Parallelizing FPGA Placement Using Transactional Memory

Steven Birk [1], J. Gregory Steffan [#2], Jason H. Anderson [#3]

Department of Computer Science, # Department of Electrical and Computer Engineering, University of Toronto
University of Toronto
10 King's College Road
Toronto, Canada
[1] sbirk@eecg.toronto.edu
[2] steffan@eecg.toronto.edu
[3] janders@eecg.toronto.edu

***Abstract*—To capitalize on the growing abundance of multicore hardware, FPGA vendors have begun to parallelize the most compute intensive algorithms in their CAD software. However, parallelization is a painstaking and hence expensive process that limits the number of algorithms that can be cost-effectively parallelized. Transactional Memory (TM) promises an easier-to-use alternative to locks for critical sections in threaded code—allowing programmers to avoid deadlocks and data races, and also allowing critical sections to execute in parallel as long as they dynamically access independent data. In this paper, we present our work on using TM to parallelize simulated annealing-based placement for FPGAs. In particular, we use a software TM (TinySTM) to parallelize the placement phase of Versatile Place and Route (VPR) 5.0.2 [1]. With TM we very quickly produced a parallel and correct version of the software, allowing us to focus on incrementally tuning performance. We describe our experiences in tuning the TM system and CAD software, and the interesting algorithmic trade-offs that exist. In the end, we found that optimized transactional placement has the potential for scalable performance: our non-deterministic implementation achieves self-relative speedups over a single thread of 1.82x, 3.62x and 7.27x at 2, 4, and 8 threads respectively with little quality degradation. However, hardware support for TM is likely required to overcome the overheads of STM, as our implementation's single thread performance is 8x slower than sequential VPR.**

I. Introduction

The microprocessor technology road-map predicts a future with tens to hundreds of processors per chip and beyond, but with limited clock frequency improvements and likely simpler individual processors. Faced with the corresponding demise of sequential program performance, the software industry is compelled to parallelize existing software by introducing threads and synchronization to target these multicore processors. This challenge is especially significant for the FPGA companies whose CAD software (i) must manipulate hardware designs that are themselves growing with Moore's law, but (ii) is composed of a large number of sequential algorithms. The challenge is that parallel programming is a time-consuming and error-prone process. While progress has been made on parallelizing the most crucial algorithms (at great expense), future CAD parallelization efforts will require a more cost-effective approach.

A. Conventional Parallelization

There has been considerable work on parallelizing CAD algorithms using conventional methods, in particular for simulated annealing-based placement—the algorithm that we focus on in this paper. Unfortunately, most parallelized software running on the parallel hardware of the time had difficulty competing with the cost/performance of the corresponding sequential software running on the latest uniprocessor system. However, with multi-core processors becoming ubiquitous, parallel CAD algorithms are now a necessity. As evidence, the FPGA companies have begun to parallelize their CAD software, and Altera has recently published their efforts on parallelizing the simulated-annealing-based placement phase of the Quartus CAD software [2]. Analytical placement has also been parallelized by Chan and Schlag [3], targeting a network-based computing environment.

The parallel software presented by the Altera authors is the result of extensive development effort, and achieves speedups of 1.7x for two threads and 2.2x for four threads. Their approach has one thread that proposes and finalizes moves, while the remaining threads evaluate moves concurrently. All moves are tracked in an ordered buffer, and a software structure called a "dependency checker" is used to monitor memory accesses and detect any conflicts between moves that are being evaluated in parallel. A key insight is that these software constructions strongly resemble the services provided by *Transactional Memory* (TM), a parallel programming paradigm that has become the focus of numerous computer systems research groups and is poised to enter the mainstream.

B. TM Parallelization

Typical parallel software development is an all-or-nothing process where the parallelized application is very unstable and produces incorrect results until bugs such as data races and deadlocks are found and fixed. Transactional memory aims to offer a smoother development process without a long period of unstable code, so that developers can focus on optimizations and performance tuning. The goal is for parallel programming with TM to be as easy as using a single global lock to protect shared data, while providing the high performance of fine-grained locking.

Transactional memory requires the programmer to specify sections of code that run as "transactions". The TM system automatically monitors all accesses to shared memory during a transaction, and detects any conflicts between concurrently executing transactions. Conflicting transactions can be aborted, rolled-back, and re-executed such that the system makes correct forward progress.

In this paper, we examine TM as a means of parallelizing FPGA CAD software. Specifically we use a case study of parallelizing simulated annealing-based placement within the Versatile Place and Route (VPR) CAD software, using the TinySTM research software TM (STM) system [4]. We call the resulting software *Transactional VPR* (TVPR), although we have so far only transactionalized the placement phase. We describe our implementation of TVPR, including optimizations and algorithmic changes. As we will demonstrate, TM shows potential for scalable parallelism, as the resulting TVPR system exhibits near-linear scaling over its single thread performance, with minimal impact on the overall quality of placement.

C. Related Work

Simulated annealing-based placement involves choosing a pair of blocks at random, swapping their positions, and evaluating the impact of this swap on a chosen cost function. Depending on the impact, the swap will either be accepted or rejected. This process continues until the cost function converges to a satisfactory value. Previous work by Chandy, Kim *et al.* [5] discusses four main methods for parallelizing this general algorithm: *Move Acceleration*, *Parallel Moves*, *Multiple Markov Chains*, and *Speculative Computation*. TVPR is an implementation of *Parallel Moves*—i.e., we evaluate multiple swaps in parallel.

Some preliminary work has demonstrated the potential to optimistically parallelize CAD algorithms. Watson *et al.* demonstrate the potential for using TM to parallelize Lee's routing algorithm through the use of a simulation of an abstracted TM system, analyzing the available parallelism and amount of work done [6]. They begin with a simple approach, and then adapt their parallel implementation further to achieve significantly more parallelism. In our work, we use a full software TM system and measure its execution on a real multicore system. Thread-Level Speculation (TLS) is a more hardware-centric form of optimistic parallelism with many similarities to TM. Prabhu and Olukotun [7] manually apply TLS to the SPEC2000 benchmark suite, which includes an earlier version of VPR. They also demonstrate that standard parallel optimizations can further improve the performance of parallel execution.

D. Contributions

In this paper, we make the following contributions: (i) we present the first implementation and evaluation of transactionalized simulated annealing-based placement on a real system; (ii) we demonstrate that this method of parallelization supports incremental performance optimization, where developers can invest extra effort to improve performance rather than debugging a broken parallel implementation; (iii) we discuss a method of using TM to provide serial equivalence; (iv) we demonstrate that an optimized transactional implementation is scalable, resulting in near linear speedup over the single-threaded TVPR.

II. Using Transactional Memory

In this section we describe how our STM system was chosen, how it functions, and its tunable parameters and features.

A. Choosing a TM System

Transactional memory systems generally fall into two categories: those that are hardware-based [8], [9], and those that are software-based [4], [10]–[12]. Hardware-based systems are presently in the conceptual stage, and can only be studied through simulation. However, such systems have the advantage that their overheads are much lower than those that are software-based. Software-based systems usually incur high overheads, but allow for the development of software that can be evaluated on real systems. To compute a full simulated annealing placement for a reasonably-sized circuit on a simulated [hardware] TM system would take weeks or months—hence for this work we opted to instead study a software-based TM system.

There are currently many different software transactional memory (STM) systems to choose from. The work of Dragojević et al. [10] compares the performance of many of the most well known STMs and introduces their own implementation, SwissTM, which they believe overcomes some of the limitations of previous implementations. For our work we compared the performance of SwissTM, TinySTM (a top performer in the comparisons performed by Dragojević et al.), and TL2-x86, which is a STAMP [13] group x86 port of SUN's well-known TL2 [11]. We used a preliminary implementation of TVPR and compared the wall clock time required to complete the placement process when using each of the above STMs with a varying number of threads. In these experiments we found that the performance of TinySTM was the best for our particular application. However, it is important to note that a "transactionalized" program can normally be ported to different TM systems fairly easily.

B. TinySTM

TinySTM [4] is a research STM, the joint work of the University of Neuchâtel (Switzerland) and the Dresden University of Technology (Germany). In our experiments we use TinySTM version 0.9.7. TinySTM is a `C` software library linked at compile time. The developer specifies transaction boundaries and annotates shared reads and writes within their source code using function macros. For example, `START` and `COMMIT` macros mark the start and end of transactions respectively.

Whenever an annotated write occurs during runtime, a call is instead made to the TinySTM library's TM "write" function. This function takes the memory address of the write, applies a shift, and uses the resulting value as an index into a fixed-size hash table to find the associated TM "lock" for that address. If the lock is already owned (i.e. by another transaction) a conflict has occurred and the transaction must abort. Otherwise, the thread takes ownership of the lock and logs both the address and value of the write into a transaction-specific data structure within the library called the "write set". A similar TM "read" function is used whenever a shared read is encountered. A successful read (i.e. a read to a location whose lock is not held) is added to another transaction-specific data structure known as the "read set".

At the end of a transaction (i.e. where the `COMMIT` macro occurs at runtime) a final function call is made that attempts to "commit" the transaction. At this time, a TM "validate" function iterates over the read set for the transaction to ensure that all reads are still valid. If so, entries in the transaction's write set are written to main memory, and all held locks in the hash table are released. If an entry in the read set is no longer valid (i.e. was written by another transaction), the transaction must abort. Whenever a transaction must abort, a call to the library's TM "abort" function is made which drops all locks owned by the transaction and restarts the transaction by jumping back to the beginning.

C. *TinySTM Parameters*

Most STM systems offer many tunable parameters to improve performance, including TinySTM. The tunable parameters available include those that determine whether the system is *optimistic* or *pessimistic*, how the system reacts at the time of an abort (referred to as the "contention manager") and others which are specific to the locking mechanism used by TinySTM. Optimistic parameters are those that favor low-contention environments (where aborts are rare) because they reduce the time required to commit transactions at the cost of making aborts slower. Pessimistic choices are more appropriate for systems of modest contention, as they mean aborts occur faster and are detected earlier. We consider the following implementation options, which can each be either optimistic or pessimistic:

- *Versioning*: This parameter controls when TinySTM makes modifications to main memory. With *lazy versioning* (called *write-back* by TinySTM), updates are made to main memory at commit time, where they are written from the TM system's buffers. With *eager versioning* (called *write-through* by TinySTM), modifications to main memory are done immediately upon memory access, and an undo log is used in the case of a transaction abort.
- *Lock Acquisition*: This parameter controls when the TM system acquires locks for memory accesses. With *lazy* lock acquisition (called *commit-time* by TinySTM), the TM system only checks for locks at commit time. With *eager* acquisition (called *encounter-time* by TinySTM), locks are acquired immediately upon a memory access.

A *contention manager* is used to determine how a transaction reacts after an abort. An example policy implemented by a contention manager is for the aborting transaction to back-off for a pre-determined amount of time before re-executing. TinySTM offers a choice of several simple contention managers [14].

TinySTM also offers parameters to tune its locking mechanism. Since memory locations are mapped to locks using a hash table, there is the possibility of false-sharing due to contiguous areas of memory being mapped to the same lock. TinySTM offers two ways to trade-off between the complexity of the hash table and the probability of false-sharing. The first is to control the number of entries in the hash table (i.e., the total number of locks), and the second is to control the memory address shift used to map an address to a lock—which specifies the granularity of memory regions that map to the same lock. We describe our tuning of these parameters later in Section III-D.

III. Parallelizing VPR with TM

We separate the description of our parallel implementation into three categories. We first describe the simulated annealing algorithm, the high-level modifications made to support multiple cores, and how we set the TinySTM parameters. We then describe the more specific modifications, which are classified as either optimizations or algorithmic changes. Optimizations include those that are standard to parallel programming, and others that are more specific to CAD. Algorithmic changes are simple modifications that we introduced to allow TVPR to better interact with the TM system.

A. *Simulated Annealing*

The algorithm used for placement in VPR is called *simulated annealing*. The process begins with a random placement of all the logic *blocks* to be placed and proceeds through a series of iterations of making random *swaps* of blocks (i.e. switching their positions), and evaluating the *cost* of the change using a chosen cost function. If the change is accepted (either because it results in a benefit to the cost function, or probabilistically for swaps that negatively impact cost) it is maintained in the placement, otherwise the blocks are reverted to their original positions. The number of iterations that the algorithm performs is governed by a concept of *temperature*. Iterations can be thought of as temperature *steps* where during each step a fixed number of swaps are performed. At the end of each step, the temperature value is reduced based on the success rate of the swaps performed, and a new step begins. Once the temperature reaches a particular stopping value, the arrangement of logic blocks at that time is reported as the final placement.

B. *Parallel Moves with TM*

Our parallel implementation falls into the *Parallel Moves* category. This approach requires very few changes to the

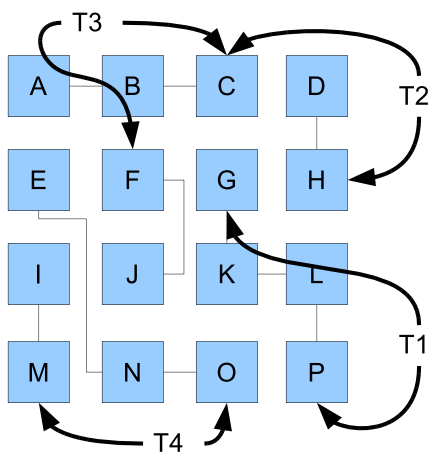

Fig. 1. A representation of *parallel moves* using TM. Thread T2 and Thread T3 will conflict accessing block C.

underlying algorithm already used in VPR. We implement each swap attempt as a transaction. Therefore, as long as no two swaps access the same data structures, there are no conflicts and they execute in parallel. This idea is illustrated in Figure 1, which shows both blocks and a few very basic nets connecting them. We label threads as T1-T4; thread T2 and T3 will conflict as they both require access to block C. Aborts, however, are not limited to conflicts on block-related structures: for example, if block O and P in the figure were connected to the same net then threads T1 and T4 would also conflict. It is worth reinforcing that combinational timing paths through a circuit are not a source of conflicts: While it is true that altering the placement of a block on a timing path may affect the timing slack of other blocks on the same path, timing analysis and slack updates are not made after individual block moves; rather, such updates occur only at the end of an annealing iteration.

Since both swaps and transactions are speculative (meaning that they each may succeed or fail), we use different terminology to refer to the success and failure of each of them:

- *Transactions* either commit successfully, or must abort and re-execute.
- *Swaps* are either accepted and maintained in the overall placement, or else they are rejected and reverted back to their original positions.

Getting started with TM was a very quick process. Including time to familiarize with the VPR source code and TinySTM, producing the first version of TVPR running on multiple cores took approximately 1 month for a single developer.

Integrating TinySTM into VPR involved parallelizing the main annealing `for` loop that executes for each simulated annealing temperature step. We use the GNU `C` compiler implementation of the OpenMP [15] API to create a pool of worker threads (typically 2, 4, or 8 threads) and we evenly split loop iterations (i.e. swap attempts) among the threads in the pool. For example, with four threads, each thread is performing one quarter of the swap attempts for each temperature step.

To include TM, we were required to insert the appropriate transaction boundaries and annotate read and write accesses to shared data. Currently, TinySTM provides compiler tools that can perform read/write annotation automatically without the need of the programmer; however, at the time of writing, these tools were in a very early development stage and so we opted not to use them.

The function within VPR that performs the swap evaluation is `try_swap`. In our implementation, the entirety of the `try_swap` function is a transaction—hence the size of our transactions can vary significantly depending on the complexity of the swap being performed (i.e. the number of nets connected to the blocks) and on the size of the circuit being placed.

C. Random Number Generation

For random numbers, VPR uses a simple linear congruential generator, where each number is a function of the previous one. Such a generator is inappropriate in an environment with multiple threads where we do not know how many random numbers each thread will need. Rather than introduce contention on the generator by using TM to access new random numbers, we opted to implement a generator more suited to a multi-threaded environment.

In the interest of keeping the generator simple, we use a basic extension of VPR's generator. We introduce a *master* stream of seeds, where the VPR generator initially creates a number of random seeds equal to the number of threads being used. Each of these seeds is then used by a modified version of the VPR generator to create a stream of random numbers for each thread, known as *child* streams. Each thread then calls upon its child stream whenever it requires a random number.

An important property of our system is that new random numbers are generated on transaction abort. This means that when a transaction must abort and re-execute, it is in essence attempting an entirely new swap. This leads to a behavior that we call *swap favoritism*, where the simulated annealing process is preferring a particular type of object swap over others. We describe our first encounter with this behavior in Section IV-B, where swap favoritism had a significant impact on performance. However, even after correcting the favoritism, we still observe a more minor form of swap favoritism in our current implementation. Specifically, our system tends to favor transactions performing 'simpler' swaps, meaning swaps that involve blocks connecting to fewer and/or smaller nets. This is because such swaps have a lower probability of conflict, and take less time to evaluate (resulting in faster transactions), and therefore are more likely to commit. The result of this favoritism is an increased speedup of annealing, and can be observed in the slightly super-linear speedup of some benchmarks in our results shown in Section VI.

D. Tuning the STM for VPR

Throughout development, we experimented with the tunable parameters available within TinySTM to improve performance further, as follows:

Lock Acquisition We found that this parameter had the most significant impact on performance. Depending on which blocks are chosen for a swap (and therefore which nets) the number of memory accesses (and thus transaction size) can grow quite large. When commit-time locking is used, if all threads are performing large transactions (which are likely to conflict with each other) they will run to completion before the conflict is detected. As a result, all the time spent by threads whose transactions are aborted is wasted. Therefore we use encounter-time locking to avoid this situation as conflicts are detected immediately upon memory access.

Versioning This parameter had relatively little impact on performance. However, we did find that write-back versioning performed slightly better, most likely because our system exhibits a reasonable amount of contention. Write-back versioning allows aborts to occur faster, since updates to main memory do not need to be rolled back.

Contention Management We found that with our current non-deterministic implementation, the most simple contention manager that immediately re-executes an aborted transaction worked best—back-off or priority-based contention managers were unnecessary, due to the fact that we generate new random numbers for transactions that abort and retry (as described above in Section III-C).

TinySTM's Locking Mechanism One distinct problem with TinySTM's locking mechanism is that the optimal size of the hash table varies with the size of the circuits being placed. With a larger circuit, more memory accesses are made, and thus a larger hash table is needed to prevent false-sharing. This suggests compelling potential future work to explore dynamically tuning the hash table. To minimize false-sharing, we associate the smallest possible memory region with each lock. We perform the smallest shift possible to acquire the hash table index, and also increase the hash table size (by a factor of 16 from the TinySTM default).

IV. Optimization

Since we were able to develop a correct initial version of TVPR fairly quickly, we were able to spend the bulk of development time optimizing performance, as we describe in this section.

A. Standard Optimizations

The first two optimizations that we performed on the transactional code are standard to parallel programming in general:

- *Reductions*: Since the summation variables within the main swapping loop are updated at the end of every swap attempt, they were a significant cause of contention as all threads were trying to access them. By allowing each thread to modify local copies which are reduced at the end of the loop, we avoid this unnecessary contention.
- *Code Scheduling*: Code that does not need to execute transactionally was hoisted out of the transaction whenever possible. For example, we hoisted several calls to `malloc` that were being made for every swap attempt and creating a lot of unnecessary transaction aborts.

The effect of these simple standard techniques was quite significant, improving both performance and scalability by reducing the number of aborts and the serialization artifacts of VPR.

B. Privatization

The optimization that had the most significant impact on the performance of our parallel implementation was privatization. This is another standard parallel programming optimization that involves creating private (per-thread) copies of frequently modified variables, which also leads to fewer transaction aborts.

In our case there was one particular variable significantly impacting our runtime behavior, an integer array called `x_lookup`. This array is used locally by each swap attempt to find the second block that will be used in the swap. It acts as a cache of possible columns to choose from, and is only used if the initial block chosen is not an I/O-type block.

During initial experiments, we discovered that when this variable was shared, it was causing over 80% of all transactional aborts. Every transaction attempting a non-I/O block swap uses this data structure. Since we use encounter-time locking, once a transaction had acquired the appropriate lock for the first entry in the array, no other transactions could do the same—so subsequent transactions would repeatedly abort trying to acquire the lock. In other words, only one non-I/O swap could be in-flight at a time. However, swaps of I/O blocks were unaffected. Consequently, the transactions that committed most often were I/O swaps, resulting in I/O-type swap favoritism. By performing mainly I/O-type swaps (which are much easier to compute than non-I/O swaps), our system was exhibiting inflated speedup but with a severe degradation in quality-of-result.

By creating a local copy of this variable for each thread, non-I/O swaps were able to execute properly in parallel, eliminating the I/O favoritism. This in turn reduced our abort rate significantly, and significantly reduced our impact on the quality-of-result for the resulting placement.

C. Leveraging TinySTM's Store Buffers

To simplify the evaluation of a swap, VPR makes the appropriate changes directly to the data structures involved. This means that upon rejection, a swap must be reverted within the data structures. Since within our TM system all changes to main memory are buffered (when using write-back versioning), a swap is performed within the STM's buffers, and upon rejection is reverted within them as well. At commit-time, we are wastefully overwriting memory with the same values that are already there. We improve on this by reverting the swap in the TM buffers, and notifying the STM that it does not need to write its values back to main memory. This

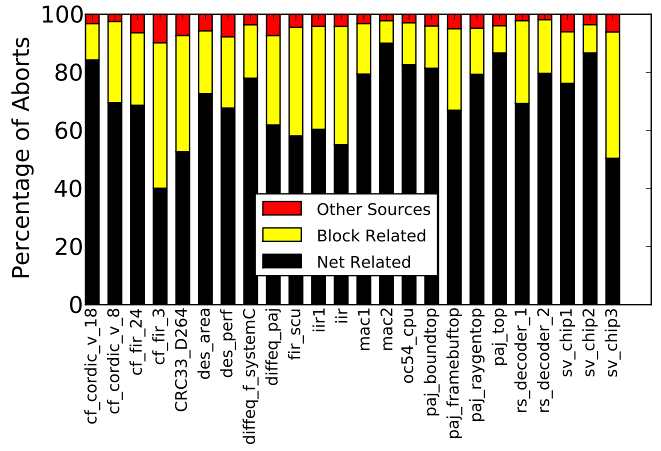

Fig. 2. Abort breakdown by VPR data structures, for all benchmarks.

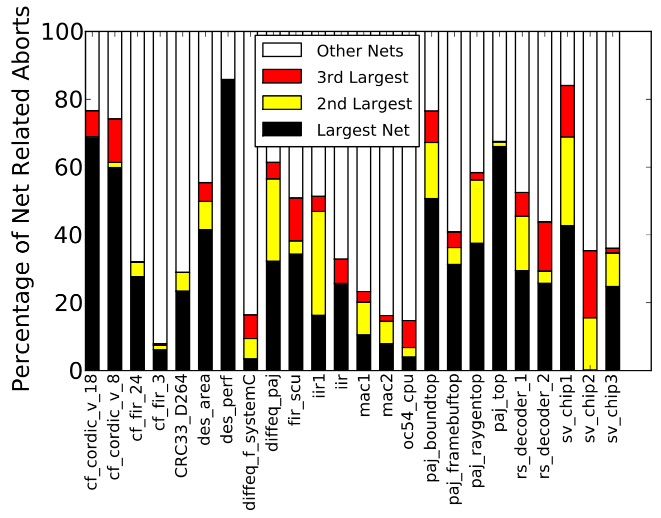

Fig. 3. Contribution of three largest nets to total net-related aborts, for all benchmarks.

change is an example of how TM and the algorithm being accelerated can work more synergistically.

D. Algorithmic Changes

Through further contention analysis, we discovered that the vast majority of remaining aborts were occurring on the most commonly accessed net-related data structure. Figure 2 displays a breakdown of total aborts by data structure. As shown, for some benchmarks, net-related data structures account for up to 90% of aborts, prompting us to examine methods of reducing the contribution of aborts caused by such structures. Further experiments revealed that for some benchmarks there were a very small number of their nets contributing to a large proportion of aborts. Figure 3 illustrates the contribution to net-related aborts seen by each benchmark by the 3 largest nets within that benchmark. The `des_perf` benchmark, for example, has one net contributing to over 80% of all its net-related aborts.

TABLE I
QUALITY-OF-RESULT IMPACT OF IGNORING HIGHLY-CONNECTED NETS. A + INDICATES A DEGRADATION, A - INDICATES AN IMPROVEMENT.

	Wire-length		**Crit. Path Delay**	
Thresh.	Avg.	Max.	Avg.	Max.
20%	-0.4%	+3.2%	+1.5%	+18.1%
10%	-0.2%	+3.5%	+1.8%	+18.1%
5%	-0.3%	+4.5%	+4.5%	+34.7%
2%	+5.3%	+71.3%	+7.4%	+40.7%

To alleviate this problem, we altered the VPR algorithm to ignore highly-connected nets during swap evaluation. We accomplish this by doing some pre-processing of the netlist to determine the size of each net in terms of the percentage of total blocks that it connects. We call this value the *net coverage*. Nets having a net coverage above a fixed threshold are ignored during swap evaluation.

One of the major concerns of this change was the adverse effect it may have on the quality-of-result. To examine this, we performed experiments using sequential VPR. Our experiments involved running sequential VPR with this modification, varying the threshold and the initial seed used. We used 4 thresholds and ran sequential VPR with 5 different initial seeds at each threshold. We then averaged the resulting estimated placed wire-length and critical path delay over all seeds for each threshold, for each benchmark. Table I lists both the average and worst-case relative change in quality-of-result vs. a 100% threshold (no nets ignored), across all benchmarks. Although there appears to be an improvement in the average placed wire-length for some thresholds, this is likely due to noise in placement (which is heuristic).

We found that a threshold of 10% had an acceptable trade-off between the amount of aborts it removed and the degradation of quality-of-result incurred. Note that in our results presented in Section VI, both the sequential and parallel version of VPR are using this “ignoring nets” featured, to avoid skewing the run-time results in favor of our approach—since by ignoring some of the largest nets we are reducing the amount of work VPR must do to evaluate a particular swap.

V. Experimental Setup

In this section, we will describe the hardware and software we used in our experiments, as well as the methodology used to acquire our results.

A. Versatile Place and Route

VPR [1] is a research placement and routing tool for FPGAs developed at the University of Toronto. In this work, we use the most up-to-date version 5.0.2, which improves on the version included as a SpecINT2000 benchmark with support for heterogeneous block types and single-driver routing. We use the benchmark circuits provided with the VPR distribution, as shown in Table II, and the CAD flow described in Figure 4. We use an FPGA architecture of 4-LUT logic blocks with a cluster size of ten. We run VPR with the default placement options.

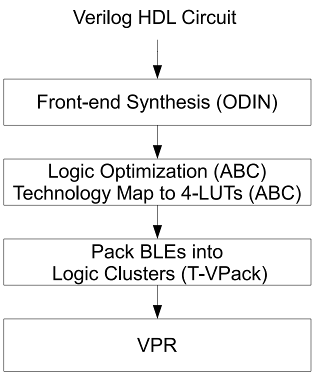

Fig. 4. CAD Flow.

TABLE II
BENCHMARKS.

Circuit	# Blocks	# Nets
paj_top_hierarchy_no_mem	6423	45151
sv_chip2_hierarchy_no_mem	5082	34532
mac2	1384	6234
oc54_cpu	443	1989
des_perf	845	4626
rs_decoder_2	315	1792
cf_cordic_v_18_18_18	716	3723
sv_chip1_hierarchy_no_mem	2060	12462
mac1	493	1837
diffeq_f_systemC	377	1713
rs_decoder_1	190	986
fir_scu_rtl_restructured_for_cmm_exp	124	667
paj_boundtop_hierarchy_no_mem	842	2489
cf_cordic_v_8_8_8	162	685
des_area	347	1026
paj_framebuftop_hierarchy_no_mem	214	626
cf_fir_24_16_16	871	4335
paj_raygentop_hierarchy_no_mem	952	2617
iir	94	521
iir1	138	511
diffeq_paj_convert	347	825
sv_chip3_hierarchy_no_mem	68	199
cf_fir_3_8_8	90	296
CRC33_D264	350	367

B. Measuring Performance

We perform our experiments on a real machine, with two Quad-Core Intel Xeon E5345 processors running at 2.33 GHz, 4 GB of main memory, and running the Linux Kernel version 2.6.18.

The implementation we report results for in this paper is not *serially equivalent* [2], and consequently, comparing the performance against the sequential version must be done carefully. VPR's simulated annealing schedule is dynamic[1], therefore our implementation, since it is not deterministic, can follow a different path to convergence each time it is run, even when given the same initial random seed. This is similar to what occurs with the original sequential algorithm when different seeds are used.

[1]Meaning that temperature adjustments are made based on cost change statistics.

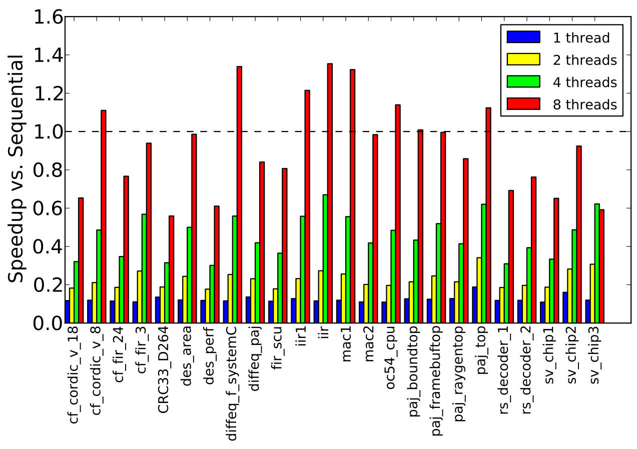

Fig. 5. Speedup vs. Sequential-IBN.

We chose a simple approach of comparing the total wall-clock time required to complete the sections of code that we have parallelized, both in sequential VPR and our implementation. We do this to avoid including the extra time involved in timing-analysis, which is not parallelized in our implementation and is an important research challenge in itself. We average the results over three trials, and both the sequential and parallel algorithms are given the same initial seed.

To determine the impact on quality-of-result, we compare both (i) the placed wire-length and (ii) the critical path delay of the resulting placement. In order to determine the value of these metrics we use the router in VPR to search for the minimum necessary tracks-per-channel in an initial sequential baseline run. After this value was determined, we increased it by 30% and fix all future sequential and parallel runs to use the fixed width. Interconnect is therefore invariant for each circuit, across all our experiments. We use the final critical path delay and placed wire-length values of the resulting routed circuit to make our quality-of-result comparison. These reported values are averaged over three trials.

VI. RESULTS

In this section, we present the performance and quality-of-result impact of our implementation of TVPR. Figure 5 illustrates the performance of TVPR against the original sequential VPR (without any STM overheads), including the "ignoring big nets" (IBN) optimization, across all benchmark circuits for 1, 2, 4 and 8 threads. Only a few benchmarks perform better than sequential VPR, and the ones that do accomplish this at 8 threads. This poor speedup is due to the overheads of STM: in our experiments, we found that TVPR using 1 thread was approximately 8x slower on average than sequential VPR. This can be seen in the figure by looking at TVPR's performance for a single thread. This extensive slowdown is due to the large amount of overhead in instrumenting all shared reads and writes inherent with any STM (as described

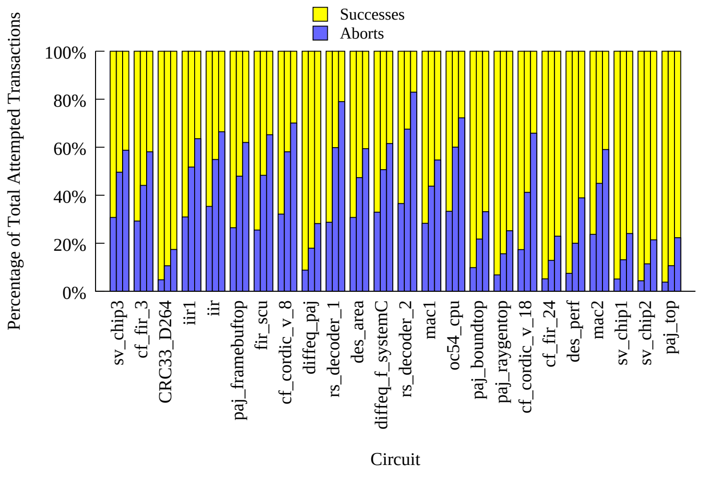

Fig. 6. Abort Rates for 2, 4, and 8 threads (shown respectively as three adjacent bars per benchmark). Benchmark circuits are sorted smallest to largest (left to right) based on total number of nets.

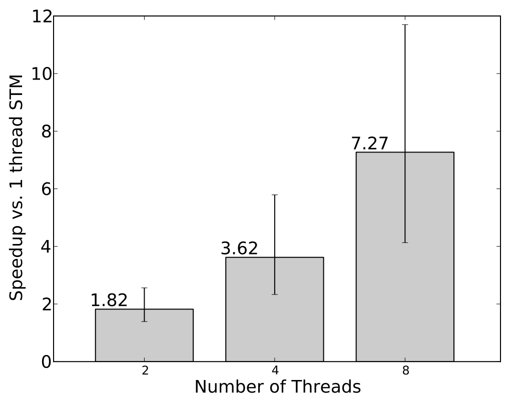

Fig. 7. Self-relative scaling on a dual-socket quad-core Xeon (i.e., up to 8 threads relative to 1 thread with STM overhead). Plot includes min/max error bars.

in Section II-B). The slight super-linear speedup demonstrated by some benchmarks (e.g. `diffeq_f_systemC`, `mac1`) is the result of the favoritism described in Section III-C.

Figure 6 illustrates the abort rate statistics of TVPR for 2, 4, and 8 threads for all benchmarks. The value of the abort rate is the percentage of total attempted transactions that resulted in an abort. We did not observe a direct correlation between the relative values of properties of the circuits themselves (i.e. number of nets, number of blocks, average per-block net fan-out) and their relative abort rates. For example, a circuit with a larger average per-block net fan-out than another circuit did not necessarily exhibit a higher abort rate. The figure also illustrates no clear correlation between the abort rates and the total number of nets in a circuit. However, we did find that the largest three benchmark circuits that we used exhibited reasonable abort rates at 8 threads. As shown in the figure, these circuits (`sv_chip1` , `paj_top` and `sv_chip2`) had abort rates at 8 threads of 24.04%, 21.39% and 22.29% respectively. Larger circuits exhibit less contention leading to smaller abort rates.

The overall impact on the quality-of-result is a maximum degradation of 5% on the post-routed wire-length, and 10% on the critical path delay, both at eight threads. On average, the impact is 1% on the post-routed wire-length, and 1% on critical path delay.

Despite the run-time relative to sequential, it is evident that TVPR scales quite well (almost linearly) with the number of threads used. Figure 7 illustrates the speedup of our implementation relative to its own single thread performance, averaged over all benchmark circuits. This plot may also be interpreted as the projected speedup, if there were no overhead.

A. Potential for Hardware TM

Hardware support has the potential to drastically improve the performance of TM. This is because operations which are expensive in software, such as instrumenting all reads and writes as well as detecting conflicts, can be optimized in hardware.

While hardware support for TM (HTM) is not yet commercially available, primitive support was announced as part of the specification for SUN's Rock processor [16] and both IBM and Intel have several research groups studying TM. Simulated studies of LogTM [8], an academic HTM, show that it can provide up to 30x greater throughput than SUN's STM called TL2. A compelling design is a hybrid TM—a mix of hardware and software that can be thought of as hardware-accelerated STM. Recent work by Lev *et al.* [17] explores (via simulation) a hybrid TM system called Phased Transactional Memory (PhTM) that performs much better than pure STM (TL2 [11]) and quite close to pure HTM. Although we found TinySTM performs better than TL2, we believe this analysis is illustrative of the potential improvement when using a hybrid or full hardware TM system.

VII. Supporting Serial Equivalence

A major concern of CAD tool developers and users is determinism. The result of placement should be the same given the same initial seed, and regardless of the number threads. This constraint has been given the name *serial equivalence* by previous work [2]. While our implementation is currently not serially equivalent, this section discusses modifications to TVPR in order to achieve serial equivalence.

The modifications involve making a change in how the random number generator's master and child streams are generated, and requires using a TM system that supports *ordered* transactions [18]. In particular, threads will perform swaps in a deterministic round-robin manner and threads will also use a deterministic random number sequence. Ordered transactions imply a fixed commit order within the TM system. An example of this is a token system, in which a thread must wait until it holds the token before it commits its transaction. The token is passed round-robin among the threads. With these changes, along with the re-use of random numbers on transaction re-execution, we can achieve both determinism and serial equivalence for any number of threads.

We expect that attempting to serialize transactions using a token within a software-based transactional memory system would result in significant slow-downs due to the extra overhead involved. However, we also expect that hardware-based systems may be able to provide this functionality without such a drastic cost to performance [18].

VIII. Conclusions

Transactional memory (TM) is a new paradigm in parallel computing that is gaining traction in industry and the research community, primarily due to its ease-of-use relative to threads, locks and condition variables. We presented the first simulated annealing-based placer (TVPR) parallelized through TM. Our approach is based on modeling placement swaps as transactions that execute concurrently, and incorporates algorithmic optimizations that reduce transaction abort rates. We found that the scalability of our algorithm is promising. In particular, near-linear speedups versus single-threaded TVPR were observed as parallelism was increased, with minimal deleterious effect on placement quality. However, results also show that TM support in hardware is likely necessary if significant speedups are to be achieved over conventional sequential VPR.

References

[1] "Versatile Place and Route," http://www.eecg.toronto.edu/vpr/, University of Toronto.

[2] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for fpgas on commodity hardware," in *Symposium on Field Programmable Gate Arrays*, 2008.

[3] P. Chan and M. Schlag, "Parallel Placement for Field-Programmable Gate Arrays," in *International Symposium on Field Programmable Gate Arrays*, 2003.

[4] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Symposium on Principles and Practice of Parallel Programming*, 2008.

[5] J. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee, "An evaluation of parallel simulated annealing strategies with application to standard cell placement," *IEEE Trans. on Comp. Aid. Design of Int. Cir. and Sys*, vol. 16, pp. 398–410, 1997.

[6] I. Watson, C. Kirkham, and M. Lujan, "A Study of a Transactional Parallel Routing Algorithm," in *International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[7] M. K. Prabhu and K. Olukotun, "Exposing Speculative Thread Parallelism in SPEC2000," in *Symposium on Principles and Practice of Parallel Programming*, 2005.

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-Based Transactional Memory," in *Intl. Conference on High-Performance Computer Architecture*, 2006.

[9] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on Chip-Multiprocessors," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2005.

[10] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *Conference on Programming Language Design and Implementation*, 2009.

[11] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.811

[12] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive Software Transactional Memory," in *Proc. of the 19th Intl. Symp. on Distributed Computing*, 2005.

[13] "Stanford Transactional Applications for Multi-Processing," http://stamp.stanford.edu/, Stanford University.

[14] "TinySTM," http://tinystm.org, TinySTM.

[15] "OpenMP," http://openmp.org/wp/, OpenMP.

[16] M. Moir, K. Moore, and D. Nussbaum, "The Adaptive Transactional Memory Test Platform: A tool for experimenting with transactional code for Rock," in *TRANSACT*, April 2008.

[17] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased Transactional Memory," in *TRANSACT*, 2007. [Online]. Available: http://www.cs.rochester.edu/meetings/TRANSACT07/

[18] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, J. Davis, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.