, in particular on large global routing instances.

In chapter 4 we discuss the global routing problem in detail and show how it can be (fractionally) solved by the resource sharing algorithm developed in chapter 3. As we are interested in an integral solution, i.e. a Steiner forest for each net, we combine our algorithm with randomized rounding. A feasible integral solution can then be found easily by an iterative refinement procedure in practice. We generalize Steiner forests to *tree hierarchies* with topology restrictions and present a block solver algorithm which for given resource prices computes a tree hierarchy of minimum cost. This allows to address an important step in hierarchical design methodologies, namely the *port assignment* at hierarchy boundaries. We discuss many implementation details of the new BonnRoute® global router developed within the scope of this thesis and propose an approach to implement very fast block solvers by fast tree enumeration. Finally, experimental results on a large number of recent industrial chips demonstrate that optimizing yield or power with our algorithm significantly improves these objectives. Our algorithm reduces *critical area*, which measures the defect sensitivity of a chip, by up to 30 percent, increasing manufacturing yield correspondingly. Thanks to the efficient parallelization of the resource sharing algorithm, the runtimes to achieve these results are drastically cut down. Even the largest chips with millions of customers and resources can be optimized in a few minutes to a few hours, depending on the optimization objective.