Speeding Up FPGA Placement: Parallel Algorithms and Methods

Matthew An, J. Gregory Steffan, Vaughn Betz
Department of Electrical and Computer Engineering
University of Toronto, Ontario, Canada
{ansiyuan, steffan, vaughn}@eecg.toronto.edu

***Abstract*—Placement of a large FPGA design now commonly requires several hours, significantly hindering designer productivity. Furthermore, FPGA capacity is growing faster than CPU speed, which will further increase placement time unless new approaches are found. Multi-core processors are now ubiquitous, however, and some recent processors also have hardware support for transactional memory (TM), making parallelism an increasingly attractive approach for speeding up placement. We investigate methods to parallelize the simulated annealing placement algorithm in VPR, which is widely used in FPGA research. We explore both algorithmic changes and the use of different parallel programming paradigms and hardware, including TM, thread-level speculation (TLS) and lock-free techniques. We find that hardware TM enables large speedups (8.1x on average), but compromises "move fairness" and leads to an unacceptable quality loss. TLS scales poorly, with a maximum 2.2x speedup, but preserves quality. A new dependency checking parallel strategy achieves the best balance: the deterministic version achieves 5.9x speedup and no quality loss, while the non-deterministic, lock-free version can scale to a 34x speedup.**

Keywords*—*FPGA placement; parallel placement; simulated annealing; transactional memory

I. Introduction

Over the last 15 years, the size of FPGA devices has been growing at nearly four times the rate of single-core CPU performance [1]. As a result, CAD tools typically spend hours compiling large designs targeting modern FPGAs. For example, a recent study of 21 large FPGA benchmarks found that placement is the most time-consuming CAD step and comprised 49% of total compile time in Altera's Quartus II CAD system [2]. The largest design that would fit in a 40 nm FPGA required over 16 hours to place.

Quartus II uses simulated annealing (SA) placement [1], which is commonly used for FPGAs because it handles both legality constraints and non-linear delay functions well. Analytic placement is also used commercially [3], but it is usually paired with annealing for fine-tuning the result [4], [5]. Consequently, reducing the computation time of SA using readily available parallel hardware is an attractive option. However, SA makes a series of local improvement "decisions" to generate a high-quality placement, and as one decision impacts subsequent decisions, the algorithm is naturally sequential and parallelization is non-trivial.

There have been several recent efforts in parallelizing SA placement for FPGAs [1], [6], [7]. We evaluate new parallel approaches that build on these prior techniques, and also leverage new processor features such as transactional memory (TM) and thread-level speculation (TLS) that aim to make parallel programming easier. Our contributions include a quantitative comparison of the speedup and quality-of-results obtained with various parallel algorithmic and programming approaches. We find that while TM and TLS simplify parallel programming, neither can achieve a compelling combination of speedup and placement quality. Our best algorithms require more programming effort than TM or TLS, but outperform prior approaches: without loss of placement quality, we can reach 5.9x speedup with a deterministic algorithm and 34x speedup with a non-deterministic one.

This paper is organized as follows. We first detail the relevant prior work, and then outline the three broad parallel approaches we investigate. We then quantitatively compare the result quality and speedup achieved with each approach, and finally conclude.

II. Background

A. Simulated Annealing Placement

SA mimics the process of controlled cooling in metallurgy to produce high-quality objects [8]. The placer begins with a random placement of blocks on the FPGA and evaluate a large number of perturbations to the placement, called *moves*. In VPR [9], moves can be split into two phases. During *proposal*, a block and a destination are randomly chosen. During *evaluation*, the block is moved to its destination (if the destination already contains a block, the two blocks are swapped) and the change in placement cost (according to some cost function) is computed. Moves that decrease the cost of the overall placement are always accepted, while those that increase cost are still accepted with some probability to avoid being trapped in a local minimum of the cost function.

B. Prior Work in Parallel SA

A simple way to parallelize SA is to distribute the work within one move across multiple threads, but there is insufficient parallelism for this approach to scale well [10]. A more effective way is to distribute moves among multiple threads to be evaluated in parallel. However, conflicts can occur when two threads attempt to move the same block to different locations (Figure 1a), different blocks to the same location (Figure 1b), or blocks connected to the same net, which in some cases will cause both threads to incorrectly evaluate the cost function for this net (Figure 1c). To ensure that the final placement is valid, the parallel algorithm must either detect and resolve conflicts

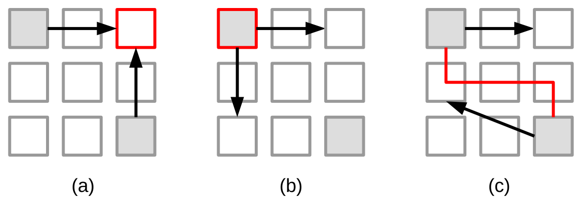


Fig. 1. Examples of move conflicts. Arrows represent moves being evaluated in parallel.

[1], [7], or prevent them by guaranteeing the independence of moves being evaluated in parallel [11], [6].

An early parallel implementation for standard cell placement is in the TimberWolf placement and routing package [11]. It partitions the chip to ensure that moves do not conflict. A related approach for FPGAs was developed in [6], where each thread generates moves in a local region distinct from those of all other threads, and also uses stale location data for the placement of blocks outside its region. Region boundaries move periodically to allow blocks to move outside their original regions. Threads also periodically broadcast updated placement locations to other threads to limit how "stale" their data can be. This algorithm scales very well (51x speedup over sequential VPR, with 16 threads) and is deterministic, but incurs a 10% quality loss.

The parallelized Quartus II placer (Q2P) speculatively evaluates moves in parallel and uses a manually coded *dependency checker* to detect conflicts [1]. Whenever possible, it resolves conflicts by providing speculative moves with updated information and repairing them. Q2P attains limited speedup (2.4x on 8 threads), but it is deterministic and maintains the same placement quality as the original sequential algorithm. The conflict detection and resolution components are more difficult to code than the partitioning-based approaches above.

Transactional VPR (TVPR) also evaluates parallel moves speculatively, and leverages software TM to automatically detect and resolve conflicts [7]. TVPR scales better than Q2P (self-relative speedup of 7.3x on 8 threads), but suffers from excessive TM overhead and only attains real speedup for some benchmarks, averaging about 0.9x across all tested benchmarks. The average quality loss of 1% is negligible. The use of TM makes TVPR non-deterministic, but easier to code than all of the placers mentioned above.

C. Transactional Memory

TM provides atomicity, consistency, and isolation for arbitrary sections of code accessing shared memory, making them behave like database transactions [12]. TM makes it possible to parallelize a program by dividing it into tasks, assigning the tasks to all available threads, and executing each task as a transaction. Since the independence of tasks is not known ahead of time, the TM system must monitor all memory accesses made by transactions and ensure that transactions which conflict are aborted and rolled back to preserve correctness. A conflict occurs when multiple transactions access the same memory location, with at least one transaction writing to it. If a transaction completes with no conflict, it can commit all its changes to main memory and make them visible to other threads.

Software TM (STM) implements tracking, conflict detection, and rollback entirely in code. To apply STM to a program, the compiler or programmer inserts extra instructions to track memory accesses made inside transactions. Hardware TM (HTM) relies on extensions to the memory subsystem to detect conflicts and perform rollbacks. Automatic tracking by the hardware provides two major advantages: tracking instructions become unnecessary and the programmer only needs to annotate transaction boundaries; and the overhead associated with tracking becomes minimal. The authors of TVPR observed that TVPR should attain much higher speedups using HTM [7]. However, unlike STM, hardware has limited capacity and hence the amount of data that can be tracked by any HTM system is limited. Transactions that reach this limit must be aborted and retried non-transactionally, which prevents parallel execution and negatively impacts performance.

Some of the latest multi-core CPUs, in particular IBM's Blue Gene/Q [13] and Intel's Haswell [14], now incorporate HTM support, motivating investigation into the utility of HTM for parallelizing CAD algorithms.

III. Parallel Placers

We present three different parallel SA algorithms, some of which are coded with multiple techniques. For algorithms with several implementations, we name them as *Algorithm-Technique*, e.g. MoveSpec-STM, which is Move Speculation implemented using software transactional memory.

A. Move Speculation

This is the approach taken by TVPR. Entire moves are made speculative by enclosing them in transactions, and the TM system automatically detects conflicts between them. On a conflict, one of the conflicting moves is aborted and rolled back by the TM system, so the other move still has a consistent view of global state. If a move successfully commits, any changes it makes to the placement will be valid. Figure 2 shows an example:

1) Thread 1 starts transaction T1 and proposes to move block A, while thread 2 starts transaction T2 and proposes to move block B. Neither transaction can see the conflict with connection AC because they have not written anything to memory yet.
2) T1 moves A to its destination and evaluates new costs for connections AB and AC. At the same time, T2 moves B to its destination. Now the conflict is detected by the TM system because T1 is reading the location of B while T2 is writing to it. Either T1 or T2 must abort, and the programmer cannot specify which one, so the behavior of Move Speculation is fundamentally non-deterministic.
3) If T1 aborts:
    a) A is restored to its original location so that T2 can correctly compute the new cost of connection AB. Cost computations done by T1 are discarded. These include the cost of connection AC, even though it was not involved in the conflict.
    b) T1 is re-executed with a new move proposal.
4) If T2 aborts:
    a) B is restored to its original location so that T1 can correctly compute the new cost of connection AB.

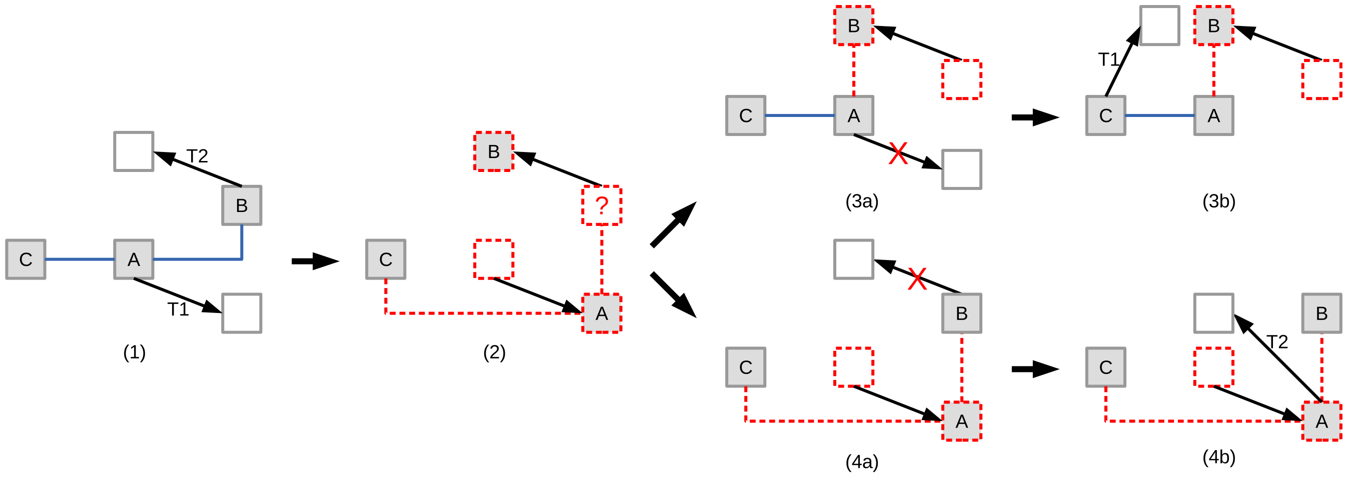


Fig. 2. Illustration of Move Speculation with two threads. Objects in red (dashed lines) have been modified by transactions and changes to them have not been committed to memory.

b) T2 is re-executed. Since each thread uses a separate random number generator to propose moves, the new move for T2 is not the same as the one for T1.

While it is possible to re-propose the original unsuccessful move, the placer could get stuck in an infinite loop of rolling back and retrying two transactions that repeatedly conflict with each other. The solution in TVPR is to always *abandon* the unsuccessful move and propose a new move when a transaction is retried. This causes the entire set of moves to change depending on when and where conflicts occur, and the result is *favoritism* (called *swap favoritism* by the TVPR authors): the placer favors shorter, simpler moves that access less memory and have a lower probability of encountering conflicts, and larger moves are frequently rolled back and never attempted again.

1) MoveSpec-STM: MoveSpec-STM is our implementation of Move Speculation by applying STM to VPR 6. MoveSpec-STM uses the same programming techniques as TVPR: throughout the placement code, we annotate transaction boundaries and all shared memory accesses within the transaction. This is necessary because STM only detects conflicts between annotated memory accesses. Annotations for TVPR were also added manually due to limited compiler support when TVPR was being developed. Searching through all functions called during move evaluation and annotating them takes a significant amount of programming effort. However, it provides opportunities to reduce the annotations to a minimum and hence lower the overhead associated with STM tracking. In MoveSpec-STM, we only annotate accesses that are essential for conflict detection (which means read-only accesses are left out) and identify data structures being used as scratch space, which can also be left out and instead replicated for each thread (privatized) to eliminate false conflicts.

An important optimization from TVPR, which we also apply to all of our parallel placers, is *ignoring big nets*. This is based on the observation that high fan-out nets tend to have very little impact on placement quality because they usually cannot be localized well and hence do not cause significant cost changes. By ignoring cost computations for these nets, the parallel placer can avoid conflicts associated with them and reduce the frequency of transaction rollbacks. TVPR uses a size threshold of 10%, which means that nets containing more than 10% of all the blocks in the design are ignored.

2) MoveSpec-HTM: MoveSpec-HTM is our implementation of Move Speculation using HTM. As discussed in Section II-C, the key advantages of using HTM are ease of programming and potentially better performance than STM. Since we cannot instruct the hardware to leave out certain memory locations for tracking, we cannot manually apply annotations to further reduce tracking overhead.

3) MoveSpec-TLS: Thread-level speculation (TLS) can be viewed as an ordered (hence deterministic) version of TM. The transaction being executed by the currently oldest thread is guaranteed to commit; and before younger threads are allowed to commit, they must wait for older threads to commit first. As a result, the commit order of transactions is always the same, regardless of the number of threads used. It also becomes impossible to abandon moves on conflict, so TLS can only exploit parallelism between consecutively proposed moves, not actively find parallelism the way MoveSpec with TM does. Our implementation of Move Speculation with TLS is called MoveSpec-TLS, and the only programming difference from MoveSpec-HTM is the type of annotation for speculative code.

B. Proposal Speculation

Abandoning moves that conflict is effective at finding parallelism, but wastes work done up to the point of detecting the conflict. We improve on this method by making conflict detection as early as possible. During move proposal, the placer identifies all dependencies of this move—blocks, grid locations, and nets this move will affect—and tags them as "in-use" or "reserved". The placer encounters a conflict if it tries to tag a dependency that has already been tagged. When a conflict occurs, the current move proposal is abandoned by untagging anything tagged so far. On the other hand, if tagging is successful, the placer can safely evaluate the proposed move concurrently with other moves in progress, because all dependencies for this move (and others in progress) have already been exclusively reserved. Conflicts during evaluation are effectively prevented. At the end of move evaluation, the

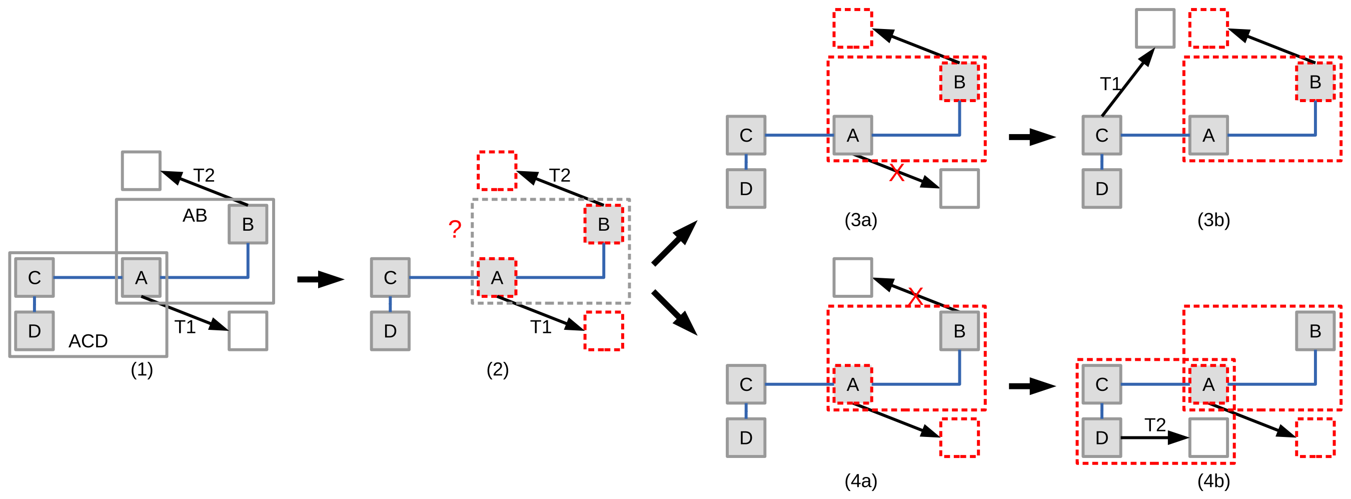


Fig. 3. Illustration of Proposal Speculation with two threads. Objects in red (dashed lines) are tagged and the tagging can be done directly in memory.

dependencies are untagged. We call this algorithm Proposal Speculation, or PropSpec for short. Figure 3 shows an example:

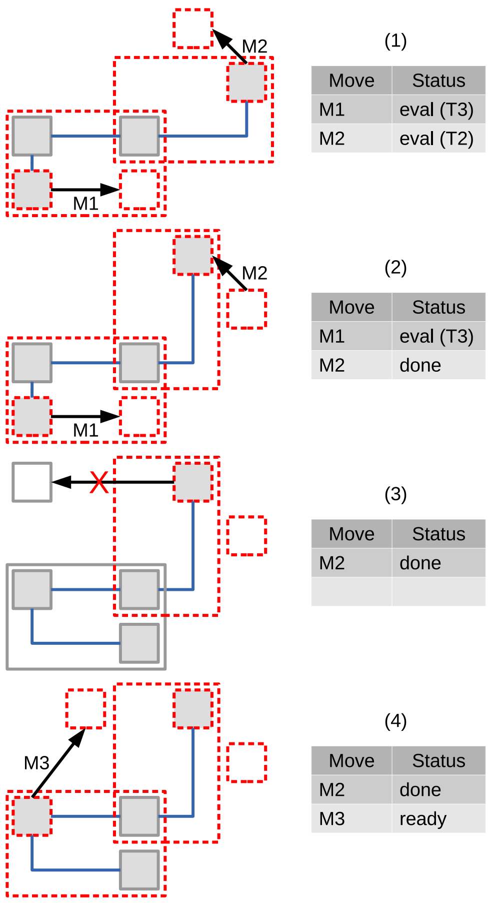
1) Net AB connects blocks A and B, while net ACD connects blocks A and D to C. Thread 1 (T1) proposes to move A, while thread 2 (T2) proposes to move B.
2) T1 successfully tags A and its destination, then tries to tag all nets connected to A (AB and ACD). T2 successfully tags B and its destination, then tries to tag AB. Execution then depends on which thread tags AB first.
3) If T2 tags AB first:
    a) T1 will be unable to tag AB. T1 then abandons its move and untags A and its destination. If T1 has tagged ACD already, it will also untag ACD.
    b) T2 proceeds to evaluate its move while T1 proposes a new move.
4) If T1 tags AB first:
    a) T2 will be forced to abandon its move and untag A and its destination.
    b) T1 tags ACD and proceeds to evaluate its move. Meanwhile, T2 proposes a new move. T2 will conflict with T1 again because D is part of ACD, even though the two moves can safely execute in parallel. (T2 will not affect any bounding box or connection costs that T1 will compute.) Note that because conflicting proposals get abandoned, PropSpec can be subject to favoritism.

Step 3b shows that the actual dependencies of a move may be specific (not all) terminals of a net, so conflict detection using entire nets is more coarse-grained and conservative, but much more efficient to tag and check given the fan-out of typical nets. Our dependency checking scheme is similar to the dependency checker from Q2P [1] but is more conservative and applied at the beginning instead of the end of move evaluation. As with Q2P, the programmer must correctly identify all dependencies for this scheme to work. Failing to tag a dependency allows another thread to potentially modify its associated data and render the proposed move illegal or improperly costed. Tagging and untagging actually take less coding than STM annotations, since the logic to determine dependencies is straightforward, consisting mainly of traversals through lists.

We implement PropSpec two ways. Since there is fine-grained parallelism between proposals, TM provides an easy way to exploit it. In PropSpec-HTM, we put the move proposal and tagging into a transaction and use HTM to detect tagging conflicts. (Note that we cannot use TLS because only move proposal is speculative but evaluation is not, and a speculative thread cannot switch to non-speculative mode before committing the entire evaluation.) In PropSpec-LF, we exploit the parallelism manually using lock-free techniques [15]: tagging is done using compare-and-swap operations, with failed operations indicating conflicts. Because we cannot leverage the TM rollback mechanism to resolve conflicts here, we manually write rollback code to untag any dependencies that have been tagged so far. Consequently, PropSpec-LF takes the most programming effort among all the algorithms we present.

C. *Serialized Proposals*

Like MoveSpec, PropSpec is fundamentally non-deterministic because it does not enforce any ordering between threads. Non-determinism is an undesirable characteristic for FPGA CAD tools because it makes debugging difficult, and is unacceptable for designers such as high-security FPGA users, who expect the same results every time they use the tool. To make PropSpec deterministic, we simply serialize the move proposals. We designate one master thread to do all the proposing, including tagging, conflict detection, and untagging. All other available threads are designated as workers and only perform move evaluations. Like Q2P, we assign each move a sequential ID. Proposed moves are inserted into a task queue of length L. Since these moves are independent, they can be evaluated by any worker in any order. Initially, moves are inserted until the queue is full. To maintain determinism, workers do not untag dependencies after evaluating a move, and finished moves only leave the queue when they reach its front. When a finished move leaves the queue, its dependencies are untagged and a new move must be inserted to keep the queue full. The constraints above ensure that before every proposal, the master always sees



(1)

Move	Status
M1	eval (T3)
M2	eval (T2)

(2)

Move	Status
M1	eval (T3)
M2	done

(3)

Move	Status
M2	done

(4)

Move	Status
M2	done
M3	ready

Fig. 4. Illustration of Serialized Proposals with two workers and a task queue of length 2. Each diagram shows tagged dependencies (dashed lines) along with the corresponding moves in the queue.

the dependencies of the previous $L-1$ moves tagged, and everything else untagged. Consequently, the placement state before every proposal only depends on L and not the number of threads, so the algorithm is deterministic. We call this algorithm Serialized Proposals, or SerialProp for short. Figure 4 shows an example with $L=2$:

1) The master proposes moves M1 and M2, tags their dependencies, and inserts them into the queue. It cannot propose another move because the queue is full. Workers T2 and T3 each choose an arbitrary move from the queue to evaluate.
2) M2 is accepted first. Since M2 is not at the front of queue, it cannot leave and its dependencies are still tagged.
3) M1 is accepted. It leaves the queue and its dependencies are untagged. M2 reaches the front but is not allowed to leave early because there is space in the queue. Now the master tries to propose M3, but encounters a conflict.
4) M3 is re-proposed successfully, tagged, and inserted. Now M2 can leave because the queue has filled up. Note that the final proposal of M3 would be the same if M1 had been accepted first.

Using the dependency checking infrastructure from PropSpec, the implementation of SerialProp is straightforward. The task queue is easily specified using OpenMP [16], and the only communication between threads that must be explicitly coded is for workers to notify the master that they have finished evaluating a particular move.

TABLE I. QUALITATIVE COMPARISON OF OUR PARALLEL PLACERS.

Name	Deterministic?	Programming Effort
MoveSpec-STM	No	Medium
MoveSpec-HTM	No	Low
MoveSpec-TLS	Yes	Low
PropSpec-HTM	No	High
PropSpec-LF	No	Highest
SerialProp	Yes	High

Table I shows the key characteristics of our parallel placer implementations.

IV. Methodology

We implemented all of our parallel placers by modifying the sequential placer in VPR 6. For MoveSpec-STM, we used TinySTM 1.0.4 [17], a newer version of the STM used in TVPR. We conducted all experiments on an IBM Blue Gene/Q (BG/Q) compute node containing 16 cores. Each core is threaded 4 ways for a total of 64 hardware threads. The BG/Q provides hardware support for both TM and TLS, and the IBM XL compiler makes it available to the programmer [13]. Following [18], we compiled all of our placers with the `-O3` and `-qhot` optimization flags. We ran all placers with default placement options, but reduced the level of effort by setting `inner_num` to 1. (This is the default for VPR 7.) For performance results, we measured the wall-clock time of placement only and excluded time taken by timing analysis. For quality of results, we used the placement estimates of wire length and critical path delay provided by VPR. All results were averaged over the 8 largest VTR benchmarks [19]. We used VPR to pack them for an FPGA architecture with 6-input LUTs and 10 LUTs per logic block, and the packed circuit sizes range from 1142 to 7761 blocks. All available heterogeneous block types (RAM, DSP, etc.) are used by at least one of the benchmarks.

V. Results

Table II summarizes the speedup and quality-of-results for all of our parallel placers at their optimal number of threads. Speedup and quality metrics are calculated relative to sequential VPR 6. Some of the placers scale beyond the number of threads given, but they lose too much quality with more threads. We allow a maximum average increase of 20% in either wire length or critical path delay.

TABLE II. PERFORMANCE AND QUALITY RESULTS FOR EACH PLACER. ONLY PLACERS MARKED WITH * ARE DETERMINISTIC.

Name	Threads	Speedup	Wire Length	Critical Path Delay
MoveSpec-STM	16	3.6	+8.5%	+2.3%
MoveSpec-HTM	4	8.1	+19.0%	+1.7%
MoveSpec-TLS*	5	1.7	-0.1%	+0.2%
PropSpec-HTM	64	17.2	+0.8%	+0.7%
PropSpec-LF	64	33.9	+1.7%	+0.6%
SerialProp*	9	5.9	+0.0%	+0.1%

A. *Move Speculation*

Figure 5 shows the quality of placement obtained with MoveSpec-STM and MoveSpec-HTM. The run time for each is varied by changing the thread count from 2 to 64; quality varies with thread count due to move favoritism (Section III-A). Figure 5 also shows the quality vs. run time for sequential VPR in two modes: annealing and quench ($T = 0$). Run time for sequential VPR is varied by changing the placement effort level via `inner_num` [9]. Note that for very low run times, quenching results in a higher quality placement than annealing because there are too few moves for effective hill-climbing. Quality and run time values are normalized to those from sequential VPR in annealing mode with `inner_num` $= 1$.

1) MoveSpec-STM: The results for MoveSpec-STM are plotted as the STM quality curves in Figure 5. At 8 threads, we obtain a 1.7x speedup at a small quality loss of 1.6% wire length increase and 0.2% delay increase. This outperforms TVPR, which also uses STM and achieves a 0.9x speedup at 8 threads. Even though MoveSpec-STM tracks a minimal amount of data via careful annotations, the ratio of speedup to thread count is still low due to STM overhead. At 16 threads (Table II), the speedup improves to 3.6x but incurs a significant quality loss of 8.7% wire length and 2.3% delay. Speedup further increases to 11x at 64 threads but the quality loss becomes unacceptable. Due to move favoritism (Section III-A), the quality losses from using more threads are worse compared to sequential VPR at lower effort levels. In other words, running MoveSpec-STM on multiple threads always produces a *lower quality* placement than running VPR on one thread for the same amount of time. This is shown in Figure 5 by the STM quality curves being above the sequential (and quench) ones.

2) MoveSpec-HTM: We expected MoveSpec-HTM to behave similarly to MoveSpec-STM, albeit with much lower overhead. Unfortunately, this is not the case due to transactional capacity limits imposed by the hardware (Section II-C). The BG/Q only monitors transactions in its L2 cache, so a transaction with a memory footprint that does not fit in the cache will abort. Furthermore, since the cache is divided into non-overlapping sets (meaning that when one set is full, it must start evicting data; it cannot make room by transferring its data to another set), a transaction that overflows *any* set will be aborted. We suspect that even if most transactions do not conflict and would normally fit in the cache, they start to abort when many are executed concurrently because the cache sets cannot accommodate all of them together.

The net result for MoveSpec-HTM appears to be extreme favoritism, because only moves that do not overflow the cache can successfully evaluate. Effectively, the algorithm is unable to consider moves requiring a large amount of memory, which usually are the ones having the most positive impact on quality. A disproportionate number of small moves leads to super-linear speedup (8x on 4 threads), since the algorithm does less work to finish the annealing process. Accordingly, the 19% wire length increase at 4 threads is much worse than the 8.5% increase for MoveSpec-STM at 16 threads (Table II). Running MoveSpec-HTM with more threads only exacerbates the favoritism, as capacity limits increase the bias toward lower quality. As shown in Figure 5, the HTM curves rise more sharply than the STM ones, but all of their shapes are similar. In both cases, the quality curves are above the sequential ones, indicating that sequential VPR offers a better quality/run time trade-off.

3) MoveSpec-TLS: According to Table II, MoveSpec-TLS is the least scalable of our algorithms (1.7x speedup on 5 threads). However, it always gives results of the same quality regardless of thread count, and achieves an average quality equal to that of sequential VPR. Generally speaking, larger circuits have better performance than smaller ones. This is consistent with the observation from TVPR that parallel moves for large circuits are less likely to conflict [7].

The limited scaling of MoveSpec-TLS is due to its inability to actively search for parallel moves, since moves must complete in order. Stalling and wasted work caused by rollbacks also contribute to overhead. When placing the circuit with the best speedup among all tested benchmarks (2.2x), 5 threads only spend 57% of their time doing useful work. Overall performance is also reduced by HTM overhead, since TLS in the BG/Q is implemented on top of its HTM support. Our results broadly agree with the (simulated) 2x speedup on 4 threads obtained in [20] by manually applying TLS to a much older (SPEC 2000) version of VPR.

B. Proposal Speculation

According to Table II, this algorithm shows the best scaling among all of our algorithms, in addition to negligible losses in quality. At 64 threads, the lock-free implementation outperforms HTM, attaining a 34x speedup with a 1.7% wire length increase and 0.6% delay increase. However, both implementations of PropSpec scale equally well up to 16 threads. We suspect that the BG/Q HTM performance stops improving beyond 16 threads because the hardware only handles transaction retries at a limited rate, and is unable to keep up when many transactions (proposals) are frequently conflicting and aborting [18].

Despite the similarity of PropSpec to MoveSpec, its impact on quality is much less problematic. In Figure 6, the quality curves for both PropSpec implementations are nearly flat, which means that quality changes very little with more threads. The amount of shared data accessed during dependency checking in PropSpec is mainly determined by the number of nets affected, which varies much less across different moves than the number of block locations and connection cost changes that must be accessed transactionally in MoveSpec. Thus, the difference in memory footprint between small move proposals and large ones is not enough to cause much favoritism toward small moves, especially as the circuit grows in size and exhibits improved locality.

C. Serialized Proposals

As Table II shows, this algorithm preserves both quality and determinism while scaling reasonably well (5.9x speedup). Like MoveSpec-TLS, SerialProp always gives the same results regardless of thread count, and also achieves an average quality equal to that of sequential VPR. Eventually, move proposal becomes the bottleneck as it is all done by one thread, so adding workers only increases the proportion of time they spend waiting for successful proposals. For some circuits, moves tend to be quick to evaluate and this scaling limit is reached at 5 workers. The maximum number of useful workers is about 10, since none of the benchmarks have an evaluation-to-proposal time ratio greater than 10. Figure 7 shows the

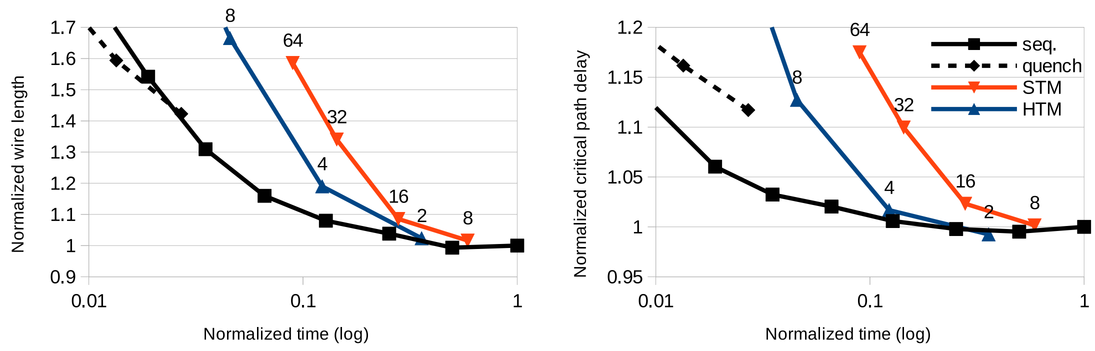


Fig. 5. Placement quality vs. time for Move Speculation. Data labels indicate thread count.

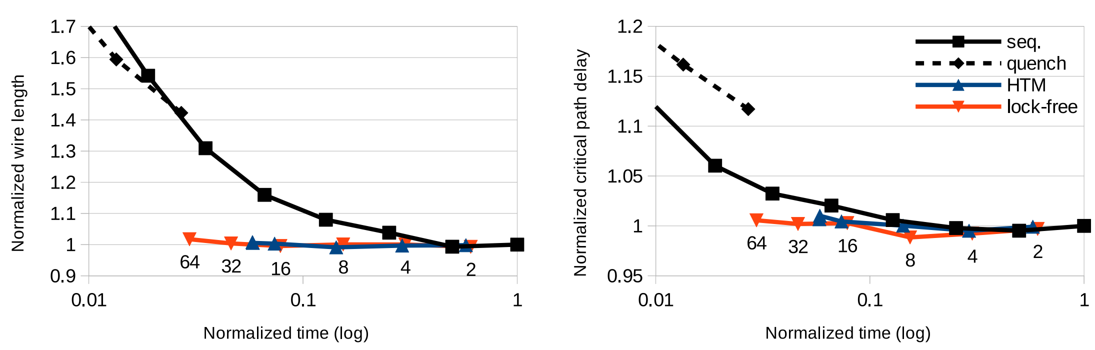


Fig. 6. Placement quality vs. time for Proposal Speculation. Data labels indicate thread count.

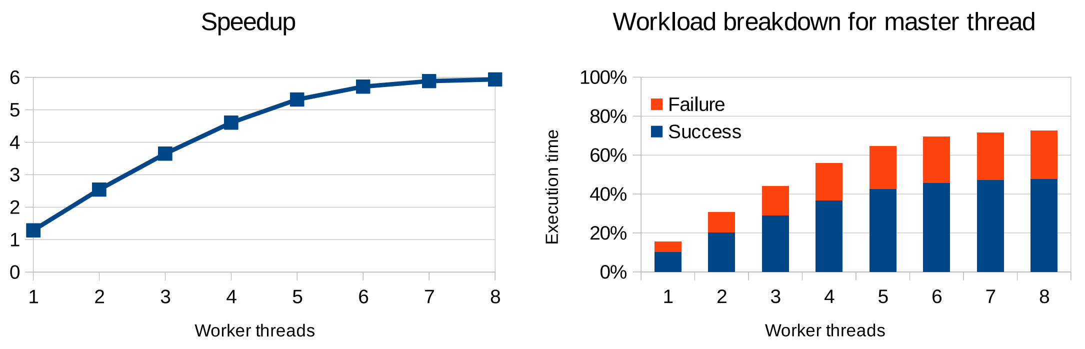


Fig. 7. Scaling behavior of Serialized Proposals. In the breakdown, "Success" means work during successful proposals, and "Failure" means work during proposals that were ultimately abandoned. The master thread spends all remaining time waiting for workers to finish move evaluation.

correlation between overall speedup and load on the master thread. While the ratio of useful work (successful proposals) to wasted work (abandoned proposals) remains fairly consistent, the increase in work done by the master diminishes with each additional worker. This is because waiting for move evaluation is unavoidable: moves in the task queue can be evaluated in any order and not all of them will finish in the same amount of time, but the waiting is done in a fixed order. The final result is that beyond 3 workers, each additional worker contributes less performance gain.

We set the task queue length L to 12 for our experiments, but it should be possible to fine-tune this parameter in order to trade quality for performance. A shorter queue decreases the probability of conflict and hence reduces time spent on failed proposals. It also discourages favoritism, but the queue would empty out more frequently and force workers to wait longer. A longer queue keeps workers busy, but encourages favoritism and makes proposal a bottleneck at fewer threads.

VI. Conclusion

We have found that speculatively evaluating moves in parallel is an effective strategy for parallelizing SA. When conflict tracking is applied to the entire move (MoveSpec), we obtain two extremes: TM scales well but has too much favoritism and quality loss, while TLS scales poorly due to too many rollbacks. While TM and TLS certainly simplify the parallelization of placement, their use results in problematic behavior. This observation may hold true for other decision-based CAD algorithms as well.

By tracking conflicts only during proposal, our dependency checking scheme greatly reduces favoritism but takes more programming effort. The best non-deterministic implementation (PropSpec) attains a 34x speedup on 64 threads with less than 2% quality degradation. The speedup is comparable to approaches based on partitioning the move space across threads, but the quality loss is much smaller. The deterministic version (SerialProp) incurs no quality loss and can reach 5.9x speedup on 9 threads, which is better scaling than Q2P. This is mainly due to the coarsening of conflict detection and checking dependencies before move evaluation instead of allowing evaluation to be speculative.

There are several improvements we can make to our parallel placers: make proposals more efficient, modify the move generator to create more independent moves, and fine-tune the mechanism for ignoring big nets. Evaluating the scalability of our algorithms the very large FPGA benchmarks from [2] is also important future work.

Acknowledgments

We are grateful to NSERC and Altera for funding. We thank Amy Wang and Rahul Gopalkrishnan providing insights into the HTM and TLS results, and also thank Henry Wong for improving the formulation of SerialProp. Computations were performed on the `bgq` supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto.

References

[1] A. Ludwin and V. Betz, "Efficient and deterministic parallel placement for FPGAs," *TODAES*, vol. 16, no. 3, p. 22, 2011.

[2] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD," *FPL*, 2013.

[3] T. Feist, "Vivado design suite," 2012. Available: http://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf

[4] Q. Wu and K. McElvain, "A Fast Discrete Placement Algorithm for FPGAs," *FPGA*, 2012, pp. 115–118.

[5] T.-H. Lin, P. Banerjee, and Y.-W. Chang, "An efficient and effective analytical placer for FPGAs," *DAC*, 2013, p. 10.

[6] J. B. Goeders, G. G. Lemieux, and S. J. Wilton, "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition," *ReConFig*, 2011, pp. 41–48.

[7] S. Birk, J. G. Steffan, and J. H. Anderson, "Parallelizing FPGA placement using transactional memory," *FPT*, 2010, pp. 61–69.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[9] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," *FPL*, 1997, pp. 213–222.

[10] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," *FPGA*, 2008, pp. 14–23.

[11] W.-J. Sun and C. Sechen, "A parallel standard cell placement algorithm," *TCAD*, vol. 16, no. 11, pp. 1342–1357, 1997.

[12] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[13] M. Gilge, "IBM system Blue Gene solution: Blue Gene/Q application development," *IBM Redbook Draft SG24-7948-00*, vol. 9, 2012.

[14] T. Jain and T. Agrawal, "The Haswell microarchitecture–4th generation processor," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 3, pp. 477–480, 2013.

[15] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," *Distributed Computing Systems*, 2003, pp. 522–529.

[16] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2008, vol. 10.

[17] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," *PPoPP*, 2008, pp. 237–246.

[18] A. Wang et al, "Evaluation of Blue Gene/Q hardware support for transactional memories," *PACT*, 2012, pp. 127–136.

[19] J. Rose et al, "The VTR project: architecture and CAD for FPGAs from Verilog to routing," *FPGA*, 2012, pp. 77–86.

[20] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," *PPoPP*, 2005, pp. 142–152.