High-Quality, Deterministic Parallel Placement for FPGAs on Commodity Hardware

Adrian Ludwin
aludwin@altera.com

Vaughn Betz
vbetz@altera.com

Ketan Padalia
kpadalia@altera.com

Altera Corporation
151 Bloor Street West, Suite 200
Toronto, ON, M5S 1S4, Canada

ABSTRACT

In this paper, we describe the application of two parallelization strategies to the Quartus II FPGA placer. The first uses a pipelining approach and achieves speedups of 1.3x on two processing cores. The second uses a parallel moves approach and achieves speedups of 2.2x on four cores. Unlike all previous parallel moves algorithms, ours is deterministic and always gives the same answer as the serial version of the algorithm, without any significant reduction in performance.

We also describe a process to quantify multi-core performance effects, such as memory subsystem limitations and explicit synchronization overhead, and fully describe these effects on a CAD tool for the first time. Memory limitations alone are found to cost up to 35% of total runtime. Unlike previous algorithms, our algorithms have negligible explicit synchronization overhead. These results are relevant to both CAD designers and to any developers seeking to parallelize existing software.

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

Keywords

Parallel placement, FPGAs, Timing-driven placement

1. INTRODUCTION

The Quartus® II design software is a commercial CAD tool used to implement designs on Altera® devices. This paper describes parallel versions of the Quartus II placer's most CPU-intensive algorithm.

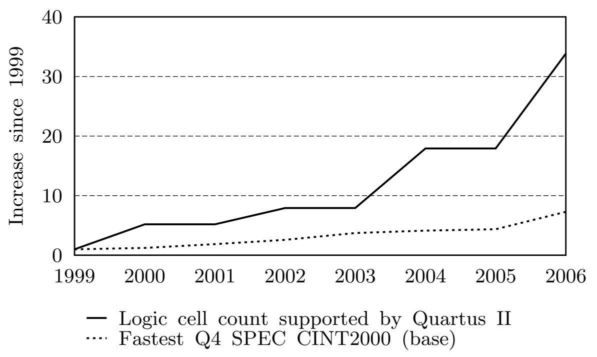


Figure 1: FPGA size vs. per-core performance.

1.1 Motivation

Since the launch of Altera's Quartus software in 1999, FPGA device sizes have increased at almost five times the rate of processor core speeds (Figure 1). In addition, there is now broad agreement that the exponential growth in per-core performance is slowing, and that future processors will have more cores to compensate for this reduction in per-core progress [1].

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 [2]. 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 multi-core processors which are now becoming common. Commodity PCs contained two cores in 2007, and four cores will become common in 2008. Since the number of cores is expected to increase exponentially for the foreseeable

future, parallel algorithms may be well suited to handle increasing FPGA sizes. This is the approach we explore in this paper.

1.2 Constraints

Parallelizing a commercial FPGA placement tool involves respecting the following constraints which are not commonly encountered in prior parallel placement work. Firstly, it must run on commodity hardware such as Windows and Linux desktops, 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 placer currently optimizes wirelength, critical path delay, localized routing congestion and power, and any replacement must deliver equivalent quality on all of these goals. It must also handle more complicated circuits than are commonly encountered in academic work, including arithmetic chains, RAM and DSP blocks and sophisticated floorplanning constraints. Creating a new placer from scratch with equal capabilities and quality would be very difficult, and for this reason we decided to parallelize the existing version.

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 (an exception is [3]), but is vital in a commercial context for two 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 thousands of regression tests prior to each release of Quartus II. It would be extremely difficult to diagnose failing tests whose results changed randomly.

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.

There are two clear advantages to including this constraint. Firstly, we can trivially show that the quality of the parallel algorithm is identical to that of the original, serial algorithm[1], thus meeting our second constraint. Secondly, testing can be significantly simplified (and automated) since any difference between serial and parallel results proves, by definition, the existence of a bug.

As we will show in the remainder of this paper, achieving serial equivalency (and hence determinism) had little impact on the speedups obtained by our algorithms, at least at two and four processors. Therefore, both the algorithms presented in this paper are serially equivalent.

2. PRIOR WORK

To the best of our knowledge, no published work in parallel placement optimizes for any criteria except wirelength. Furthermore, all previously published algorithms with good performance are non-deterministic and all deterministic algorithms have poor performance. Our algorithm, by contrast, is both serially equivalent and obtains good speedups.

Considerable research has been performed into the placement problem. The most popular approaches include recursive partitioning [4, 5], analytic [6], genetic [7] and randomized, the best-known of which is simulated annealing [8, 9, 10]. Of these, the latter is the most-studied for FPGA placement, mainly since it directly handles FPGAs' complex legality constraints, while other approaches require sophisticated legalization steps [11].

There have 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 [12].

Our placer's core algorithm uses a technique known as iterative improvement, whose low-level implementation is similar to simulated annealing. Both these algorithms consist of inner loops that repeatedly perturb a placement to form another placement, evaluate the impact of this perturbation and decide whether to accept or reject it. The perturbation is called a *move* and the evaluated impacts are called *costs.* Many thousands or millions of moves are considered while placing a typical circuit.

Three major approaches for parallelizing simulated annealing have been described: move acceleration, fine-grained parallel moves and coarse-grained parallel moves, though the lines between them are often blurred.

2.1 Move Acceleration

Move acceleration, as described in [13], 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 and has been little studied since it was originally proposed.

Modern commodity hardware suffers from severe synchronization overhead (see Section 3.2). Since the cores must synchronize when the cost evaluation is finished, this would overwhelm any gains obtained from the cost parallelization on these platforms. The algorithm we present in Section 5 proposes future moves separately from their evaluation, similar to the third core in the algorithm above, but almost entirely avoids the need for synchronization.

2.2 Fine-Grained Parallel Moves

In this strategy, 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 (non-colliding) set of moves and process them all in parallel,
2. Assign each core a partition in the placement 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.

[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.

2.2.1 *Independent Set Finding*

In the parallel moves algorithm described in [13], 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 non-colliding locations and nets, but is slower than the serial algorithm due to synchronization overhead [14]. Similarly, moves with non-colliding nets are proposed in [15] using a cell-colouring 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 [13, 15] or at best not serially equivalent [14].

The algorithm we present in Section 6 also finds independent move sets. It is serially equivalent and yields speedups of 2.2x on four cores.

2.2.2 *Partitioned Placements*

In this approach, 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 [14, 16] while others allow cells to be transferred to other partitions at any time [17, 18]. This method can produce excellent speedups; six cores linked by a LAN achieve a 5.3x speedup in [16] with no impact in wire quality. Other authors report speedups of 2-2.5x on four cores at a cost of slightly increased wirelength [17, 18]. These algorithms are all nondeterministic, and since the chip is always partitioned according to the number of cores, this approach cannot ever be serially equivalent. We do not explore this approach.

2.2.3 *Speculative Computation*

In this approach, first described in [19] 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 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 [19], but significant slowdowns are reported when applied to placement in [3]. 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 paper 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 Coarse-Grained Parallel Moves

By assigning different cores to completely different placements, more parallelism can be introduced. This technique is known as the parallel Markov chain [20]. 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 [14, 18].

Most authors have attempted to find the best speedup for similar quality of the serial version. Using four cores, speedups between 2.5x and 2.9x are reported for low to moderate wirelength increases [14, 18]. This technique cannot be serially equivalent without excessive runtime overhead and is nondeterministic if asynchronous updates are used. It is not further explored here.

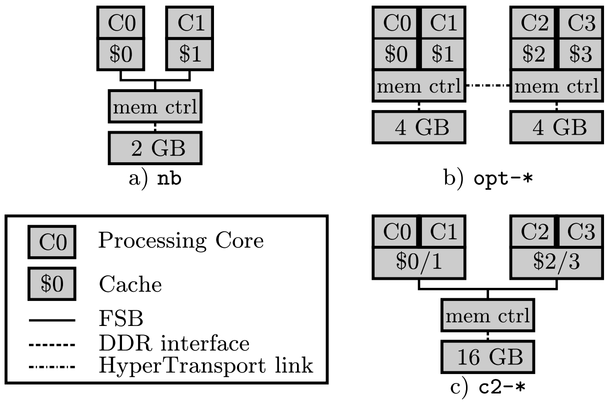


Figure 2: Hardware configurations.

2.4 Other Parallelization Strategies

More exotic parallel placement algorithms have been proposed, including hardware-assisted simulated annealing [21]. 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. EXECUTION ENVIRONMENT

3.1 Hardware Configurations

For the remainder of this paper, the term *processor* refers to a physical package on a board, whereas a *core* refers to a processing core. These terms match those used recently by the major processor manufacturers. For example, an Intel Core 2 Duo would be referred to here as a processor containing two cores, but a Pentium 4 with simultaneous multithreading (SMT, also known as HyperThreading) only contains one core. A dual-processor dual-core Opteron system contains a total of four processing cores in two packages.

Seven hardware configurations are used to test the algorithms in this paper. The first (Figure 2a) is an Intel Xeon (Netburst microarchitecture) single-core dual-processor system running at 2.66 GHz. Each core has a 512 KB L2 cache and the system has a total of 2 GB of memory. The front-side bus (FSB) has a peak bandwidth of 4.3 GB/s. The OS is Windows XP (32 bits). This configuration is called `nb`.

The next three configurations (Figure 2b) use an AMD Opteron 275 dual-core dual-processor system running at 2.2 GHz. Each core has a 1 MB L2 cache and the system has 8 GB of memory, connected in a non-uniform memory access (NUMA) configuration with 4 GB assigned to each processor. The HyperTransport link used to replace the FSB has a peak bandwidth of 8.0 GB/s. The OS is Windows XP (64-bit), though 32-bit executables are used. The `opt-dc` ("dual-core") configuration uses only C0 and C1; `opt-dp` ("dual processor") uses only C0 and C2; `opt-mc` ("multi-core") uses all four cores.

The last three configurations (Figure 2c) use an Intel Xeon 5140 (Core 2 microarchitecture) dual-core dual-processor system running at 3.0 GHz. Each of the processors shares a 4 MB L2 cache between its two cores. Two FSBs each run at 8.5 GB/s and are connected to a single 16 GB memory bank. The OS is identical to that of `opt-*`, and the configurations are similarly labelled `c2-dc`, `c2-dp` and `c2-mc`.

The algorithms in this paper were also tested on Linux using the 2.6 version of the kernel. Numeric results are not presented here but are comparable to those on Windows. In addition, we tried setting threads' affinities to individual cores to ensure that any poor thread scheduling by the OS was not affecting runtimes. We found, however, that this had no significant effect on the performance of our algorithms. Therefore, unless otherwise noted, all the algorithms presented here allow the threads to switch between cores at the discretion of the operating system.

3.2 Synchronization Overhead

While the wall-clock times for synchronization primitives such as conditional variables vary across the configurations listed above, we present some typical values here from `opt-dp` as an illustration.

A *critical section* is a block of code that can only be occupied by one thread at a time. When a thread wishes to enter an empty critical section it may do so immediately, but if another thread is already present it must stall until the first thread leaves. The time required to enter and leave an empty critical section is negligible. Critical sections are used very sparingly in our algorithms and hence are very small contributors to overall runtime.

Condition variables are used for direct synchronization between threads, and may be set to true or false. Upon encountering a condition variable that is true, the thread continues normally, but if it is false, a thread stalls until the variable is set to true by another thread. Since these variables are typically set to false, these stalls can be a significant contributor to total runtime. In our experience, threads take about $60\mu s$ to be restarted by the OS after stalling, which is comparable to the $90\mu s$ required to perform a single placement move. When we refer to "explicit synchronization" or "stalls" in this paper, we are referring to condition variables unless otherwise noted.

We briefly explored creating our own fast, lightweight synchronization primitives using spinlocks instead of system calls. However, there were three reasons we decided against using them. Firstly, faster synchronization would have had only a small impact on the performance of pipelined placement (see Section 5.2) and a negligible impact on that of parallel moves (see Section 6.2). Secondly, spinlocks are a form of busy waiting, and would therefore prevent cores from doing any useful work while threads were stalled. Thirdly, we would have had to tune the spinlocks for all the platforms supported by Quartus II, increasing the testing burden and reducing portability. For these reasons, we decided to delegate all synchronization to the operating system.

4. ATTRIBUTING RUNTIME

Parallel CAD algorithms rarely achieve ideal speedups, but we know of no prior work in parallel placement to quantitatively describe the factors which limit parallel performance. We developed the following method to illuminate these factors.

The centerpiece of this method is to create a serial flow which, as much as possible, mimics the runtime behaviour of the parallel flow. We call this new flow the *parallel-equivalent* (`peq`) flow. As an example, to create the `peq` flow for our pipelined algorithm (Section 5), we note that one thread speculatively proposes moves and places them into a buffer, while the other thread evaluates them. Therefore, the `peq` flow also proposes, buffers and evaluates moves, but does so in the same thread.

The `peq` flow is also instrumented to collect runtime information about different parts of the algorithm. We have used the high-precision timers provided by the Windows API to measure runtimes to a high degree of accuracy. For reasonable instrumentation, we have found their overhead to be very small (between 1-5%, depending on the configuration). We call all runtimes measured by these timers *bottom-up* measurements (abbreviated to `bu` in the figures in Sections 5.3.2 and 6.3.2). By contrast, any runtime measurement of the entire algorithm is called an *end-to-end* measurement (abbreviated to `e2e`).

The `peq` flow allows us to separate multi-core effects, such as inter-core memory latency and synchronization overhead, from more fundamental algorithmic limitations. In addition, it mimics the data access patterns of the parallel algorithm and therefore can reproduce some problems in the memory subsystem, such as decreased cache locality.

Our attribution method proceeds as follows:

1. Run the original, unmodified serial algorithm. Divide the total time by the number of moves run to obtain the end-to-end time per move.
2. Run the `peq` flow, set to mimic the serial flow as closely as possible. For example, in the pipelined placer's `peq` flow we set the number of speculative moves to zero, allowing only one to be "in-flight" at any time. We call this the `peq1` flow. We attribute any discrepancy between the original flow and `peq1` to the added code infrastructure required to run the parallel algorithm.
3. Run the `peq` flow, set to mimic the parallel flow as closely as possible. For example, if the algorithm has an average of x moves in-flight at a time, we allow the `peq` flow to do the same. We call this the `peqX` flow. If tasks that are identical in both `peq1` and `peqX` become slower, we attribute the difference to decreased memory subsystem efficiency, such as decreased cache locality. We also use the high-precision timers to measure any overhead induced by allowing a more realistic workload (such as move reproposals; see Sections 5.2 and 6.2).
4. Run the parallel flow. We directly instrument some multi-core effects, such as the time spent blocked in stalls. We also attribute slowdowns between `peqX` and the parallel flow to multi-core memory issues, such as bus contention or inter-core latencies such as cache snoops.

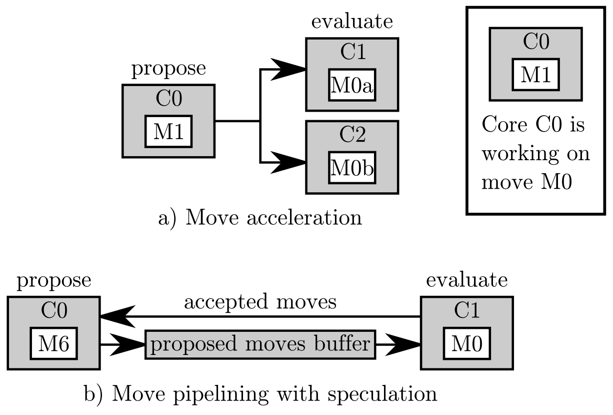


Figure 3: Accelerated vs. pipelined moves.

For `peq1` and `peqX`, the end-to-end and bottom-up measurements should match closely. Any large difference means the algorithm is not fully instrumented. For the parallel flow, we can divide all the bottom-up components by the ideal speedup; the sum of these adjusted times should also match the end-to-end runtime.

5. PIPELINED MOVES

5.1 Description

Our first parallel placer uses the pipelined aspects of the move acceleration strategy described in Section 2.1, but with several novel features to eliminate the explicit synchronization. As shown in Section 3.2, stalls' overheads are too high to achieve any speedup if they are common.

Figure 3 contrasts our pipelined move strategy with that of [13]. The basic idea is that if we can partition our runtime into two phases, the earlier shorter than the later, we could use an additional core to make the earlier phase "disappear." We want this imbalance because a perfectly balanced partitioning would require explicit synchronization between the stages; the cost is that the imbalance does limit the potential speedup. Our final partitioning assigns about 40% of a move's runtime to the earlier stage and 60% to the later. We refer to the earlier stage as the *proposal* stage, though some evaluation also occurs there, and the latter as the *evaluation* stage, which also accepts and rejects moves. With the critical path consisting of 60% of the serial runtime, we have a maximum speedup of about 1.7x.

There can be considerable variation in the runtimes of the two stages, so the assumption that the evaluation phase need never wait for the proposal phase does not always hold. In addition, at least in the pipelining scheme in Figure 3a [13], the proposal stage must still stall until the evaluation phase finishes the previous move. To solve both these problems, we allow the proposal stage to speculatively propose multiple moves and insert a buffer between the two stages (see Figure 3b). The proposal stage then races ahead of the evaluation, and variations in any one move's runtime does not cause the evaluation stage to stall.

5.2 Implementation

The method described above is not deterministic; the evaluation phase may change the placement at any time by accepting a move, which would lead to the proposal phase proposing different moves if they collide as described in Section 2.2.1. To resolve this problem, the database containing the locations of all the cells is duplicated between the two stages so that we can guarantee that the placement seen by the proposal stage never changes while proposing a move[2]. When a move is accepted, it is also sent back to the proposal stage to allow it to update its copy of the database.

This is still not deterministic. To enforce determinism, when a move is accepted, any speculatively proposed move that may collide with the accepted move must be reproposed[3]. If we ensure that the reproposed move is identical to the same move in the serial placer (a relatively simple matter), then the algorithm becomes not only deterministic, but serially equivalent.

Since not all moves collide, we can avoid reproposing those moves that are independent of the accepted move. We use a simple heuristic as our dependency checker. The proposal stage builds a list of all LAB locations it examines while proposing the move, and the evaluation stage remembers the last move to affect each location. It can then check whether any locations have been changed since the new move was first proposed. If such a collision is found, the move is reproposed prior to evaluation. Otherwise, it is not affected by previous moves and can be evaluated normally. For this algorithm, the rate of reproposals is quite low, on the order of 2% or less.

We experimentally found that allowing no more than six speculative moves to be in progress gave the best runtime. We need at least one speculative move to keep two processors busy, and our high synchronization overhead required one additional move. The remaining four moves are used to smooth out variations in the proposal and evaluation times. Note that using much faster synchronization (such as spinlocks, see Section 3.2) would only reduce the length of this queue by one move. The algorithm is not very sensitive to this parameter; values from about four to eight were reasonable. Too many moves could cause additional collisions and cache pollution, as is shown below.

We refer to this tuned version of the algorithm as `pipe7`, since it allows seven moves to exist at a time, one being evaluated and six being speculatively proposed.

5.3 Results

5.3.1 Overall Results

Figure 4 shows the geometric mean speedup of the pipelined iterative improvement algorithm on a set of 11 Stratix® II FPGA benchmark designs. These circuits are a collection of IP and customer circuits used internally for product development at Altera, and range from approximately 10k logic cells to 100k cells, the latter using the largest Stratix II device. We also tested a set of 40 Stratix III FPGA circuits and achieved very similar results.

The best configuration is `c2-dc` with an average 1.3x speedup (from 1.2x to 1.5x across circuits), and the worst

[2]This also prevents us from 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.

[3]It does not suffice to reject the move, as does the algorithm described in 2.2.1 [13], as a move's validity would then depend on exactly when the proposal stage received the last update to its location database.

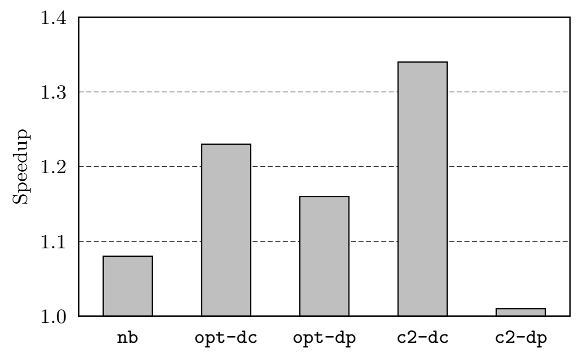


Figure 4: Pipelined moves speedups.

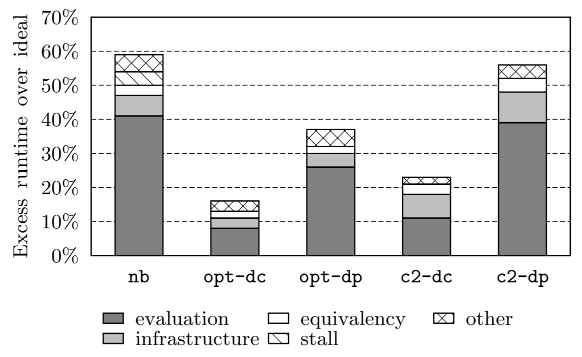


Figure 6: Pipelined moves overhead.

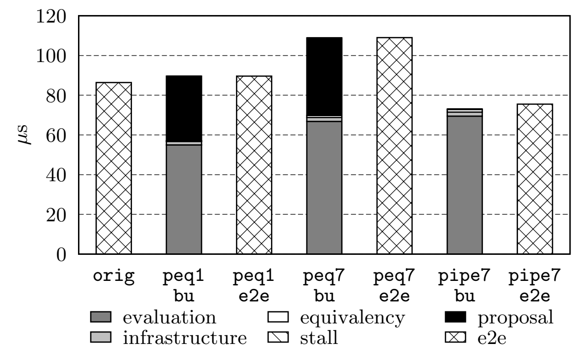


Figure 5: Pipelined moves attribution (opt-dp).

c2-dp with no speedup (from 0.9x to 1.2x), closely followed by nb. Clearly, the achieved speedup is strongly influenced by the platform architecture.

We also tested the effectiveness of the dependency checker by disabling it and simply re-proposed all moves that had been speculatively proposed after an earlier move was accepted. We found that this reduced the speedup by approximately 10%.

5.3.2 Attribution

Our best result of 1.3x is significantly lower than the ideal result of about 1.7x. Given the differences in the results for each configuration, and that a key difference between them is their memory subsystems, it is reasonable to assume that memory inefficiency is a major limiting factor. We used the procedure outlined in Section 4 to quantitatively determine the causes of performance loss versus the ideal speedup. Since seven moves are active in pipe7, we used peq7 to mimic the parallel flow. Our studies were conducted on a single circuit with a size of about 20k logic cells.

In peq1, we measured proposal and evaluation times. In peq7, we added reproposal times and dependency checks, and in pipe7 we added stall times. In addition, we attributed the difference between peq1 and the original flow to parallel infrastructure. Since the dependency checks and reproposal times turn out to be extremely small, we have combined them into an "equivalency" bin, representing all the time necessary to make the algorithm serially equivalent. The bottom-up and end-to-end measurements for all of these flows are presented in Figure 5 for opt-dp, and they correspond very well, indicating that the flows are well instrumented[4]. In the bottom-up pipe7 column, the proposal stage is not shown as it is assumed to be off the critical path.

While the infrastructure, equivalency and stall runtimes are very small, the evaluation runtime has a large jump between peq1 and peq7. As discussed in Section 4, this indicates that the memory inefficiency on this configuration is likely decreased cache locality. Since the evaluation runtime does not jump further between peq7 and pipe7, we conclude that inter-core latencies and bandwidth limitations are not a factor for opt-dp.

In Figure 6, the pipe7 overheads are presented for all two-core configurations[5]. The overheads are shown as a percentage of the "ideal runtime," which is defined as the runtime of the evaluation stage in peq1. Note that the ideal runtime is about 60% of serial, so an overhead of 66% would wipe out any gains ($60\% \times 166\% = 100\%$) . For the evaluation stage, the difference between pipe7 and peq1 is considered overhead, and the "other" bin accounts for the difference between the pipe7 bottom-up and end-to-end measurements.

Three results are significant. Firstly, for most configurations the time spent in stalls is so small as to be negligible; their large runtime penalty is avoided by reducing their frequency. Secondly, the overhead of the evaluation stage (8–40%, equivalent to 7–25% total runtime) accounts for most of the overhead on most configurations. Since the code executed by this stage is identical between the orig and pipe7 flows, this must be entirely due to a bottleneck in the memory subsystem. Thirdly, the overhead caused by maintaining serial equivalency is very small, less than 4% on all configurations (less than 3% runtime). This indicates that we would have very little to gain by dropping serial equivalency or determinism as requirements.

While there is no increase in evaluation runtime between peq7 and pipe7 on opt-dp, this is not the case on other configurations such as nb and c2-dp (not shown). This could be caused by inter-core latency, limited FSB bandwidth, or some combination of the two. We make some effort to quantify these causes in the next section.

[4]The bottom-up and end-to-end runtimes for the other two-core configurations are not shown here but are also well-instrumented.

[5]The results in Figure 4 do not line up perfectly with those in Figure 6, particularly for opt-dc and c2-dc. This is because Figure 4 shows an 11-circuit average while Figure 6 shows only one of those circuits.

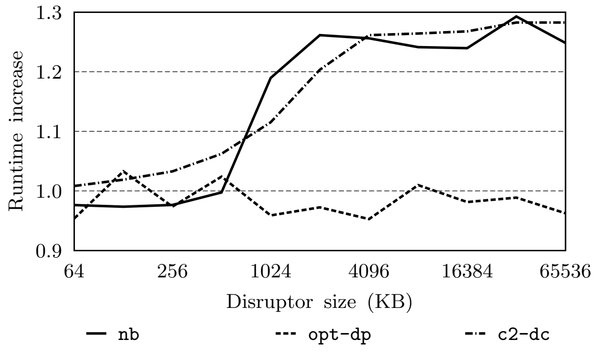
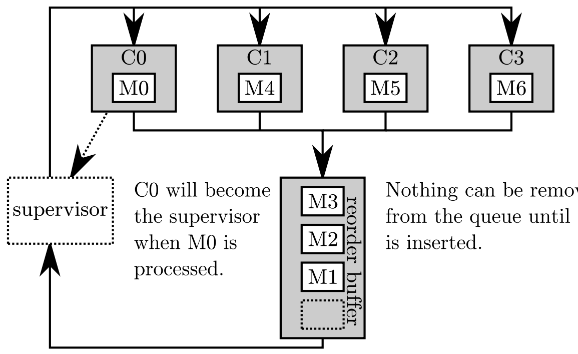


Figure 7: Effect of disruptor.



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.

Figure 8: Parallel moves with speculation.

5.3.3 *Further Study of Memory Inefficiencies*

We conducted additional tests to directly show the existence and nature of the memory inefficiencies by using a *disruptor thread.* A disruptor thread simply reads and writes to arrays of various sizes on one core while we run the `peq1` flow on the other[6]. The results are shown in Figure 7. On `nb`, runtimes of the `peq1` flow are not affected until the disruptor's size passes 512 KB, the size of its cache, at which point the flow's performance drops sharply but levels off above 2 MB. By contrast, on `c2-dc`, performance decreases until the disruptor fully uses the shared cache but levels off above that. This strongly suggests that bus contention is a main bottleneck for `nb`, and cache contention for `c2-dc`. The other three configurations (`opt-dp`, `opt-dc` and `c2-dp`, of which only the first is shown) show no effect at all from the disruptors, proving that they do not have a problem with raw bandwidth to main memory. The disruptor cannot measure inter-core latency, and this is likely a factor on all configurations except `opt-*`, as shown in Section 5.3.2.

Based on all of these results, we can draw some conclusions about the characteristics of the memory subsystems of the tested configurations. The shared cache in `c2-dc` appears to provide the best inter-core communication, though per-core efficiency drops as both cores attempt to use the same limited cache for their private working sets. The memory systems on `opt-*` provide the next-best performances, suffering only from reduced cache locality. The `nb` and `c2-dp` configurations are the slowest of all, suffering from reduced cache locality, inter-core latency and (for `nb`) bandwidth limitations.

Similar conclusions for these types of configurations have also been reported for other fine-grained algorithms [22].

5.4 Summary

Pipelined moves produce a speedup between nil and 1.3x, depending on the hardware configuration used, with the best results on the most recent architectures. Memory inefficiency is the largest performance bottleneck, comprising between 7% and 25% of total runtime. Some of this bottleneck is caused by cache pollution needed to keep the move buffer full. Serial equivalency cost less than 3% runtime.

While the algorithm could theoretically scale above two cores, in practice it is very difficult to partition the move runtime into more than two independent and balanced stages in our iterative improvement algorithm. Consequently, in the next section we investigate an alternative parallel approach, which will also largely solve the problem of cache pollution.

[6]For these experiments, both threads were locked to a specific core, to ensure the OS scheduler did not interfere with the results.

6. PARALLEL MOVES

6.1 Description

This algorithm partitions a moves runtime into two stages: *processing* and *finalization.* The processing stage consists of move proposals and evaluations, takes the vast majority of the runtime and occurs in parallel. Finalization consists of the dependency checker, reprocessing moves that have collided, and making the accept/reject decisions. It is relatively fast and occurs in serial.

The algorithm can be thought of as a master-worker configuration with the master finalizing moves. The workers can speculatively process moves so that they do not stall while the master finalizes earlier moves, and these moves are buffered as in the pipelined algorithm, with the addition that the buffer reorders them such that they are finalized in serial order. One can also view this algorithm as a limited implementation of a transactional memory system, where the modified locations in memory are manually tracked by the dependency checker.

This algorithm has two major advantages over pipelined placement. Firstly, assuming the finalization time is negligible, the ideal speedup increases from 1.7x to N, where N is the number of available cores. Secondly, a move is now processed in one step, entirely by one core, improving memory locality and eliminating the cache pollution from the pipelined algorithm caused by keeping a buffer full of active moves.

6.2 Implementation

As in the pipelined algorithm, we use a dependency checker to check for moves that collide due to sharing the same LAB location, though this occurs even less often than it did in parallel moves due to the shorter queue. However, since moves are now also being evaluated in parallel, their *costs* may collide as well, for example, if two moves affect the same net. While we could simply re-evaluate moves after a costs collision (as we re-propose them after a location collision), we found that there were too many such collisions to make this feasible. Therefore, we *always* re-evaluate moves if any collision is possible, but we also extended the dependency checker[7] to allow us to skip *portions* of the runtime-intensive cost components that were not affected by the collision.

Our initial, simplistic implementation of this algorithm with the master as a separate thread suffered from massive synchronization overhead. Since the finalization runtime is much smaller than the processing time, the master spent most of its time stalled and required one explicit synchronization per move to wake it up. Additionally, note that a lightweight, spinlock-based stall (as discussed in Section 3.2) as the master would not even be able to share a core with a worker thread, effectively preventing one core from doing any useful work.

Instead, our algorithm factors the master's tasks into a function known as the *supervisor*, which can be called by any thread after it has processed a move. Only one thread can "become" the supervisor at a time by calling this function, and it can only do so if the move buffer contains the next move in serial order. In Figure 8, when C0 completes M0, it will become the supervisor and finalize moves M0 to M3, and broadcast the results back to the other processors[8]. Note that the other cores are not stalled and are doing useful work. The concept of having a thread change roles was described in [22] to improve cache efficiency, but we use it mainly to avoid stalls. This change alone improved the performance of our algorithm by approximately 30% at two cores.

We found that allowing four in-flight moves per core gave the best results, though three was also reasonable. Fewer than three moves caused cores to become idle due to the wide variability in move times. Five or more moves had no impact on runtime, but since moves have a non-negligible memory footprint, we chose to limit the number of moves at the minimum necessary for good performance.

We wrote a new parallel-equivalent (`peq`) flow to match the new algorithm. In the new version, a single thread processes x moves. It then finalizes one move, processes a new move, and continues alternating one move at a time.

6.3 Results

6.3.1 Overall Results

Figure 9 summarizes the overall results and compares them to the pipelined algorithm. All results were collected as in Section 5.3.1. The `*-dp` and `*-mc` results are shown together, with `*-dp` used for two cores and `*-mc` for four.

The parallel moves algorithm achieves significant speedups. It outperforms pipelined placement on two cores, with an average speedup of 1.6x on `opt-*` (from 1.4x to 1.7x across circuits). In addition, it scales to four cores and achieves an average 2.2x speedup (from 1.6x to 2.8x) on `opt-mc`.

We again attempted to test the effectiveness of the dependency checker by simply reproposing all speculative moves after a move is accepted. In general, the dependency checker

[7]The exact resources tracked by the dependency checker vary depending on the cost. For example, if a small net counts towards a bounding box cost but a clock net does not, only the former will be tracked by the dependency checker. The programmer must manually track all resources used to evaluate a move, as the specific implementation of the costs determine exactly what must be tracked.

[8]The supervisor also reproposes and/or re-evaluates moves as necessary.

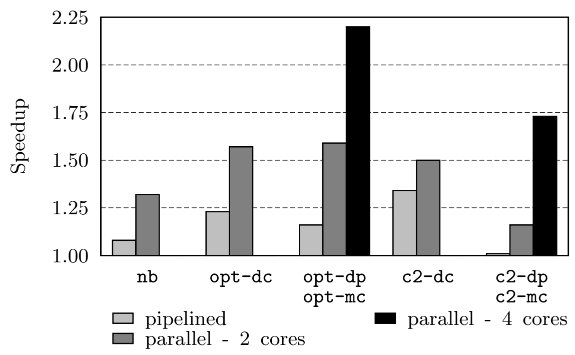


Figure 9: Speedups for parallel and pipelined moves.

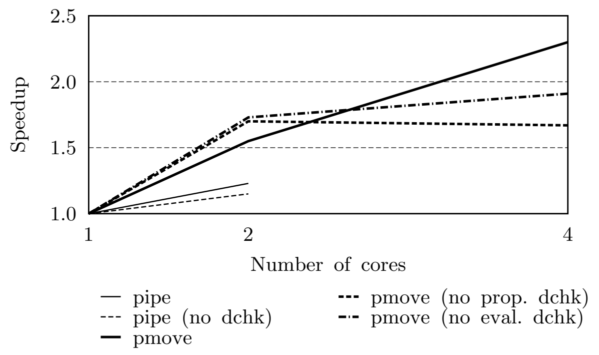


Figure 10: Various dependency checkers (`opt-dp`).

had little effect at two cores and was very helpful at four cores. The exception was on `opt-dp`, where the dependency checker decreased performance by 10% at two cores, though it is still helpful at four, where it increases performance by 40%. Figure 10 shows the effect of disabling various parts of the dependency checker on this configuration on both the pipelined and parallel moves algorithms. These results indicate that the dependency checker may have room for improvement.

6.3.2 Attribution

We performed attribution studies on the parallel moves algorithm as described in Section 4. The `prl8` label indicates that two cores processed up to eight in-flight moves. When running on two cores, we measured that the reorder buffer contained an average of one move, so we set the `peqX` flow to `peq3` (one move in the buffer plus two being processed).

In addition to the reproposal runtime, we also measured reevaluation and supervisor runtimes but combined them all into an "equivalency" measurement since they are all relatively small, reevaluation time being the largest. In Figure 11, we show the runtime components for `opt-dp`. The `prl8 bu/2` column presents identical information as in the `prl8 bu` column, but divided by the ideal 2x speedup. This correlates very well with the end-to-end measurements. The overheads are shown in Figure 12, with the "ideal runtime" defined as the `peq1` proposal and evaluation time.

Virtually no time is spent in stalls. For its part, serial equivalency causes an overhead of less than 5%, costing less than 3% of total runtime. Finally, we note no increase be-

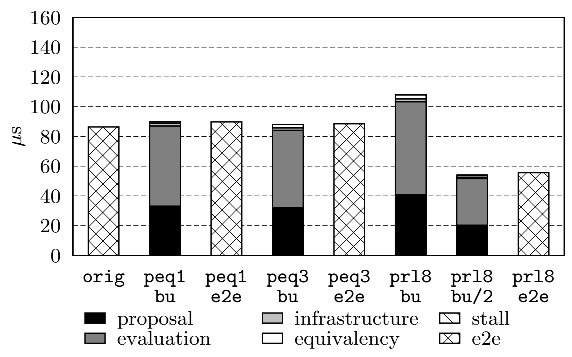


Figure 11: Parallel moves attribution (opt-dp).

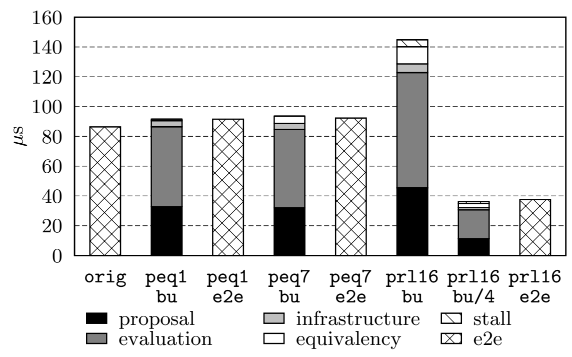


Figure 13: Parallel moves attribution (opt-mc).

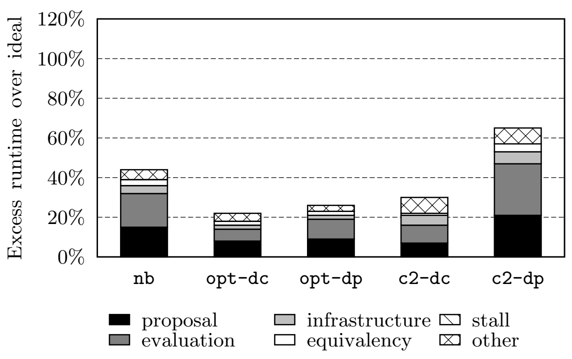


Figure 12: Parallel moves overhead – two cores.

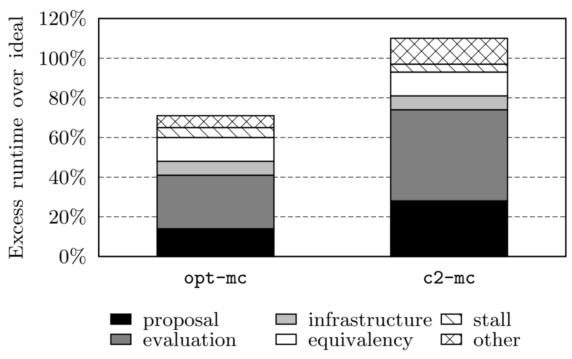


Figure 14: Parallel moves overhead – four cores.

tween peq1 and peq3 runtimes on opt-dp (similar results are reported on other configurations). This shows that, unlike in the pipelined algorithm, memory locality has not been degraded (for example, by cache pollution); all the latency is added by multi-core memory effects.

The improved speed-up over pipelined moves comes mainly from the higher ideal speedup of 2x. Memory overhead has also improved on some configurations. For example, on opt-dp, the runtime increase in the evaluation stage declined from 20% in the pipelined algorithm to 15% for parallel moves, and for overall processing declined from 23% to 16%. The most likely cause of this reduction is better cache locality due to the same core proposing and evaluating a move. However, for c2-dp there was no improvement, with an overhead of 50% in both the pipelined and parallel moves algorithms. We believe this is due to the inter-core bandwidth on the FSB being saturated, as it was with the disruptor tests on nb in Section 5.3.3. Overall, memory caused an overhead of 13–50%, equivalent to 10–30% of total runtime.

The "other" category is larger here than in the pipelined algorithm, though it is still 5% or less of total runtime on all configurations. This is likely caused by the overhead of critical sections and the overhead of the timers themselves.

The attributions for the four-core cases are shown in Figure 13 and Figure 14. The move buffer contained an average of 3 moves for a total of seven moves in flight, so we used peq7 as the peqX flow. Once again, memory performance is the main limiting factor and is significantly worse at four cores than at two. On c2-mc, the memory overhead increased from 15% at two cores (c2-dc, Figure 12) to 75% at four (c2-mc, Figure 14), equivalent to 35% of all runtime. On opt-mc, it increased from 15% at two cores (opt-dc) to 40% at four (opt-mc), or 25% of all runtime. The serial equivalency overhead is higher than at two cores at 12% (or 7% total runtime), but it is still relatively small.

6.4 Summary

Compared to pipelined moves, the parallel moves algorithm has better scalability and higher performance due to more inherent parallelism. Overhead due to memory is still the main bottleneck, though memory locality has improved and this helps performance on some configurations. Serial equivalency caused little overhead at two cores and somewhat more at four, but we believe this situation can be improved with more research into the dependency checker.

7. CONCLUSIONS AND FUTURE WORK

We have demonstrated two approaches to parallelizing a move-based iterative improvement placement algorithm: pipelined and parallel moves. We have improved on prior approaches by nearly eliminating synchronization overhead, and by using a dependency checker to minimize re-computation when we incorrectly speculate about the placement state. Pipelined placement achieved a significant speedup of up to 1.3x on two cores, but parallel moves achieved a larger speedup of 1.6x on two cores and 2.2x on four. In addition, both algorithms are deterministic and serially equivalent, greatly easing debugging and release testing and preserving the high quality of results of our existing placement algorithm at a cost of less than 3% of total runtime on two cores and 8% on four.

We presented a novel and systematic approach to measuring parallel runtime bottlenecks relative to the algorithm's ideal speedup. Interestingly, this procedure showed that the performance of our algorithms is restricted primarily not by insufficient parallelism but instead by the memory subsystem performance. Microprocessor vendors are aware that memory is a critical factor for multi-core systems, and have announced improvements in upcoming products. AMD's new quad-core Opteron line features a shared L3 cache, similar to the shared L2 cache on the Core 2 Duo, and Intel plans to replace the FSB in its future chipsets with a point-to-point communication system similar to AMD's HyperTransport system. Both of these advances will make fine-grained parallelism somewhat more efficient.

The fact that memory presents the largest limitation to parallel performance has major implications for algorithm developers. On architectures with a front-side bus, raw bandwidth to memory can be an issue, suggesting the designer should lower overall memory usage. On the other hand, core-to-core latencies are also a large factor, suggesting we duplicate as much state as possible across cores and only transmit updates between them. Obviously, these two solutions are contradictory, and further research will be required to determine when each technique is appropriate.

We have identified several probable enhancements to improve the efficiency of our parallel moves algorithm and plan to explore them in the future. With these refinements, and the expected hardware improvements to memory performance, we expect this algorithm to scale to eight or sixteen cores. More scalable algorithms will likely be required on larger systems.

8. REFERENCES

[1] H. Sutter, "A fundamental turn toward concurrency in software," *Dr. Dobb's J.*, Mar. 2005.

[2] Altera Corp., "Quartus II Incremental Compilation for Hierarchical & Team-Based Design," in *The Quartus II Handbook, Version 7.1, Vol. 1, Ch. 2*, 2007.

[3] J. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee, "An evaluation of parallel simulated annealing strategies with application to standard cell placement," *TCAD*, vol. 16, pp. 398–410, Apr. 1997.

[4] M. Sarrafzadeh, M. Wang, and X. Yang, *Modern Placement Techniques.* Boston: Kluwer Academic Publishers, 2003.

[5] C. Alpert, L. Hagen, and A. Kahng, "A hybrid multilevel/genetic approach for circuit partitioning," tech. rep., CS Dept., UCLA, Los Angeles, CA, USA, 1996.

[6] J. M. Kleinhans, G. Sigl, F. Johannes, and K. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *TCAD*, vol. 10, pp. 356–365, Mar. 1991.

[7] S. N. R. Borra, A. Muthukaruppan, S. Suresh, and V. Kamakoti, "A parallel genetic approach to the placement problem for field programmable gate arrays," in *IPDPS*, (Nice, France), p. 184, 2003.

[8] S. Kirkpatrick, C. Gelatt Jr., and M. Vecchi, "Optimization by simulated annealing," *Science*, pp. 671–680, May 1983.

[9] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *JSSC*, pp. 510–522, Apr. 1985.

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

[11] M. Hutton and V. Betz, "FPGA synthesis and physical design," in *Electronic Design Automation for Integrated Circuits Handbook* (L. Scheffer, L. Lavagno, and G. Martin, eds.), vol. 1, ch. 13, pp. 13.1–13.32, Taylor and Francis CRC Press, 2006.

[12] P. Chan and M. Schalg, "Parallel placement for field-programmable gate arrays," in *FPGA*, (Monterey, CA, USA), pp. 33–42, 2003.

[13] S. Kravitz and R. Rutenbar, "Placement by simulated annealing on a multiprocessor," *TCAD*, pp. 534–549, Jul. 1987.

[14] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Parallel algorithms for FPGA placement," in *GLSVLSI*, (Chicago, IL, USA), pp. 86–94, 2000.

[15] P. Banerjee, M. Jones, and J. Sargent, "Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors," *TPDS*, pp. 91–106, Jan. 1990.

[16] W. Sun and C. Sechen, "A loosely coupled parallel algorithm for standard cell placement," in *ICCAD*, (San Jose, CA, USA), pp. 137–144, 1994.

[17] S. Kim, J. A. Chandy, S. Parkes, B. Ramkumar, and P. Banerjee, "ProperPLACE: A portable parallel algorithm for standard cell placement," in *IPPS*, (Cancún, Mexico), pp. 932–941, 1994.

[18] J. Chandy and P. Banerjee, "Parallel simulated annealing strategies for VLSI cell placement," in *VLSID*, (Bangalore, India), pp. 37–42, 1996.

[19] E. Witte, R. Chamberlain, and M. Franklin, "Parallel simulated annealing using speculative computation," *TPDS*, vol. 2, pp. 483–494, Oct. 1991.

[20] E. Aarts, F. deBont, and E. Habers, "Parallel implementations of the statistical cooling algorithm," *Integration, the VLSI J.*, vol. 4, pp. 209–238, Sep. 1986.

[21] M. Wrighton and A. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *FPGA*, (Monterey, CA, USA), pp. 33–42, 2003.

[22] S. Vadlamani and S. Jenks, "Architectural considerations for efficient software execution on parallel microprocessors," in *IPDPS*, (Long Beach, CA, USA), 2007.