Efficient and Deterministic Parallel Placement for FPGAs

ADRIAN LUDWIN and VAUGHN BETZ, Altera Corporation

We describe a parallel simulated annealing algorithm for FPGA placement. The algorithm proposes and evaluates multiple moves in parallel, and has been incorporated into Altera's Quartus II CAD system. Across a set of 18 industrial benchmark circuits, we achieve geometric average speedups during the quench of 2.7x and 4.0x on four and eight processors, respectively, with individual circuits achieving speedups of up to 3.6x and 5.9x. Over the course of the entire anneal, we achieve speedups of up to 2.8x and 3.7x, with geometric average speedups of 2.1x and 2.4x.

Our algorithm is the first parallel placer to optimize for criteria other than wirelength, such as critical path length, and is one of the few deterministic parallel placement algorithms. We discuss the challenges involved in combining these two features and the new techniques we used to overcome them. We also quantify the impact of maintaining determinism on eight cores, and find that while it reduces performance by approximately 15% relative to an ideal speedup of 8.0x, hardware limitations are a larger factor and reduce performance by 30–40%. We then suggest possible enhancements to allow our approach to scale to 16 cores and beyond.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*; B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: FPGAs, parallel placement, timing-driven placement

ACM Reference Format:
Ludwin, A. and Betz, V. 2011. Efficient and deterministic parallel placement for FPGAs. ACM Trans. Des. Autom. Electron. Syst. 16, 3, Article 22 (June 2011), 23 pages.
DOI = 10.1145/1970353.1970355 http://doi.acm.org/10.1145/1970353.1970355

1. INTRODUCTION

The Quartus II design software is a commercial CAD tool used to implement designs on Altera FPGA devices. This article describes the parallel simulated annealing algorithm used by Quartus II and improves on the approaches described in Ludwin et al. [2008].

1.1 Motivation

Since 1998, FPGA device size has increased at nearly four times the rate of per-core processor performance (Figure 1). In addition, there is now broad agreement that growth in per-core performance is slowing, and that future processors will have more cores to compensate for this reduction in per-core progress [Sutter 2005].

Authors' address: A. Ludwin and V. Betz, Altera Corporation; email: {aludwin, vbetz}@altera.com.

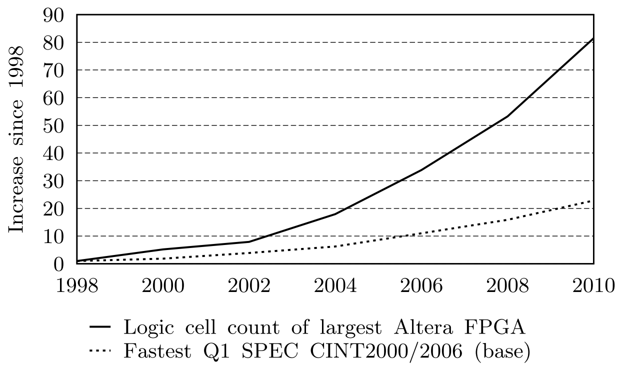


Fig. 1. FPGA size vs. per-core performance.

There are three possible approaches to keeping the runtime of FPGA CAD reasonable.

(1) Discourage flat compilation of the entire design, and instead force users to compile partitions of their designs incrementally and assemble the partitions to form the entire design [Altera 2010]. This approach, taken by many ASIC flows, can mitigate runtime but at the cost of increased design complexity and disallowed optimizations between partitions.
(2) Find faster single-threaded algorithms, while sacrificing little or no quality. This approach has been very productive but it is risky to depend entirely on this approach to offset the exponential growth in FPGA cell counts.
(3) Create parallel algorithms, possibly by modifying existing ones, to take advantage of the multicore processors which are now becoming common. Commodity PC processors now contain four to eight cores. Since the number of cores is expected to increase exponentially for the foreseeable future, parallel algorithms may be well suited to handle the increasing size of FPGAs. This is the approach we explore here.

There are several computationally complex steps in converting an HDL description of a design into the set of bit settings needed to program an FPGA to implement the design [Hutton and Betz 2006]. Usually placement, which involves choosing a good location for every circuit element in the design, is the largest single consumer of CPU time in the FPGA CAD flow. For the largest 28 nm FPGAs, placement involves choosing the location of about a million circuit elements (cells), and can require one or more CPU hours. Consequently this work focuses on accelerating the placement problem.

1.2 Constraints

Parallelizing a commercial FPGA placement tool involves respecting several constraints which are not commonly encountered in prior parallel placement work. Firstly, it must run on commodity hardware such as Windows and Linux desktops, which are the predominant platforms in the FPGA design community.

Secondly, most FPGA designers will not tolerate a significant degradation in quality relative to existing tools. The Quartus II placement algorithm optimizes wirelength, critical path delay, localized routing congestion, and power, as described in Section 3. Furthermore, key Intellectual Property (IP) cores, such as high-speed memory interfaces, require particularly high-quality timing optimization. Finally, commercial tools must handle complicated circuits that include elements such as arithmetic chains, RAM and DSP blocks, and sophisticated floorplanning constraints. Therefore, while the prior work only considers wirelength-driven optimization on relatively simple architectures, our algorithm targets complex architectures and delivers equivalent quality and features to those of the existing algorithm.

Thirdly, the placer must be deterministic. That is, when run multiple times, it must always return exactly the same result. This constraint is rarely studied in prior work (two exceptions are Chandy et al. [1997] and Sun and Sechen [1997]), but is vital in a commercial context for several reasons.

— When a bug is reported, we must be able to reproduce the problem. Nondeterminism makes this extremely difficult, even if the problem is not caused by the parallel algorithm.
— We run tens of thousands of regression tests prior to each release of Quartus II. It would be difficult to diagnose failing tests whose results changed randomly.
— In our experience, many FPGA designers do not accept nondeterministic tools. Security-conscious designers consider nondeterminism to be inherently untrustworthy, and nondeterminism can complicate timing closure by making it impossible to reproduce an earlier placement.

In addition to determinism, there is an even stronger constraint we can apply to our algorithm, known as *serial equivalency*. This is the property that the algorithm must give exactly the same answer, regardless of how many processing cores are used. A serially equivalent algorithm is clearly deterministic as well.

While serial equivalency is not as critical as determinism, it has three clear advantages. Firstly, the quality of the parallel algorithm is easily shown to be identical to that of the original, serial algorithm[1], thus meeting our second constraint. Secondly, testing can be simplified (and automated) since any difference between serial and parallel results proves, by definition, the existence of a bug. Finally, we have found that designers appreciate serial equivalency, since it allows them to move their designs between different machines, or to temporarily reduce the number of processors devoted to Quartus II, and still be assured that they will get the same answer. As we will show in the remainder of this article, achieving serial equivalency had an acceptable impact on the speedups obtained by our algorithm. Therefore, the algorithm presented in this article is serially equivalent.

1.3 Article Organization

This article is organized as follows. The next section summarizes relevant prior work. Section 3 describes the serial Quartus II placement algorithm, and Section 4 describes our parallel algorithm. Section 5 describes the hardware platforms on which we test our algorithm, and Section 6 details our results. Section 7 summarizes our contributions and suggests areas for future work.

2. PRIOR WORK

An earlier, less advanced version of this work appeared in Ludwin et al. [2008]. This article advances our work in several ways. We develop a new, more scalable method for resolving "collisions" between cores. We apply our parallel techniques to the entire anneal, rather than just to the quench. We present speedup results on more recent multicore processors, including two eight-core machines. Finally, we develop a new, more useful method to determine and quantify the sources of efficiency loss that reduce our speedup from the ideal.

[1]In practice, some minor modifications were made to the serial algorithm to make parallel development possible. These had no impact on any of our quality metrics.

```
P = InitialPlacement();
T = InitialTemperature();

while (ExitCriterion() == False) {
   while (InnerLoopCriterion() == False) {  /* One temperature */
      Pnew = PerturbPlacementViaMove(P);    /* Propose move */
      ΔCost = Cost(Pnew) - Cost(P);        /* Evaluate move */
      r = random(0,1);                      /* Finalize move */
      if (r < exp(-ΔCost/T)) {
          P = Pnew;                         /* Accept move */
      }
   } /* End one temperature */
   T = UpdateTemp(T);
}
```

Fig. 2. High-level algorithm for simulated annealing placement, adapted from Betz [2007].

With regards to other published work, we advance the state-of-the-art in several ways. Firstly, we optimize many placement objectives, including timing, whereas the previous published work in parallel placement optimizes only for wirelength. Secondly, our algorithm is deterministic, while most previously published algorithms with good performance are nondeterministic. Combining these two features represents a significant advance, as discussed in Sections 3 and 4.

2.1 Placement Algorithms

Considerable research has been performed into the placement problem. The most popular approaches include recursive partitioning [Alpert et al. 1996; Caldwell et al. 1999; Sarrafzadeh et al. 2003], analytic [Chan et al. 2000; Kleinhans et al. 1991; Viswanathan and Chu 2004], genetic [Borra et al. 2003], and simulated annealing [Betz et al. 1999; Kirkpatrick et al. 1983; Sechen and Sangiovanni-Vincentelli 1985]. There have also been attempts to parallelize these various approaches for over twenty years. Recursive partitioning is parallelizable in a reasonably obvious way, at least after the first cut, provided there are enough cutlines to occupy all available cores. Analytic placement can benefit from parallelized matrix operations, and higher-level parallelism has also been extracted in Chan and Schlag [2003].

However, simulated annealing is the most-studied algorithm for FPGA placement, mainly since it directly handles FPGAs' complex legality constraints, while other approaches require sophisticated legalization steps [Hutton and Betz 2006]. The Quartus II placement algorithm (hereafter referred to as "Q2P") is based on simulated annealing and is more fully described in Section 3. In the remainder of this section, we provide an overview of simulated annealing and prior research into parallel simulated annealing.

Figure 2 gives a high-level overview of placement via simulated annealing. An initial poor-quality placement is iteratively modified by proposing small perturbations, or *moves*, to the placement state. The moves are then evaluated for their *costs*, which are heuristics to measure quality, such as the wirelength expected to be needed to route the placement or the critical path delay. Moves that reduce the placement's costs are always accepted. Those that make the placement worse still have some probability of being accepted, to allow the algorithm to escape local minima. As the anneal progresses, the "temperature" is reduced, which gradually reduces the probability of accepting moves that increase the placement cost. Furthermore, the distance across which cells are moved is steadily reduced as the quality of the placement improves.

When the temperature reaches 0, we are said to be "quenching" the placement, and only changes that reduce cost will be accepted.

2.2 Parallelizing Simulated Annealing Moves

The most popular approach to parallelizing simulated annealing has been to parallelize the execution of individual moves. Some less-popular methods are described in Section 2.3, though some of those methods may overlap somewhat as the lines between different parallelization strategies can be blurred.

To parallelize moves, entire moves are proposed and evaluated in parallel by cores all working on the same placement. Clearly, this could lead to conflicts if multiple processors accept moves that affect the same cells or nets, a situation known as a *collision*. There are several published approaches to resolving this problem.

(1) Find an independent (noncolliding) set of moves and process them all in parallel,
(2) Assign each core a partition in the placement area such that different processors' moves tend not to interact, or
(3) Make assumptions about future decisions of the annealer and speculatively process moves based on these assumptions.

2.2.1 Independent Set Finding. In the parallel moves algorithm described in Kravitz and Rutenbar [1987], the first core to accept a move forces all other cores to reject the moves they have in progress. This does not significantly affect the quality of the final result and achieves a 3.5x speedup on four cores when the acceptance rate is low. A more recent attempt uses one core to propose moves with noncolliding locations and nets, but is slower than the serial algorithm due to synchronization overhead [Haldar et al. 2000]. Similarly, moves with noncolliding nets are proposed in Banerjee et al. [1990] using a cell-coloring heuristic to reduce collisions between nets, but the placement is still partitioned into rows to prevent cell collisions. The parallel speedup is not reported. These algorithms are nondeterministic [Banerjee et al. 1990; Kravitz and Rutenbar 1987] or at best not serially equivalent [Haldar et al. 2000].

In Ludwin et al. [2008], we presented a serially equivalent algorithm that found independent move sets on-the-fly, and achieved a speedup of 2.2x at low acceptance rates. In this article, we present two improved versions of this algorithm. The first makes incremental improvements to [Ludwin et al. 2008] to eliminate false sharing and achieves a speedup of 2.5x on the same platform, at low acceptance rates. The second version is more decentralized but still maintains serial equivalency, and achieves a speedup of 2.9x on the same platform at low acceptance rates.

2.2.2 Partitioned Placements. In this approach, the placement area is divided into multiple partitions. Errors in the costs of nets that span partitions are usually tolerated, and updates are broadcast periodically to prevent the errors from becoming too large. Some implementations occasionally modify the partitions to allow cells to migrate across the entire chip [Haldar et al. 2000; Sun and Sechen 1997] while others allow cells to be transferred to other partitions at any time [Chandy and Banerjee 1996; Kim et al. 1994]. Some authors report speedups of 2–2.5x on four cores at a cost of slightly increased wirelength [Chandy and Banerjee 1996; Kim et al. 1994], and many of these implementations are nondeterministic. However, Sun and Sechen [1997] obtained excellent speedups of 5.3x using six cores linked by a LAN, with no impact in wirelength quality. Their approach is also deterministic since cores are fully isolated from one other, except during updates that occur at predetermined times. The authors also show that the cross-partition error in their bounding box cost tends towards zero over the course of the anneal.

Unfortunately, this approach was not directly applicable to our algorithm. Sun and Sechen heavily exploit spatial locality, but as we describe in Section 3, this locality is not present to the same degree in Q2P. Firstly, Q2P proposes large moves, whereas partitioning depends on the ability to improve placement using only localized moves within a single partition. Secondly, Q2P disproportionately proposes moves for a small number of cells; if these cells are located in a small number of partitions, much of the advantage from parallelization will be lost. Lastly, it is far from certain that the error in a timing cost (particularly a path-based timing cost) would converge in a similar fashion to that of a bounding box cost. Therefore, we have not used this approach, though we discuss ways we might adapt partitioning to Q2P in future work in Section 7.

2.2.3 Speculative Computation. In this approach, first described in Witte et al. [1991] for the task assignment problem, the decision tree of the annealer is mapped to the number of available cores. For example, on a three-core system, while core C0 evaluates move M1, cores C1 and C2 propose and evaluate move M2, with C1 assuming that M1 will be accepted and C2 that it will be rejected. Once a decision is reached for M1, the result for M2 is immediately selected and the annealer proceeds to M3. This technique preserves serial equivalency. With N cores, we can speculate between $\log_2 N$ and N moves into the future, depending on the ratio of accepted to rejected moves. High speedups are indeed reported in Witte et al. [1991], but significant slowdowns are reported when applied to placement in Chandy et al. [1997]. The largest cause, once again, is synchronization overhead, and furthermore, it takes as long to speculatively perform a placement move as it does to fully evaluate it, limiting speedups when the acceptance rate is high.

Both of the algorithms in this article speculatively propose moves, but they do not require that previous moves are rejected. Instead, a dependency checker is used to ensure that speculatively proposed moves do not interact with any accepted moves.

2.3 Other Parallelization Strategies

Move acceleration, as described in Kravitz and Rutenbar [1987], parallelizes the evaluation of each move by evaluating different parts of the costs on two cores, and additionally by using a third core to propose the next move. This algorithm yields a speedup of 2x on three cores, but is not easily scalable to larger numbers of cores and has been little studied since it was originally proposed. In addition, we were unable to successfully parallelize our own cost evaluation function, due to the high synchronization overhead we discuss in our previous paper [Ludwin et al. 2008].

In this previous paper, we did use a similar strategy to "pipeline" our algorithm into two cores, one to propose future moves and another to evaluate them. We were only able to achieve speedups of up to 1.3x due to cache and memory bandwidth limitations, and had no success in attempting to scale the algorithm further by using multiple pipelined stages. Therefore, we do not further develop it in this article.

At the other extreme, by assigning different cores to completely different placements, more parallelism can be introduced. This technique is known as the parallel Markov chain [Aarts et al. 1986]. Every so often, the core with the lowest-cost placement broadcasts its solution to the other cores. To obtain a speedup of X, the number of moves per core is simply divided by that number, though quality may suffer. Broadcasts are typically sent asynchronously to increase efficiency [Chandy and Banerjee 1996; Haldar et al. 2000].

Most authors have attempted to find the best speedup for similar quality of the serial version. Using four cores, speedups between 2.5x–2.9x are reported, at the cost of some wirelength increase [Chandy and Banerjee 1996; Haldar et al. 2000]. This technique cannot be serially equivalent without excessive runtime overhead and is nondeterministic if asynchronous updates are used. It is not further explored here.

More exotic parallel placement algorithms have been proposed, including hardware-assisted simulated annealing [Wrighton and DeHon 2003]. However, this approach is currently unable to handle circuits larger than about 400 cells when implemented on modern FPGAs. While interesting for future research, it is not explored here.

3. THE QUARTUS II PLACEMENT ALGORITHM

The core Quartus II placement algorithm (Q2P) is derived from VPR 4.30 [Betz et al. 1999], and as such uses simulated annealing as its main component. In the ten years of development since it branched from VPR[2], many advances have been incorporated into the algorithm, which make extracting parallelism from our algorithm considerably more difficult than in the prior work.

In this section, we describe the new features of Q2P relative to VPR and their impact on parallelization. We also attempt to compare the serial performance of Q2P to VPR and other published algorithms.

3.1 New Features

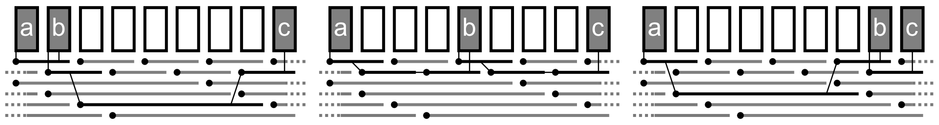
Q2P has three major areas of improvement over VPR: directed moves, improved costs, and multilevel placement. The first two of these challenge any attempt to parallelize our algorithm as will be discussed shortly.

— *Directed moves.* Traditional simulated annealing algorithms propose moves entirely at random within a window that shrinks as the anneal progresses. In Q2P, the vast majority of our moves are *directed*; that is, while they are partially random, both the selection of the block(s) to move and the destination location(s) are heavily biased. This allows Q2P to explore the search space far more efficiently than with purely random moves. Furthermore, the blocks can move much farther than is possible in conventional simulated annealing, as described shortly. A similar technique was first described in Vorwerk et al. [2007, 2009]; Q2P does not use the moves described in those articles, but uses approximately ten different types of directed moves. In particular, about a quarter of our moves target the 1% of cells that are on the most critical paths in the circuit.

— *Improved costs.* Q2P uses eight costs to optimize wirelength, timing, power and localized congestion; they are combined linearly to produce an overall cost used to accept or reject moves. The wirelength and congestion costs are calculated using bounding boxes conceptually similar to those of VPR, with the latter also making use of periodic congestion analyses. The power costs are specific to Altera devices. The timing costs are substantially more complex than those of VPR, and combine aspects of edge-based and path-based approaches.

— *Multilevel placement.* VPR uses clustering to create logic blocks of several LUTs and registers. Similarly, Q2P also clusters the netlist into logic, RAM, and DSP blocks, and performs much of its placement with these large blocks. In addition, Q2P is able to decompose its logic clusters back into their component LUTs and registers late in the placement process, allowing for "fine-tuning" of the placement details. This technique first appeared in Timberwolf [Sun and Sechen 1995] and has also been proposed as an extension to VPR [Chen and Cong 2004]. This results in a

[2]For the remainder of this article, "VPR" refers to VPR 4.30, not VPR 5.0 [Luu et al. 2009] unless otherwise noted.



(a) initial: 450ps (b) intermediate: 600ps (c) final: 450ps

Fig. 3. High costs in intermediate placements.

substantial improvement in various quality metrics, but since it greatly increases the effective size of the placement problem, it comes at a significant runtime cost.

The overall effect of the first two of these improvements is somewhat akin to embedding many local improvement placers in a simulated annealing framework, thereby combining the best features of each into a unified algorithm.

We now describe why Q2P moves cells across wider distances than in conventional simulated annealing. Prior work assumes that a placement can be continuously improved by making many small steps, so even when only very localized moves are allowed, the placement can change considerably as cells slowly migrate towards better destinations. However, the routing architecture of modern FPGA devices inhibits the ability to make changes in small increments.

An illustration of this problem is shown in Figure 3(a), which shows a short path of three logic cells. Cells “a” and “c” cannot be moved (for example, because they may be device IOs fixed by board constraints, or have other critical fanout), but “b” is free to move between them. In addition, cell “b” should ideally be towards the right of the device (for example, due to additional nets not shown here), so we should accept moves that move it towards the position in Figure 3(c).

This device uses two types of wire for routing: slow, short wires with high connectivity and fast, long wires with limited connectivity. In our illustration, the short wires have length two and a 100ps delay, and the long wires have length six and a delay of 150ps. The longer wires are only connected to each other and to the short wires, and cannot drive or be driven directly by logic cells. Due to this architecture, while the initial path delay from “a” to “c” is only 450ps, the path in Figure 3(b) has a much higher delay of 600ps. Therefore, the timing cost will discourage any move from the left to the center of the device, even though this is a necessary step when moving to the right using localized moves.

While this illustration is simplistic, similar situations do occur in real circuits, and cells would become trapped in deep local minima[3] when only localized moves are allowed. However, Q2P’s directed moves can skip such intermediate positions, and these moves are more likely to be accepted.

Finally, in addition to the three areas outlined earlier, we have made a large number of runtime optimizations to the code. For example, as we are evaluating the costs, we are able to detect that a move will have a very small chance of being accepted, and therefore abort the rest of the calculations. We have also made a large number of changes that are necessary to support Altera-specific devices[4]. Key among these is support for dedicated device-wide routing resources that are used to distribute high-fanout signals such as clocks. Since these resources are limited, the placement algorithm must take care not to place too many signal sinks in a given region.

[3]Theoretically, a slower annealing schedule would be able to escape such local minima. In practice, the runtime required for such a schedule would be prohibitive.

[4]Some of these improvements, such as heterogeneity, are also present in VPR 5.0.

In summary, Q2P contains many changes relative to VPR that are required to produce high-quality results in a reasonable runtime on modern FPGA devices. Unfortunately, many of these changes directly impede our ability to parallelize the algorithm.

— The timing-driven moves disproportionately target a very small number of cells. All prior work described in Section 2.2 assume that the placer has an equal probability of moving any cell from any region of the chip at any time. By reducing the number of cells that are targeted for moves, we reduce the amount of inherent parallelization available for exploitation.
— The directed moves may move cells across a wide distance. Localized moves improve spatial locality and hence increase parallelization, but our directed moves reduce spatial locality.
— The path-based components in our timing cost imply that changes in one region of the device may impact costs in a physically distant region. Similarly, tracking the restricted clocking resources must be done across the entire region spanned by the resources, which can be a significant portion of the chip. These further reduce spatial locality (and therefore the available parallelism) since fewer parts of the design are fully independent of each other.

Despite these limitations, we found that we are still able to extract a considerable amount of parallelism from our algorithm without simplifying it. The methods involved were considerably more complex than those that have been studied in prior work, and are fully described in Section 4.

3.2 Performance Comparison to Published Algorithms

To show that Q2P has high result quality, and hence is a worthy target for parallelization, we compare its performance to that of other published algorithms, including VPR. Simulated annealing is often believed to scale more poorly than other approaches, such as partitioning or analytic placement. However, thanks to the improvements we outlined before, we show that our placer scales well to larger problems.

It is difficult to directly compare the quality of Q2P to VPR or any other published algorithm, as it is very tightly integrated into the rest of Quartus II and is specialized for Altera devices. Other placers (such as VPR) either work on idealized devices or for standard cell circuits that our placer does not support (such as FastPlace [Viswanathan and Chu 2004]). However, since Q2P is derived from VPR, it is possible to disable the various algorithmic enhancements that are not needed to produce a result on Altera devices, leaving us with an approximation of the VPR placement algorithm[5] that we can use for a comparison. To expand this comparison, we can compare our results to those in Bian et al. [2010], who adapted three state-of-the-art ASIC placers to target simplified FPGA architectures and compared them to VPR.

We believe our comparison is conservative for Q2P relative to VPR, since it is not actually possible to remove all optimizations we have made to the algorithm. Similarly, the comparison by Bian et al. is conservative for VPR relative to the ASIC placers, since VPR optimizes for wirelength and critical delay but the ASIC placers only optimize for wirelength. Therefore, the comparison between the Q2P and the alternatives are weighted in favor of the alternative algorithms.

We compared our placer to our approximated VPR (hereafter "qVPR") using the same industrial circuits described in Section 6.1. We ran the two algorithms in both

[5]As this article is not concerned with the performance of the clusterer or router, we used the Quartus II versions of these algorithms for all experiments in order to isolate the effects of the changes to the annealer.

Table I. Comparison of Q2P versus qVPR: Default Settings

		crit. path (ns)		wire (1e5)		time (s)		relative		
circuit	% LABs	qVPR	Q2P	qVPR	Q2P	qVPR	Q2P	path	wire	time
circuit1	34%	3.6	3.4	2.6	2.4	135	82	0.926	0.916	0.607
circuit2	37%	3.6	3.3	3.3	3.1	209	107	0.920	0.958	0.513
circuit3	26%	2.9	2.5	3.4	3.1	291	174	0.862	0.904	0.597
circuit4	20%	3.8	3.6	4.8	4.5	517	208	0.939	0.946	0.403
circuit5	71%	3.9	3.7	3.0	2.8	136	88	0.940	0.938	0.644
circuit6	46%	6.0	5.0	8.2	8.0	686	271	0.842	0.978	0.395
circuit7	91%	3.9	3.6	4.9	4.4	325	221	0.914	0.896	0.681
circuit8	100%	4.0	3.5	5.1	4.7	401	287	0.880	0.920	0.717
circuit9	35%	3.5	3.1	5.9	5.5	658	443	0.871	0.933	0.674
circuit10	73%	3.2	2.9	8.4	8.4	828	411	0.895	0.998	0.497
circuit11	99%	3.9	3.6	9.5	9.1	935	542	0.926	0.956	0.580
circuit12	96%	6.5	6.2	15.0	14.5	1497	700	0.955	0.966	0.467
circuit13	90%	3.5	3.0	8.5	8.5	1061	1514	0.852	0.996	1.426
circuit14	100%	2.9	2.4	20.3	17.6	1180	1035	0.831	0.863	0.877
circuit15	100%	7.0	5.2	14.7	13.0	1329	1886	0.745	0.886	1.419
circuit16	75%	5.4	3.1	14.6	14.1	2511	1341	0.564	0.965	0.534
circuit17	99%	3.4	3.0	14.6	13.5	1217	1054	0.903	0.924	0.866
circuit18	100%	3.7	2.5	25.2	20.2	3426	1638	0.655	0.802	0.478
Average								0.850	0.929	0.641
High-utilization average								0.818	0.891	0.774

Table II. Comparison of Q2P to Other Placers

inner-num		relative		
qVPR	Q2P	path	wire	time
timing-driven:				
1	1	0.850	0.929	0.641
10	1	0.883	0.984	0.065
10	10	0.882	0.874	0.611
wirelength-driven:				
1	0.1	–	1.244	0.087
1	1	–	0.917	0.674
10	1	–	0.968	0.069
10	10	–	0.863	0.589

(a) Q2P vs. qVPR

placer	wire	time
IBM, 1 cell/LAB:		
Capo	1.053	0.541
mPL	1.149	0.296
FastPlace	1.316	0.048
FastPlace+MDP	1.064	0.023
QUIP/IWLS, 10 LUTs/LAB:		
Capo	0.990	1.887
mPL	1.124	2.439
FastPlace	1.299	0.085
FastPlace+MDP	1.149	0.039
Industrial, 20 LUTs/LAB:		
Q2P in = 1	0.917	0.674
Q2P in = 0.1	1.244	0.087

(b) Bian et al. and Q2P vs. VPR/qVPR

timing-driven and wirelength-only mode, and varied their effort levels by adjusting the number of moves performed per temperature (a parameter called "inner-num"). The per-circuit results with default effort levels and timing-driven optimization are shown in Table I, and the results of varying the effort levels are shown in Table II(a). For all relative numbers, qVPR is used as the denominator, which means that lower numbers are better for Q2P for critical path length, wire usage, and runtime. As in Section 6.1, we only consider the runtime of the moves themselves, not of other steps such as timing analysis.

We found that Q2P significantly outperforms qVPR in timing, wirelength, and runtime. In fact, we outperform qVPR's timing and wirelength results in every single circuit, and simultaneously outperform its runtime in all but two circuits. On average, Q2P achieves an 15% better critical path length, with the best results in the circuits with the highest utilization (over 99% of all LABs fully or partially used). It also achieves 7% better wirelength usage and about 35% lower runtime. Most notably, qVPR is unable to match Q2P's wirelength or critical path results, even when it was given 10x the runtime to do so. This indicates that Q2P has been fundamentally improved relative to qVPR, and presumably to VPR as well.

Finally, in Table II(b), we indirectly compare Q2P to the ASIC placement algorithms studied in Bian et al. [2010]. In that work, the authors ran each algorithm on four different benchmark sets with varying amounts of available whitespace. We summarize the results of two of these benchmarks here: the IBM ASIC circuit set and the QUIP/IWLS set with a cluster size of 10 LUTs[6]. In addition, we have only shown the near-zero whitespace results. In our experience, designers typically purchase the smallest devices possible, so this is the most realistic testcase.

Our results suggest that Q2P is at least competitive with other published algorithms, even when it is restricted to only optimizing wire. Using VPR/qVPR as a common baseline, and using default effort levels, Q2P matches or outperforms all other algorithms with respect to wirelength use. Compared to Capo [Caldwell et al. 1999] and mPL [Chan et al. 2000], its runtime is worse on the IBM circuit set but significantly better on the 10-LUT QUIP set that more closely approximates real FPGA circuits. In addition, by reducing the effort level of Q2P by a factor of ten, we achieve similar results to FastPlace in terms of both wirelength and runtime, so long as its "native" detailed placer is used.

We cannot match the results of FastPlace when using the MDP detailed placer described by [Bian et al. 2010]. However, we believe these results show that Q2P is competitive with other published algorithms, even when using the wirelength-only comparison that ignores much of the value of our own placer, and that does not capture the full complexity of a modern FPGA device. Consequently, these results motivate our decision to parallelize Q2P rather than attempting to use an entirely different approach.

4. PARALLEL ALGORITHM DESCRIPTION

4.1 Overview

As described in the preceding section, the Quartus II placer algorithm (Q2P) is a variant of simulated annealing that uses highly directed moves and many cost components, both of which significantly reduce the spatial locality that many prior works exploit to introduce parallelism. Consequently, parallelization strategies such as partitioning (which directly exploit spatial locality) or precomputing noncolliding moves (which exploit the simplicity of random moves) are not appropriate for our algorithm. Instead, we use a novel variant of independent set finding (Section 2.2.1) that identifies colliding moves on-the-fly, instead of in advance. We also include features that allow us to "repair" colliding moves to ensure determinism and serial equivalency. These two features allow us to use the existing directed moves and costs that are essential for our algorithm's high quality.

At a high level, our parallelized annealer can loosely be thought of as a master-worker configuration, where the worker cores propose moves and evaluate their costs in parallel, and a single master core resolves collisions between moves in serial. We call the parallel portion of a move's lifetime *processing*, and the resolution phase *finalization*. The finalization phase includes the decision as to whether the move should be accepted or rejected, based on the results of the processing phase. To be deterministic,

[6]Bian et al. [2010] did find that the ASIC placers tended to perform better on the QUIP/IWLS benchmark sets when using smaller cluster sizes, but the size of 10 LUTs most closely approximates modern FPGA devices. For example, Altera's Stratix IV devices support 20 LUTs per cluster.

a move must have all of its collisions resolved before it can be finalized. Threads use message passing for most of their communication.

Processing comprises about 99% of a move's runtime in a serial compilation, as finalization is usually extremely fast. For this reason, we found that a true master-worker-style algorithm was impractical. Devoting an entire core to finalization was wasteful, as it would be idle for the majority of the time. Even using a thread turned out to be inefficient, due to the time it took to "wake" a thread that had been sleeping. Instead, we use two methods to control the flow of moves through our algorithm, which are described in Sections 4.4 and 4.5.

When a move is accepted, a database containing the location of each cell must be updated. If this database were shared among all of the cores, any change made by one core would be immediately seen by all other cores, which may be partway through proposing or evaluating a move. This could lead to us proposing absurd moves, such as exchanging a cell with itself in another location. Even if this move were never evaluated, it would likely violate an assertion and crash the program as it was being proposed. Therefore, each core receives its own copy of the database, and a core that accepts a move must send a message to the other cores, who update their own databases in between processing moves. This is similar to Sun and Sechen [1997], who were forced to duplicate their placement database because it was parallelized across completely different machines.

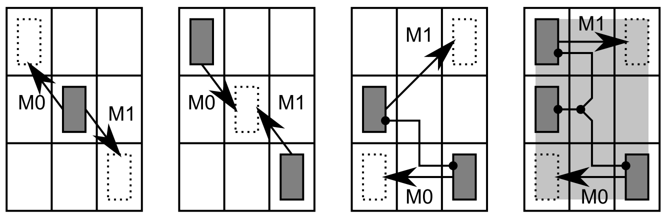
4.2 Repairing Colliding Moves

As described in Section 2.2, speculatively processed moves may collide. By duplicating the placement databases, we ensure the algorithm will not crash, but the algorithm will be nondeterministic unless further action is taken. We cannot simply reject moves that may be invalidated by the accepted move, as do Kravitz and Rutenbar [1987], since moves are completed asynchronously.

Instead, we must "repair" the move in a deterministic fashion. One way to do this is by fully reprocessing a move if there is any chance it has collided with an earlier move. For example, as move MN is first proposed, we could record the last previous move that was accepted ("MP"). If we eventually find that another move later than MP (but before MN) in serial order had been committed by another core, we observe that there is a chance that MN may have been involved in a collision. We might then discard MN's speculative results, and reprocess it from scratch.

Through careful treatment of random numbers, it is fairly easy to ensure that the reprocessed move is identical to the same move in the serial placer. Each move consumes several random numbers: the type of directed move, the source cell, and its destination location require a minimum of three random numbers, but in practice many more may be required based on the type of move and the state of the placement. This makes it difficult or impossible to use a single PseudoRandom Number Generator (PRNG) for the algorithm, or even to use a single PRNG per core, since we do not know in advance which move will be processed by which core. Instead, we use a new PRNG for *every* move, each of which is seeded by a combination of the move's ID and a user-specified seed. When a move needs to be reprocessed, we simply reset its own PRNG back to its initial state. This ensures that the last time a move is reprocessed, the same stream of random numbers is generated as in the serial algorithm, and as long as the rest of the placement is identical to the serial algorithm, the proposed move and all of its costs will be identical to the equivalent serial move as well.

The linear congruent PRNG, used by VPR and the C runtime library, turned out to be inappropriate when seeded with the moves' IDs, as it produced pseudorandom sequences that were offset by one for every move. We found that the Coveyou PRNG



(a) cell (b) location (c) connection (d) net

Fig. 4. Collisions types.

[Knuth 1997] gave equivalent quality of results as the linear congruent PRNG, was simple to implement, and performed well for our purposes when seeded by consecutive integers.

4.3 The Dependency Checker

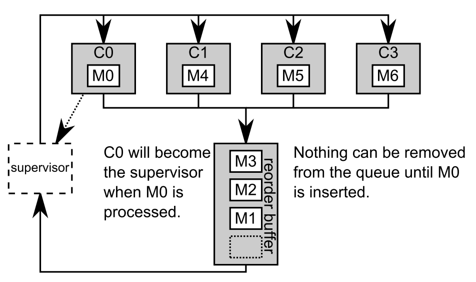
Fully reprocessing any moves that *may* have collided only works well at extremely low acceptance rates (well under 1%), but we found it inefficient for higher acceptance rates. In designs with hundreds of thousands of cells or more, many moves do not actually collide, so this approach throws away valid moves and recomputes them, wasting CPU time. Therefore, we have created a *dependency checker* to track each move's dependencies, and only reprocess any part of the move found invalid.

There are two types of collisions the dependency checker must detect and repair efficiently. The first involves cells and their locations. If two moves affect either the same cell (Figure 4(b)) or the same location (Figure (4(a)), then if one move is committed, the other cannot be performed. Moreover, in the equivalent serial flow, the later move would never have been proposed in the first place. These dependencies[7] are called *proposal dependencies*, and can only be resolved by a *full reprocessing*, in other words, reproposing the later move from scratch and recomputing any costs that had already been calculated.

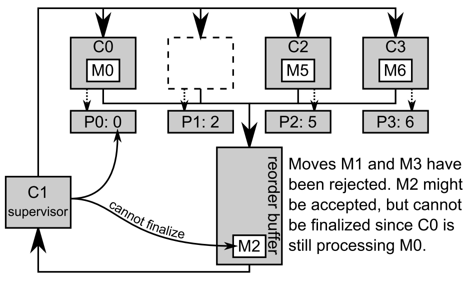
The second kind of collision involves elements of the cost computation. In this case, if the earlier move is committed, the later move can still be performed. However, the cost delta that was computed for the later move relied on state that is now out-of-date, and hence the delta may be incorrect, which may affect the decision to accept or reject the move. For example, if one cells feeds another (Figure 4(c)) and both are moved in parallel, the decision to accept or reject the later move should take into account the result of the earlier move. This may occur even if the cells do not communicate with each other but are fed by the same source (Figure 4(d)), since either move may change the Half-Perimeter Wire Length (HPWL) of the net, shown here in grey. These dependencies are called *evaluation dependencies*. If one is encountered, the move does not have to be reproposed nor does it even have to be fully reevaluated. Instead, only those portions of the cost function involving the affected elements must be recalculated. We call this a *partial reprocessing*.

As shown in the previous examples, the exact resources tracked by the dependency checker vary depending on the cost. For example, if a small net counts towards the HPWL cost but a clock net does not, only the former will be tracked by the

[7]In practice, the affected cells themselves do not need to be tracked, since if two moves affect the same cell, they must also affect the same location.


C0
M0
C1
M4
C2
M5
C3
M6
supervisor
C0 will become the supervisor when M0 is processed.
M3
M2
M1
reorder buffer
Nothing can be removed from the queue until M0 is inserted.

(a) centralized


C0
M0
C2
M5
C3
M6
P0: 0
P1: 2
P2: 5
P3: 6
C1
supervisor
cannot finalize
M2
reorder buffer
Moves M1 and M3 have been rejected. M2 might be accepted, but cannot be finalized since C0 is still processing M0.

(b) decentralized

Fig. 5. Finalization strategies.

dependency checker. The programmer must manually track all resources used to evaluate a move, as the specific implementation of the costs determines exactly what must be tracked.

It is helpful to think of the dependency checker as a limited implementation of a transactional memory system. The only difference is that modified locations in memory are tracked manually, rather than being automatically handled by the platform. Since only a tiny fraction of the memory accessed by the algorithm can cause dependencies, this is also more efficient, as others have found in similar physical design problems (for example, Watson et al. [2007]).

4.4 Centralized Finalization

For the dependency checker to work correctly, moves may only be finalized once we are certain that no other move will be accepted. Our first approach to ensuring this is to insert every move into a *reorder buffer* after processing. This buffer allows moves to be inserted in any order, but only allows them to be removed in strictly ascending order. Then, instead of having a single "master" core (or thread) responsible for removing moves from this queue and finalizing them, we allow any one core to take on this role at a given time. We call this core the *supervisor* while it is performing this role. As a part of finalization, the supervisor also performs dependency checks and reprocesses moves as described in Sections 4.2 and 4.3. Only one core may behave as the supervisor at a time; we enforce this through a shared variable as we found message passing to be too slow for this purpose.

An illustration of this process can be found in Figure 5(a). When core C0 completes moves M0, it will become the supervisor and finalize moves M0 to M3, and broadcast the results back to the other cores. Note that if M0 is accepted, moves M1 and M3 must have their dependencies checked by the supervisor before they can be finalized. The supervisor also performs any reprocessing that is required.

Each move is described by a significant amount of state (and hence memory), and so we found it essential to limit the number of moves that are active at a time. If *all* in-flight moves are currently waiting in the reorder buffer, the cores must stall and wait for one to be finalized. We have found that four moves per core gives good performance, and discuss the implications of varying this number in Section 6.1.

4.5 Decentralized Finalization

During the latter parts of the anneal, including the quench, the vast majority of all moves proposed are rejected. While a rejected move has no impact on any other core, the centralized strategy of Section 4.4 ensures that many such moves will be examined by at least two cores. This requires that move data be transferred between different caches. In addition, we found that a single move that took a long time to process could cause the other cores to become idle as their moves sat in the reorder buffer, even if all the moves might eventually be rejected. To address these limitations, we developed a method that avoids the need for any centralized control to finalize a rejected move.

This approach works as follows. Threads continue to use message passing for the majority of their communication, but they add two additional means of communication: a shared counter to indicate the ID of the next move to perform, and per-core variables containing the move currently being processed by that core, known as the *progress of the core*. The shared counter allows moves to be started independently of the supervisor, and the progress variables are used to inform other cores when it is safe to finalize a move, as described next.

If all moves are being rejected, each core updates its own progress, and no core ever checks the progress of another core. Thus, with the exception of the shared counter, there is very little intercore communication.

If a core wishes to accept a move, only then does it insert the move into the reorder buffer. As in Section 4.4, the core will then attempt to supervise the algorithm unless another core is already doing so. The supervisor then reads the progress of all cores to determine whether all other cores are working on later moves. If this is the case, the move is finalized and, if accepted, it is broadcast to the other cores to allow them to update their local states. Otherwise, the slowest core is told to become the supervisor after it completes its current move.

For example, in Figure 5(b), move M2 is likely to be accepted and is waiting to be finalized. Core C1, the supervisor, examines the progress of all the cores. Cores C2 and C3 are processing moves M5 and M6, which are ahead of M2, but C0 is still processing M0. Hence C2 knows it cannot finalize M2 as not all earlier moves have been processed. Instead, it sends a message to C0 to supervise once it has finished M0, and goes back to processing regular moves. Assuming that C0 rejects M0, C0 will become the supervisor and will observe that all other cores are now working on moves later than M2. It will therefore be safe to finalize M2. In this way, the progress is used to enforce serial equivalency.

While the multiple methods of communication make this algorithm somewhat more difficult to implement than the centralized version, the basic principles remain largely unchanged. The most significant exception is that, since rejected moves are no longer supervised, we must provide a way for them to be reprocessed when necessary. To accomplish this, each core maintains a queue of *rejected records* of moves it has provisionally rejected, consisting of the moves' IDs and some statistics used for profiling.

When a move is committed, all records in the queue before the committed move have their statistics collected and then are discarded. All records after the committed move are transferred to a *reprocessing queue*, since they were proposed and rejected based on speculative state that we now know was incorrect. The moves they represent can be reprocessed before the core attempts to process any new moves.

To speed reprocessing, each core also maintains a cache of recent moves. As cores examine each record in the reprocess queue, they also check their cache to determine whether the move itself is still available. If it is, the dependency checker can be called as it is in the supervisor, to skip as much of the reprocessing as possible. If the move is not available in the cache (for example, if it had been evicted to process a newer move) it must be reprocessed by scratch.

As in Section 4.4, we chose a cache size of four moves per core, and discuss the impact of this choice in the next section.

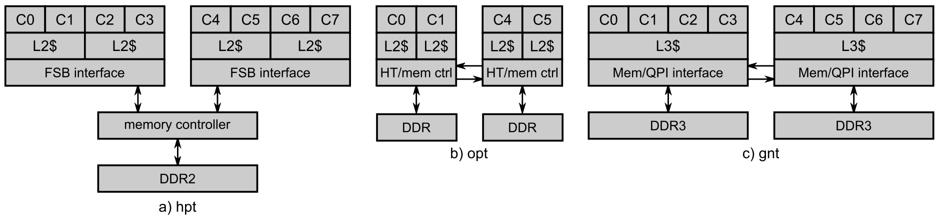


Fig. 6. Hardware diagrams.

Table III. Hardware Specifications

Name	Processor	C	N	Shared$	Bus	Memory
`hpt`	Intel Xeon E5450 3.0GHz “Harpertown”	4	2	6MB L2/C2	FSB@1.3GHz 42GB/s	DDR2@333MHZ 10.6GB/s
`opt`	AMD Opteron 275 2.2GHz “Italy”	2	2	n/a	HT@600MHz 9.6GB/s	DDR@200MHz 6.4GB/s
`gnt`	Intel Xeon E5570 2.9GHz “Gainstown”	4	2	8MB L3/4C	QPI@3.2GHz 102.4GB/s	DDR3@400MHz 51.2GB/s

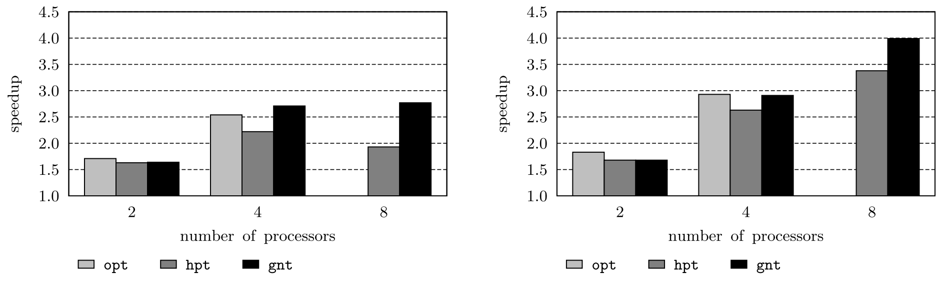
5. TEST ENVIRONMENT

All results given in this article were collected using 64-bit executables on Windows XP. The algorithms in this work were also tested on Linux using the 2.6 version of the kernel, and the results were comparable.

Three machines are used to test our algorithm. Their block diagrams are shown in Figure 6, and their specifications are given in Table III. In this table, “C” is the number of cores per processor, “N” is the number of processors, and “Shared$” describes the shared cache in the form “type of cache/number of cores sharing cache.” “Bus” and “Memory” describe the capabilities of the bus and memory subsystems, respectively: “FSB” is a front-side bus, “HT” is AMD’s HyperTransport, and “QPI” is Intel’s QuickPath Interconnect. All three systems are cache coherent; the FSB is bidirectional whereas HT and QPI use multiple unidirectional links. The bus memory bandwidths are the aggregate theoretical bandwidth of the entire system. For example, `opt` contains two unidirectional HT links, each with a nominal bandwidth of 4.8GB/s; therefore, the total bus bandwidth is 9.6GB/s. For its part, `hpt` contains two front-side buses, each with a nominal bandwidth of 21GB/s; however, they share the same 10.6GB/s connection to memory.

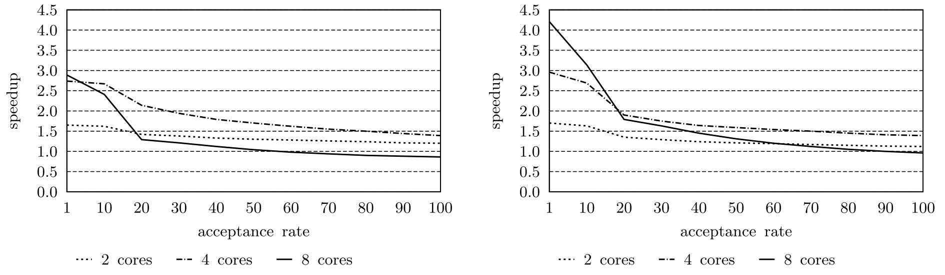
One interesting note is that `hpt` has uniform access to main memory from all cores, while cores in `opt` and `gnt` have direct access to some memory and only indirect access to other memory. This type of configuration is known as “NonUniform Memory Access,” or NUMA. We did not attempt to exploit NUMA in our algorithm.

When all the cores in a system were being used, we found that constraining software threads to specific cores had no effect, but this was not true when only some cores were in use. We typically achieved the higher performance when constraining the cores to be close together, that is, when sharing caches, or failing that, when they were on the same processor. The one exception was `hpt` where we got better results with two pairs of threads on each processor, with each pair sharing a cache. We believe this is because an `hpt` “quad-core” is in fact two dual-core dies that share a set of pins to the FSB, so moving threads to a separate processor doubles the effective memory bandwidth. Regardless, we always constrain the threads to adjacent cores, unless otherwise noted, since this most closely duplicates a machine with only the specified number of cores.



(a) centralized finalization (b) decentralized finalization

Fig. 7. Quench results.



(a) centralized finalization (b) decentralized finalization

Fig. 8. Anneal results: gnt only.

6. RESULTS

6.1 Overall Performance

The following results were collected using an internal build of Quartus II 10.0 on a set of 18 Stratix IV circuits. These circuits are a collection of IP and customer circuits used internally for product development at Altera, and range in size from approximately 20,000–300,000 cells. The averages shown are geometric averages unless otherwise noted.

The quench results are shown in Figure 7. During the quench, the centralized method achieved average speedups of up to 1.7x, 2.7x and 2.8x, on two, four, and eight processors, respectively, showing good performance at four processors and very poor scalability to eight. By contrast, the decentralized method achieved better performance and scalability with speedups of 1.8x, 2.9x, and 4.0x, respectively, showing good performance at four processors and moderate scalability to eight.

Decentralized finalization also outperformed the centralized method at all stages of the anneal (Figure 8), with the highest advantage occurring at the lower acceptance rates that characterize the quench. This is expected since the decentralized method was designed to work well when a large number of moves are being rejected.

As can be seen in Figure 8, the best annealing performance was achieved at four cores, while the best quench performance was achieved at eight. Therefore, our overall speedups were obtained by limiting ourselves to four processors during the anneal but switching to eight for the quench. Our best overall speedups are 1.5x, 2.1x, and 2.4x on two, four, and eight processors, respectively, using the decentralized finalization method. Note that these speedups are only for the *moves* performed during

Table IV. Per-Circuit Speedups (total, anneal, and quench) on `gnt` Using Decentralized Finalization

		two cores			four cores			eight cores		
name	kCells	t	a	q	t	a	q	t	a	q
circuit1	19.7	1.5	1.4	1.7	2.2	1.7	3.0	2.1	1.4	4.2
circuit2	28.7	1.6	1.4	1.7	2.5	1.8	3.2	2.7	1.5	5.4
circuit3	35.3	1.4	1.3	1.6	1.8	1.5	2.6	1.6	1.2	3.0
circuit4	42.1	1.4	1.3	1.6	2.0	1.6	2.9	1.8	1.2	3.9
circuit5	43.6	1.5	1.4	1.6	2.3	2.0	2.8	1.9	1.3	3.7
circuit6	58.2	1.3	1.2	1.6	1.6	1.3	2.6	1.4	1.1	2.7
circuit7	60.9	1.6	1.4	1.8	2.5	1.9	3.2	2.8	1.7	5.1
circuit8	68.6	1.4	1.2	1.7	2.0	1.5	3.1	1.9	1.3	4.6
circuit9	73.2	1.5	1.3	1.7	2.2	1.7	3.0	2.0	1.4	4.4
circuit10	110.2	1.3	1.2	1.6	1.6	1.4	2.4	1.5	1.2	2.7
circuit11	112.3	1.5	1.3	1.7	2.3	1.7	3.1	2.5	1.5	4.5
circuit12	120.4	1.3	1.3	1.6	1.9	1.7	2.7	1.7	1.4	3.4
circuit13	129.4	1.5	1.3	1.6	2.2	1.8	2.4	2.2	1.5	2.6
circuit14	151.9	1.4	1.3	1.7	2.1	1.9	3.1	1.9	1.6	4.3
circuit15	190.6	1.6	1.3	1.9	2.8	1.9	3.6	3.7	2.0	5.9
circuit16	192.2	1.4	1.2	1.7	2.1	1.7	3.0	2.2	1.5	4.6
circuit17	199.1	1.4	1.2	1.7	2.4	2.1	3.1	2.4	1.8	4.5
circuit18	300.4	1.4	1.2	1.7	2.4	1.9	2.9	2.6	1.7	4.1
geomean		1.4	1.3	1.7	2.1	1.7	2.9	2.1	1.4	4.0

simulated annealing, as is reported in most prior work; our overall placement speedups are slightly lower due to some compile time being spent in less frequent operations such as timing analysis, not all of which have been parallelized.

Per-circuit results are also given for the decentralized method on `gnt` in Table IV. Quench speedups at eight cores ranged from a low of 2.6x to a high of 5.9x. There was no strong relationship between circuit size and speedup.

The discrepancies in performance between `hpt`, `opt`, and `gnt` are explained by their differing memory subsystems. `opt` and `gnt` use dedicated point-to-point links (HyperTransport and QuickPath Interconnect, respectively) for intercore communication, in addition to dedicated on-chip memory controllers. On the other hand, `hpt` must use the same front-side bus for both memory bandwidth and intercore communication. In addition, `gnt` has a much higher bandwidth relative to clock speed than does `hpt`.

The only tuneable parameter is the number of moves that can be in flight at any one time. We found that reducing this number below four per core significantly hurt performance using both centralized and decentralized finalization, as the cores were forced to stall until the memory associated with the moves was released. Under some circumstances, we did find a modest improvement in performance by raising the number of moves in flight during the quench. By contrast, during the anneal, this reduced performance due to an increase in move collisions; that is, it was better for the cores to stall than to occupy themselves with work that was likely to be rejected by the dependency checker. Even during the quench, increasing the number of moves in flight too high eventually resulted in a drop in performance. We found that a limit in the range of four to eight moves in flight per core was reasonable in practice.

6.2 Attributing overhead

Our decentralized speedup of 4.0x on eight processors on `gnt` implies a 50% reduction in performance, or an overhead of roughly 100% versus the ideal speedup. That is, we doubled the runtime before reducing it by eight through parallelization. It is useful to determine the cause of this overhead to suggest future improvements[8]. In particular, we wish to identify the importance of each of the following factors.

(1) *Algorithmic*. What is the overhead involved with ensuring deterministic results? What is the overhead caused by resolving collisions?
(2) *Infrastructure*. What is the intrinsic overhead of adding parallelism to the annealer, for example, from synchronization primitives?
(3) *Hardware*. Are there any fundamental limitations of the computer itself which make it impossible to achieve a perfect speedup?

In Ludwin et al. [2008], we describe a method of attribution which worked well for a small number of processors. However, as the number of cores increases and the algorithms becomes more complex, we found that this method did not produce interesting results. Instead, we found it useful to investigate the sources of overhead by removing features from our algorithm. We show the results of this procedure shortly on a representative circuit (circuit14 in Table IV) which exhibited a speedup of 4.2x on `gnt` during the quench, or an overhead of 92%. The experiments were performed only during the quench, since (as described next) the experiments were significantly easier to perform. However, in our experience, we have found that the conclusions derived from the quench apply relatively well to the rest of the anneal.

6.2.1 Algorithmic Overhead. As described in Section 4.2, we maintain determinism by reprocessing moves that were invalidated. However, during the quench, the vast majority of moves are rejected, and reprocessing rarely changes this decision. By simply discarding moves that are speculatively rejected, we can investigate the effect of giving up determinism with minimal impact on the quality of results.

When we perform this experiment on our test circuit, the speedup improves from 4.2x to 4.8x, which corresponds to a decrease in overhead from 92% to 67%, a reduction of 25 points. However, we also observed a 2.5% reduction in the number of moves that are actually accepted. Therefore, to maintain equivalent QoR, we must increase the base number of moves we perform by the same amount. This reduces the speedup from 4.8x to 4.7x (an overhead of 71%). In other words, sacrificing determinism in this way reduced our overhead by approximately 20 points.

What happens if we stop reprocessing moves completely, as in Kravitz and Rutenbar [1987]? That is, when one move is accepted, we discard all later moves that are already in flight, even if they had been speculatively accepted as well. This allows us to remove the dependency checker and its associated overhead, and also to ignore the relative ordering of the moves, asynchronously committing the first move that was accepted and rejecting all others. While this experiment does improve the speedup to 5.4x (an overhead of 49%, or a drop of 18 points), it also reduces the number of accepted moves by a full 20%[9], at the cost of a noticeable increase in wirelength and critical path delay. When we increase the number of moves performed to compensate, we find that the speedup is reduced to 4.4x, that is, less than we observed when only discarding the speculatively rejected moves (4.7x).

The reason for this result is that it is far faster to reprocess a move that was likely to be accepted than it is to start a new move from scratch. Therefore, we could treat

[8]During our development, a similar set of experiments actually provided the impetus to develop the decentralized finalization method.
[9]Recall that these experiments are being performed only during the quench. Applying this approach to the anneal would result in a much higher drop in the number of accepted moves.

this reprocessing (for speculatively accepted moves) as a performance enhancement. However, consider what would happen if our algorithm only proposed noncolliding moves so that the speculative accept/reject decision was always correct. In this scenario, reprocessing would become redundant, so the unadjusted speedup result of 5.4x would represent the true performance. Therefore, this experiment shows the impact of proposing moves with collisions that must be resolved.

This hypothetical, collision-free algorithm would have an overhead of 49%, whereas our actual deterministic algorithm has an overhead of 92%. This 43-point reduction represents the maximum improvement that might be achieved through the elimination or faster detection of collisions. From the results of our first experiment, we believe it is reasonable to attribute approximately 20 points of this overhead to our requirement to maintain determinism, and the remaining 20–25 points to the effect of collisions. However, the 20% overhead for determinism itself could be reduced through better move generation.

6.2.2 Other Sources of Overhead. As shown in Section 6.1, the performance of our algorithm is strongly correlated with the number of moves accepted. We have found it instructive to observe the performance of the algorithm when no moves at all are accepted; that is, we propose and evaluate the same number of moves, but reject them regardless of the result. Of course, the resulting algorithm is no longer of any use, but we have found that when applied only in the quench, it does not significantly change the distribution of moves that are performed, and therefore is a reasonable way to investigate the algorithm's overhead.

Rejecting every move results in a speedup of 5.6x, relative to a serial algorithm that also rejects every move. This corresponds to an overhead of 44%, or a reduction 5 points. This represents an estimate of the amount of quality that can be sacrificed for increased performance during the quench, for example, by not accepting moves that showed only a small improvement in quality.

Now, since state never changes, we remove the per-core copies of the placement database and allow only one move per core, to examine the impact that the multiple copies have on processor cache and memory bandwidth usage. This improves performance to 5.8x, reducing overhead 6 points to 38%. Since the algorithm itself is unchanged by this step, all of this impact is due to improved cache efficiency and decreased bandwidth needs.

Finally, we can statically assign sets of moves to each core, since they are now by definition noncolliding. This allows us to remove the last forms of intercore communication: the message passing system as well as the shared move counter. The algorithm is now reduced to a set of simple `for` loops, processing and immediately discarding moves. This improves performance to 6.2x, reducing the overhead to 29%, a drop of 9 points. We attribute this 9% overhead to the algorithm's infrastructure.

At this point, the cores are no longer communicating in any way. It is therefore most likely that the remaining overhead of 29% is due to the inability of the hardware to supply the cores with enough memory bandwidth and cache[10].

The summary of our findings for this one circuit on both eight-core platforms is shown in Table V, with all figures rounded to the nearest 5%. While there is some discrepancy between the two platforms, they both show similar patterns.

— Both show the overhead of determinism to be relatively small, with an upper limit of 15%–20% of the serial algorithm.

[10]Some processors can throttle their clock speeds when multiple cores are in use, to reduce power consumption and heat generation. However, we did not observe this effect.

Table V. Approximate Overhead at Eight Cores (one circuit)

Factor	hpt	gnt
Determinism (upper limit)	15%	20%
Collisions (lower limit)	15%	25%
Quality	0%	5%
Increased memory	15%	5%
Infrastructure	20%	10%
Inherent memory limitations	60%	30%
Total	125%	90%

— To achieve a result of similar quality to our current algorithm, resolving collisions costs an overhead of approximately 15%–25% by forcing us to reprocess moves that had been speculatively accepted.
— Inherent memory limitations are a significant factor, with an overhead of 30% on the more modern `gnt` architecture and 60% on the older `hpt`. When combined with the increased memory required for thread safety, memory limitations account for an overhead of 35% and 75%, respectively, on these two architectures.

The implications of these results are discussed in our conclusion.

7. CONCLUSIONS AND FUTURE WORK

We have described a complex parallel simulated annealing algorithm that achieves good performance on multiple objectives, while still maintaining determinism and serial equivalency. We used two methods to maintain serial equivalency, and showed that the more decentralized algorithm achieved significantly higher performance, scaling well to four cores and moderately well to eight cores.

The Quartus II placement algorithm has had many tens of person years invested in it to achieve high QoR in a reasonable CPU time. Despite its complexity, we were able to parallelize the algorithm without simplifying it. Interestingly, while we have not made any changes to try to minimize the interaction between moves, we have found large amounts of parallelism to exploit, even at eight cores.

In addition, we have quantified the overhead required to maintain serial equivalency, communicate state updates between cores, and maintain thread safety. At eight cores, we found that maintaining determinism added an overhead of at most 15–20% relative to the unmodified serial algorithm. The cost of recovering from collisions was somewhat larger with at least 15–25% overhead, and the limitations of our computers' memory subsystems cost between 30–60%, with modern computers showing lower overhead.

It is likely that hardware vendors will continue to improve their multicore memory bandwidth. But it is also clear that unequal memory access will become increasingly prominent as we progress to sixteen and more cores per system, due to both shared caches and NUMA effects. To take advantage of emerging platforms, developers will therefore need to pay increasing attention to the memory access patterns of their algorithms.

These results point towards several potential improvements. For example, one could create a more sophisticated dependency checker that required less intercore communication, perhaps by maintaining per-core copies of the checker and only transmitting updates between cores. This would reduce the cost of maintaining determinism, but would not improve collisions or other memory limitations.

A far more productive approach will likely be create new directed moves that can take advantage of spatial locality. Most previous work that used strictly partitioned placements found some degradation in results from this approach, even after allowing for occasional cross-partition optimizations. The only exception [Sun and Sechen 1997] required that the partitions be frequently moved to allow cells sufficient mobility. As we described in Section 2.2.2, their exact approach is not appropriate for our algorithm, but we believe the general concept of creating additional spatial locality may be adapted. By biasing different cores to work on different parts of the placement, instead of strictly enforcing a partitioning, we expect we can greatly reduce the number of collisions while still allowing optimizations that span the entire chip. This would also lead to better cache and NUMA memory locality, and so would allow us to simultaneously tackle all three major sources of overhead.

The precise method by which we bias the directed moves will likely depend on the number of cores in use by the algorithm. We may therefore choose to give up serial equivalency so as to optimize performance on any given number of cores. Alternatively, we could optimize the algorithm for the *specific* number of cores that are most commonly available. The trade-off between serial equivalency and performance will likely change over time as the number of cores increases.

Better directed moves may also open new opportunities to improve other areas of the algorithm. For example, a much simpler dependency checker might suffice to rule out collisions between a majority of moves, reducing not only the work involved with reprocessing moves, but the overhead associated with tracking their interactions. This would further reduce intercore memory traffic. Many such interrelated improvements will likely be needed to ensure that high-quality parallel placers continue to scale to sixteen cores and beyond.

REFERENCES

AARTS, E., DEBONT, F., AND HABERS, E. 1986. Parallel implementations of the statistical cooling algorithm. *Integr. VLSI J. 4* 3, 209–238.

ALPERT, C., HAGEN, L., AND KAHNG, A. 1996. A hybrid multilevel/genetic approach for circuit partitioning. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems*. 298–301.

ALTERA. 2010. Quartus II incremental compilation for hierarchical and team-based design. In *The Quartus II Handbook*, Version 10.0. Vol. 1, Chapter 2.

BANERJEE, P., JONES, M., AND SARGENT, J. 1990. Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors. *IEEE Trans. Parall. Distrib. Syst. 1*, 91–106.

BETZ, V. 2007. Placement for general purpose FPGAs. In *Reconfigurable Computing*, A. DeHon and S. Hauck Eds. Morgan Kauffman, Chapter 14, 299–317.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Boston, MA.

BIAN, H., LING, A., CHOONG, A., AND ZHU, J. 2010. Towards scalable placement for FPGAs. In *Proceedings of the ACM/SIGDA 18th International Symposium on Field Programmable Gate Arrays*.

BORRA, S. N. R., MUTHUKARUPPAN, A., SURESH, S., AND KAMAKOTI, V. 2003. A parallel genetic approach to the placement problem for field programmable gate arrays. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 184.

CALDWELL, A., KAHNG, A., AND I.L.MARKOV. 1999. Optimal partitioners and end-case placers for standard-cell layout. In *Proceedings of the International Symposium on Physical Design*. 90–96.

CHAN, P. AND SCHLAG, M. 2003. Parallel placement for field-programmable gate arrays. In *Proceedings of the ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays*. 33–42.

CHAN, T., CONG, J., KONG, T., AND SHINNERL, J. 2000. Multilevel optimization for large-scale circuit placement. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 171–176.

CHANDY, J. AND BANERJEE, P. 1996. Parallel simulated annealing strategies for VLSI cell placement. In *Proceedings of the 9th International Conference on VLSI Design*. 37–42.

CHANDY, J., KIM, S., RAMKUMAR, B., PARKES, S., AND BANERJEE, P. 1997. An evaluation of parallel simulated annealing strategies with application to standard cell placement. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. 16*, 4, 398–410.

CHEN, G. AND CONG, J. 2004. Simultaneous timing driven clustering and placement for FPGAs. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications*.

HALDAR, M., NAYAK, A., CHOUDHARY, A., AND BANERJEE, P. 2000. Parallel algorithms for FPGA placement. In *Proceedings of the 10th Great Lakes Symposium on VLSI*. 86–94.

HUTTON, M. AND BETZ, V. 2006. FPGA synthesis and physical design. In *Electronic Design Automation for Integrated Circuits Handbook*, L. Scheffer et al., Eds., Vol. 1., Taylor and Francis CRC Press, Chapter 13, 13.1–13.32.

KIM, S., CHANDY, J. A., PARKES, S., RAMKUMAR, B., AND BANERJEE, P. 1994. ProperPLACE: A portable parallel algorithm for standard cell placement. In *Proceedings of the 8th International Symposium on Parallel Processing*. 932–941.

KIRKPATRICK, S., GELATT JR., C., AND VECCHI, M. 1983. Optimization by simulated annealing. *Sci. 220*, 4598, 671–680.

KLEINHANS, J. M., SIGL, G., JOHANNES, F., AND ANTREICH, K. 1991. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 10*, 3, 356–365.

KNUTH, D. 1997. *Seminumerical Algorithms* 3rd Ed. Addison-Wesley, Reading, MA.

KRAVITZ, S. AND RUTENBAR, R. 1987. Placement by simulated annealing on a multiprocessor. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 534–549.

LUDWIN, A., BETZ, V., AND PADALIA, K. 2008. High-Quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays*.

LUU, J., KUON, I., JAMIESON, P., CAMPBELL, T., YE, A., FANG, W., AND ROSE, J. 2009. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceedings of the ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays*.

SARRAFZADEH, M., WANG, M., AND YANG, X. 2003. *Modern Placement Techniques*. Kluwer Academic Publishers, Boston, MA.

SECHEN, C. AND SANGIOVANNI-VINCENTELLI, A. 1985. The TimberWolf placement and routing package. *IEEE J. Solid-State Circ. 20*, 2, 510–522.

SUN, W. AND SECHEN, C. 1995. Efficient and effective placement for very large circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 14*, 3, 349–359.

SUN, W. AND SECHEN, C. 1997. A parallel standard cell placement algorithms. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 16*, 11, 1342–1357.

SUTTER, H. 2005. The free lunch is over. A fundamental turn toward concurrency in software. *Dr. Dobb's J. 30*, 3.

VISWANATHAN, N. AND CHU, C. 2004. FastPlace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. In *Proceedings of the International Symposium on Physical Design*.

VORWERK, K., KENNINGS, A., GREENE, J., AND CHEN, D. 2007. Improving annealing via directed moves. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications*. 363–370.

VORWERK, K., KENNINGS, A., AND GREENE, J. 2009. Improving simulated annealing-based FPGA placement with directed moves. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 28*, 2, 179–192.

WATSON, I., KIRKHAM, C., AND LUJÁN, M. 2007. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*.

WITTE, E., CHAMBERLAIN, R., AND FRANKLIN, M. 1991. Parallel simulated annealing using speculative computation. *IEEE Trans. Parall. Distrib. Syst. 2*, 4, 483–494.

WRIGHTON, M. AND DEHON, A. 2003. Hardware-Assisted simulated annealing with application for fast FPGA placement. In *Proceedings of the ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays*. 33–42.

Received March 2010; revised January 2011; accepted March 2011