Parallelizing CAD: A Timely Research Agenda for EDA

Bryan Catanzaro
Department of Electrical Engineering and Computer Sciences
Berkeley, CA
catanzar@eecs.berkeley.edu

Kurt Keutzer
Department of Electrical Engineering and Computer Sciences
Berkeley, CA
keutzer@eecs.berkeley.edu

Bor-Yiing Su
Department of Electrical Engineering and Computer Sciences
Berkeley, CA
subrian@eecs.berkeley.edu

ABSTRACT

The relative decline of single-threaded processor performance, coupled with the ongoing shift towards on chip parallelism requires that CAD applications run efficiently on parallel microprocessors. We believe that an *ad hoc* approach to parallelizing CAD applications will not lead to satisfactory results: neither in terms of return on engineering investment nor in terms of the computational efficiency of end applications. Instead, we propose that a key area of CAD research is to identify the *design patterns* underlying CAD applications and then build CAD application frameworks that aid efficient parallel software implementations of these design patterns. Our initial results indicate that parallel patterns exist in a broad range of CAD problems. We believe that frameworks for these patterns will enable CAD to successfully capitalize on increased processor performance through parallelism.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming
D.2.11 [**Software Architectures**]: Patterns
J.6 [**Computer-aided Engineering**]: Computer-aided Design

General Terms

Algorithms, Design

Keywords

Pattern, Framework, Manycore, Parallelization

1. WHY MANYCORE PARALLELISM?

The decline of single-thread performance increases over the past half-decade has been well documented and understood [2]. Current and future performance increases will be provided primarily through increased on chip parallelism. Consequently, computationally demanding applications are moving towards parallel implementations. Movements towards parallelism in the past were always dominated by continual improvements in single thread performance, which made computational speedups due to parallelism less attractive, especially when viewed from an economic return on investment perspective. The realities of semiconductor manufacturing today have now led the entire industry towards parallelism in order to continue scaling application performance.

When discussing parallel software, it is useful to distinguish between *multicore* parallelism, designed for an evolutionary path where the number of cores slowly evolves from 1→2→4→.., and *manycore* parallelism, in which the ramp up of cores rises much more quickly: 32→64→128→. Current x86 microprocessors, such as the Intel Core 2 Duo, are an example of multicore platforms, whereas graphics processors, such as the G80 from Nvidia with 128 processors on a die, are an example of manycore processors, despite not being truly general purpose.

Superficially, it appears that CAD researchers must decide whether to pursue an implementation strategy that targets the multicore evolutionary path or manycore parallelism. However, parallel programmers commonly observe that programming for greater than 32 cores is fundamentally different than programming for smaller numbers of cores. Thus, as Moore's Law will easily enable 32 cores per die even on the slower multicore path, it seems important to target manycore targets today. Although parallelism at this scale can be difficult to extract, it will be necessary within the near future. Therefore, as we re-architect CAD software for parallelism, we must keep large scale parallelism in mind.

2. COARSE-GRAINED PARALLELISM IN CAD

Because CAD is so computationally demanding, parallelism has been applied to CAD for some time. A few basic approaches have broad application, such as: running a number of scripts and choosing the best result (logic synthesis); running a variety of different initial starting points (floorplanning, placement); running (or generating) independent vector sequences (simulation/testing) or Monte-Carlo simulation (capacitance extraction). It will be important to continue to exploit such coarse-grained parallelism in the future; however, because most of these approaches act on the entire design at once, it is not clear that memory and I/O capabilities associated with individual processors in multicore and manycore architectures will efficiently accommodate this kind of parallelism due to the heavy memory requirements associated with processing a complete design. Thus, while we will continue to exploit coarse grained parallelism at the system level, we must also seek ways to exploit finer-grained parallelism with a single processing element.

3. FINE-GRAINED PARALLELISM IN CAD

While it is easy to parallelize multiple independent runs of CAD software, it is less clear how to parallelize a single invocation of a CAD operation such as logic optimization, static-timing analysis, floorplanning, or placement. Past experience in high-performance computing as well as early attempts at parallelizing CAD applications have shown us that simply incrementally re-coding existing CAD software using POSIX Threads, OpenMP, or MPI is unlikely to produce acceptable results. Nor are there parallel programming languages or

DAC 2008, June 8–13, 2008, Anaheim, California, USA

programming models which are mature enough to allow CAD application developers to migrate their applications to parallel targets in a timely manner. Improved compiler support and implicit programming models [53] will surely help; however, we believe that such approaches will not suffice to fully utilize future manycore targets. Thus, our position is that CAD applications need to be re-architected for parallel targets. This does not mean that every significant element of CAD software needs to be recoded. Instead, as we will see in the example in Section 6, with the proper architectural structure, many key CAD algorithms, such as the battery of optimizations in logic synthesis, do not need to be rewritten. Before examining this in detail, we give a summary of the general principles of architecting parallel software.

4. PARALLEL SOFTWARE ARCHITECTURE USING PATTERNS AND FRAMEWORKS

As described by Christopher Alexander, design patterns describe time-tested solutions to recurring problems within a well-defined context [1]. An example of a design pattern is Alexander's "family of entrances" pattern, which addresses the recurrent problem of allowing easy comprehension of multiple entrances for a first-time visitor to a site. A pattern language is a collection of related and interlocking patterns, constructed such that the patterns flow into each other as the designer solves a design problem. The pattern language is implicitly organized to reflect a design methodology. Because every designer sees a given problem from a unique perspective, design patterns are not defined such that each problem has a unique path through the pattern language. Instead, a pattern language serves as a guide to the architect, helping to define common problems and codifying useful solutions. In other words, a pattern language does not impose a rigid methodology. Instead, it supports designers as they reason about an architecture by providing a common vocabulary to describe the problems encountered during the design process, as well as giving reference to a collection of useful solutions.

The observation that patterns and pattern languages could be useful for software architecture is not new; in fact there is an established community of researchers defining pattern languages for software architecture. While principally focused on lower-level implementation details, one influential work on patterns for software design is the book *Design Patterns: Elements of Reusable Object-Oriented Software*, which outlined a number of patterns useful for object-oriented programming [10]. Another work we have found useful is an *Introduction to Software Architecture* [11], which describes a variety of structural patterns useful for organizing software. *Patterns for Parallel Programming* [16] was the first attempt to systematize parallel programming using patterns. Recently, the Berkeley View reports [2][3] presented a set of 13 computational "dwarfs", which are best understood as design patterns for programming.

Patterns are *conceptual* tools which support a programmer's ability to reason about a software project, but they are not an implementation mechanism. A programming framework is a software environment that supports the *implementation* of the solution proposed by the associated design pattern. The difference between a programming framework and a general programming model or language is that in a pattern framework the customization is performed only at specified points that are harmonious with the style embodied in the original design pattern. An example of a successful sequential programming framework is the Ruby on Rails framework that is based on the Model-View-Controller pattern. Users of the framework have ample opportunity to customize the framework but only in harmony with the core Model-View-Controller pattern.

Frameworks include libraries, code generators, and runtime systems, which assist the programmer with implementation by abstracting difficult portions of the computation and incorporating them into the framework itself. In the past, parallel frameworks have not generally been designed by careful examination of recurrent problems in application domains, but instead were imposed by assumptions made by framework developers. We believe that basing frameworks on pervasive design patterns will make parallel frameworks more broadly applicable.

To summarize, implementing parallel software using patterns and frameworks involves identifying the patterns that underlie computations, constructing frameworks that assist in developing the parallelism in a given computation while mapping well to parallel hardware, and then using the frameworks to construct parallel applications. To apply this approach to CAD the first step is to identify the key design patterns in CAD. To be more specific about this step, the next section identifies the dominant design patterns encountered in a contemporary RTL synthesis CAD flow.

5. DESIGN PATTERNS AND FRAMEWORKS FOR CAD

5.1 Design Patterns in CAD

In the following we perform a reasonably comprehensive survey of the applications in a register-transfer level (RTL) synthesis flow, and summarize the key design patterns. Our methodology [2][3] is to first reduce an application to its key design patterns, and then use frameworks for the design pattern to facilitate the parallel implementation of the application.

5.1.1 Hardware-Description Language (HDL) Synthesis:

Graph Algorithms:

Given a description of a circuit in a HDL such as Verilog, perform a syntax-directed translation and graph-rewriting rules to create a circuit netlist. Note we refer to a *graph algorithms* pattern rather than a *graph traversal* pattern as in [2], because problems like graph partitioning are solved by graph algorithms but not graph traversal.

5.1.2 Technology Independent Optimization:

Graph Algorithms:

Contemporary approaches using And-Invert-Graphs are graph-oriented [45].

Backtrack and Branch-and-Bound:

The ESPRESSO family of algorithms [46] relies on branch-and-bound techniques to minimize two-level logic.

5.1.3 Technology Mapping:

Graph Algorithms:

Although they use dynamic programming, techniques such as DAGON [30] and the Lehman-Watanabe Algorithm [31] are probably better viewed as local graph rewriting approaches than general uses of the dynamic-programming design pattern.

5.1.4 Delay Modeling:

Dense and Sparse Linear Algebra:

Newton-Raphson methods can be used to solve the interdependent setup/hold time of latches and registers [44]. This method relies heavily on matrix computations.

5.1.5 Timing Analysis:

Graph Algorithms:

Computation for finding the critical path in a circuit is based on computing the longest path in a graph. The computation for identifying hold time violations is based on computing the shortest path in a graph. Statistical considerations can be added in this formulation.

Backtrack and Branch-and-Bound:

The elimination of false paths and the computation of the floating-mode delay [33] require a search using techniques similar to those employed in circuit testing (cf. 5.1.14).

Sparse Linear Algebra:

Path-based statistical static timing analysis can be formulated as a sparse matrix problem [32].

5.1.6 Sequential Circuit Optimization:

Graph Algorithms:

Retiming involves constructing the retiming graph and manipulating it using graph algorithms [29]. Clock schedule optimization may be solved by traversing the circuit netlist, updating clock information [34].

5.1.7 Floorplanning:

Graph Algorithms:

The sequence pair method [23] transforms the sequence pair into a constraint graph, and then reconstructs the floorplanning topology. B*-tree [6] and TCG-S [14] both manipulate graph representations of the floorplanning topology directly.

Map Reduce:

Simulated annealing approaches to floorplanning, such as [47], can be cast as Map Reduce operations.

5.1.8 Placement:

Dense Linear Algebra:

Most analytical placers use some iterative nonlinear objective solvers to optimize the objective function. NTUplace3 [7] and APlace [12] use the conjugate gradient method, while mPL6 [5] uses the Uzawa iterative algorithm.

Graph Algorithms:

There are several placers that use graph partitioning techniques [48] to resolve the overlapping problem among cells. For example, Capo [17], Dragon [20], and GORDIAN [13].

Sparse Linear Algebra:

Some placers use sparse quadratic programming methods to minimize the wire length. RQL [21], Kraftwerk[19], FastPlace3 [22], and GORDIAN [13] all apply this method.

Structured Grid:

The FastPlace3 [22] uses iterative local refinement technique to refine the placement in a grid by the placement situation in nearby grid.

5.1.9 Global and Detailed Routing:

Graph Algorithms:

For Steiner tree construction, FLUTE [9] traverses each net, partitioned by its coordinates, and then maps each partition to a pre-optimized routing table. For global routing, FastRouter2.0 [18] and BoxRouter2.0 [8] both use rip-up-and-reroute with maze routing to refine the routing results. DUNE is a gridless detail router that manipulates the connection graph [24].

Dynamic Programming:

During the rip-up-and-reroute phase, FastRoute2.0 [18] uses monotonic routing, and DpRouter [4] uses dynamic pattern routing. Both are dynamic programming based approaches.

Backtrack and Branch-and-Bound:

BoxRouter2.0 [8] uses integer programming to assign wires into available spaces.

5.1.10 DFM

Graph Algorithms:

To detect and correct alternating-aperture phase shift masking problems Chiang el al. [27] use a conflict cyclic graph and an embedded planar graph to represent the layout topology.

Map Reduce:

RADAR [26] uses a lithography hotspot map (LHM) to approximate the intensity of OPC at different regions. When calculating the LHM, RADAR transforms lithography intensity at nearby blocks, collects all information, then it reduces them to the LHM of the specified region.

5.1.11 Design Rule Checking and Compaction:

Graph Algorithms:

The scanline algorithm [25] traverses the layout of the circuit based on the coordinates of the cells. The resulting design rule constraint graph may be compacted using longest path graph traversal [49].

5.1.12 Model Checking:

Graph Algorithms and Backtrack/Branch-and-Bound:

Symbolic model checkers, such as NuSMV2 [41], provide both BDD-based and SAT-based approaches to solve the problem.

5.1.13 Equivalence Checking:

Graph Algorithms:

Malik *et al.* propose a BDD-based approach [40] for equivalence checking.

Backtrack and Branch-and-Bound:

Goldberg *et al.* use a SAT-based approach [39] for equivalence checking. Contemporary approaches judiciously apply both BDD and backtracking-based techniques.

5.1.14 ATPG:

Backtrack and Branch-and-Bound:

PODEM [35] was among the first to use backtrack and branch-and-bound techniques to generate test patterns, and Larrabee uses a SAT-based approach [50].

5.1.15 Delay Testing:

Graph Algorithms:

The hazard-free robust path delay fault testing method [38] traverses the circuit graph, generates its ENF, and then finds testing vectors.

5.1.16 HDL Simulation:

Graph Algorithms:

Parallel HDL simulation has already been explored using agents in a graph structure [51].

5.1.17 Circuit Simulation:

Dense Linear Algebra and Sparse Linear Algebra:

SPICE [43] uses linear algebra to solve the Kirchoff voltage and current laws, branch equations, nonlinear equations, and differential equations involved in circuit simulation.

5.2 CRITICAL PATTERNS AND FRAMEWORKS

The prior section indicates that much of CAD can be supported by three key design patterns: graph algorithms, branch-and-bound search, and linear algebra. Linear algebra is the design pattern whose parallelism is best understood [54], so we will not discuss it further. Branch and bound algorithms focus on the search of very large spaces, so parallel frameworks such as [52] have been developed to accelerate the search process. The load-balancing and data sharing problems encountered by previous attempts at such frameworks will be alleviated by the high degree of integration which characterizes future manycore systems, enhancing the usefulness of parallel branch-and-bound frameworks. The graph algorithms pattern is by far the most

widely used pattern in CAD. Unfortunately, it is currently relatively weakly supported by parallel frameworks [55]. Therefore, the rest of this paper explores the potential for developing a parallel graph-algorithms framework that can be applied to parallelize CAD. In particular, we explore how the concepts of Section 5 can be employed to build a parallel framework for logic optimization.

6. A PARALLEL FRAMEWORK FOR LOGIC OPTIMIZATION

In this section we work through a high-level architecture for a parallel logic-optimization framework. At the top level, the framework follows the pipe-and-filter structural pattern [11] as is shown in Figure 1. For simplicity's sake we presume the input is a textual description of the netlist to be optimized together with user constraints regarding timing, area, and power. This textual format is parsed, an internal database is built, and then the core function of the framework, the actual logic optimization process, is performed. Finally, an optimized netlist is produced. Very modest amounts of parallelism are available at the steps of input processing, database building, and output; however, we presume that some parallelism is available at these steps and, further, that the time of these steps is modest relative to the time devoted to optimization.

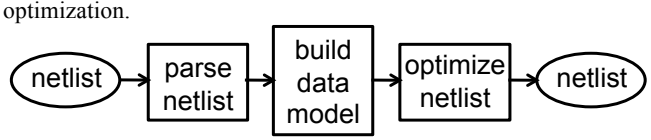

Figure 1: A Basic Flow

The representation of the circuit in the database/repository is the key to realizing significant parallelism. At the core of the system is an agent-and-repository structural pattern [11] illustrated in Figure 2. In this pattern the access of independent agents to the central repository is scheduled by a repository manager. In general, to provide fine-grained parallelism for graph manipulation tasks such as logic optimization, the graph must be partitioned. The first step toward creating this partitioning is to divide the netlist into register-bounded sub-circuits. Partitioning the netlist in this fashion allows the timing optimization agents to operate independently, which maximizes parallelism of the optimization. However, there are many engineering challenges to developing a register-bounded partitioning of a circuit such as: multi-cycle paths, latch-based timing schemes, multi-modal operation, and tri-state enables. As a result of these influences the number of gates in a single register-bounded sub-circuit may grow too large to be useful. Thus we have to further partition the netlist. Partitioning circuits in a way that achieves high load-balancing and low synchronization overhead is an open problem for CAD. In fact, we would argue that it is *the critical problem* in finding fine-grained concurrency in CAD.

Currently, we envision two approaches to solve this problem. The first approach is to quantify the interdependency among regions, and then partition the circuit into disjoint sub-circuits of similar size while minimizing the interdependency objective. Disjoint partitioning of the circuit allows for more parallelism, but will reduce quality of results. The second approach is to define slightly overlapping partitions of the circuit, similar to the "ghost regions" in structured grid problems arising in computational physics problems. Overlapping partitions overcome the sub-optimality imposed by rigid partitioning, although they require more memory storage and more careful updates among

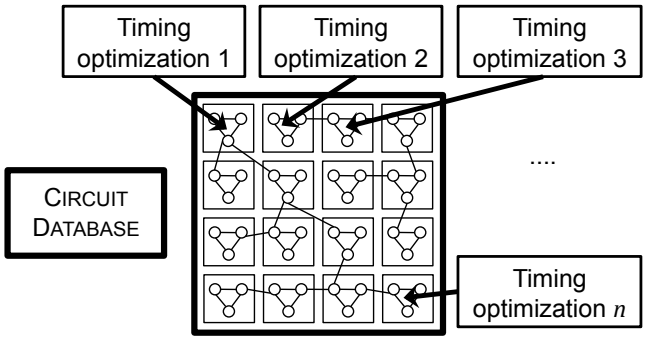

Figure 2: Timing Agents Acting on Repository

overlapping partitions. In summary, to achieve fine-grained parallelism we must exploit the data parallelism latent in the circuit. More research needs to be done to find practical methods for revealing this parallelism.

The next point to consider in evaluating the potential for a manycore implementation of logic optimization is the memory footprint of an individual node in the netlist. Depending on the sub-problem in logic optimization, such a footprint can be as small as 4 bytes for technology independent logic optimization or as large as 4K bytes in an integrated placement-synthesis system. It is useful to examine this data in the context of micro-architectural trends. Near future *multicore* architectures give each core a traditional L1 & L2 cache, and then provide a large shared L3 or Last Level Cache. L1 sizes are usually 32 to 64 kB, while L2 caches are usually around 256 to 512 kB. Shared L3 caches range from 2-8 MB on die. In contrast, it seems likely tha*t manycore* architectures will favor more cores on a chip and simpler memory hierarchies. It may be that the L1 remains around 32kB, and the shared Last Level Cache remains several megabytes, while private L2 caches are omitted completely. To make up for these simpler memory hierarchies, manycore architectures will utilize high bandwidth off chip memories, perhaps even a cache composed of stacked DRAM in the chip package (or even more radical 3D integration approaches) to provide substantial memory capacity at very high bandwidth. Additionally, the integration provided by having all the processors and memory integrated so closely will allow for much cheaper data sharing between processors, which should enable less duplication of shared data in the caches.

For example, if the netlist node size is 4 bytes, a 32 kB cache can hold 8k nodes, and sizeable computations can be done very effectively in local L1. However, if the node size is 4k bytes, most of the work will have to be done out of Last Level Cache and off-chip memory, which will reduce performance. Memory usage and inter-processor bandwidth will be an important factor in determining the granularity of partitioning used in manycore logic synthesis.

Parallel frameworks must be implemented well in order to be useful. For example, the engineering of the agent-and-repository framework that houses the circuit database and schedules the timing optimization agents will be critical to realizing parallel performance. Such an agent-and-repository framework has two difficult invariants that must be maintained: 1) No agent corrupts the logical functionality of the original circuit and 2) Each agent may operate independently as though it were the only agent operating on this sub-circuit. The first invariant ensures overall correct functionality, and enforcing this invariant will significantly simplify debugging the concurrent agents. The

second invariant dramatically simplifies the construction of the timing optimization agents. Thus, despite the many challenges to engineering such a framework, the investment in this engineering pays off immediately: *the individual logic optimization agents themselves do not need to be recoded from the original sequential case*.

While the architecture of this framework may appear to be cumbersome, we believe that it nicely showcases the key elements of the design. By bringing out the parallelism in the underlying problem, we entirely obviate the need to recode the individual logic optimization agents themselves. We believe that an overall design approach that localizes managing concurrency to a few critical pieces, such as the database/repository, will be more successful than an approach that demands that parallel programming concerns pervade the design.

7. CONCLUSION

Single-chip multiprocessors are sure to impact computationally intensive application areas such as CAD. We argue that we should *not* focus on near-term opportunities to realize modest amounts of parallelism, but boldly focus on microprocessor targets with greater than 32 processing elements.

When facing this challenge it becomes clear that incremental approaches based on thread-based programming or compiler optimization improvements will be inadequate to fully realize the performance afforded by highly parallel machines. Instead, we argue that CAD software must be re-architected to discover and express high degrees of parallelism. Following [2][3], we argue that the key to architecting parallel software is identifying parallel design patterns (known as dwarfs), and to this end we comprehensively mined the RTL synthesis flow to identify the recurrent design patterns in that flow. Despite the software complexity and algorithmic density in that flow, our review showed three design patterns predominate: graph-algorithms, backtracking/branch-and-bound, and linear algebra. Of these the most challenging design pattern to parallelize is graph algorithms. To explore this topic further we described a high-level architecture for a logic optimization framework employing graph algorithms.

The development of graph partitioning algorithms that minimize interdependence between partitions is a pivotal problem, and the success of a parallel logic optimization framework rests heavily on it. Nevertheless, as netlists are highly data parallel and relatively localized, we remain optimistic that such partitioning approaches will be found, and that with a proper architecting of a central agent-and-repository database the bulk of the code associated with the individual logic optimization agents can remain unchanged. As few parallel programming experts currently exist, we believe a strategy that centralizes and localizes the burden of managing concurrency will be more successful than one that distributes the burden of parallel programming over the entire software development workforce.

At the center of our inquiry is a simple question: Is CAD amenable to parallelization on manycore processors? We are optimistic. An integrated circuit is a static data object with millions of elements and is amenable to representation as a graph with low average out-degree and high locality. Extracting parallelism here should be easier than in many media applications, such as video decoding, which deal with streams of changing data with complex and unpredictable interdependencies. In short, we believe there are ample opportunities for discovering parallelism in CAD applications. Finding and demonstrating them on parallel computers is a timely research agenda.

8. ACKNOWLEDGEMENTS

Thanks to Jacob Avidan, Alan Mishchenko, Richard Rudell, and Tom Spyrou for discussion on various topics related to the logic optimization framework described here. Thanks to Patrick Groeneveld, Desmond Kirkpatrick, Noel Menezes, Sachin Sapatnekar and Leon Stok for critical comments that improved this paper. Tim Mattson deserves special thanks for helping us to understand dwarfs in the larger contexts of design patterns and pattern languages. The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

9. REFERENCES

[1] C. Alexander *et al*. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, USA, 1977.

[2] K. Asanovic *et al*. The landscape of parallel computing research: a view from Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[3] K. Asanovic *et al*. The landscape of parallel computing research: a view from Berkeley 2.0. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2007.

[4] Z. Cao *et al*. Dprouter: A fast and accurate dynamic-pattern-based global routing algorithm. ASPDAC '07:, pp. 256-261, 2007.

[5] T. F. Chan *et al*. MPL6: a robust multilevel mixed-size placement engine. ISPD '05, pp. 227-229, 2005.

[6] T.-C. Chen, Y.-W. Chang, and S.-C. Lin. Imf: interconnect-driven multilevel floorplanning for large-scale building-module designs. ICCAD '05, pp. 159-164, 2005.

[7] T.-C. Chen *et al*. A high-quality mixed-size analytical placer considering preplaced blocks and density constraints. ICCAD '06, pp. 187-192, 2006.

[8] M. Cho *et al*. Boxrouter 2.0: architecture and implementation of a hybrid and robust global router. ICCAD '07, pp. 503-508, 2007.

[9] C. Chu and Y.-C. Wong. Fast and accurate rectilinear steiner minimal tree algorithm for vlsi design. ISPD '05: pp. 28-35, 2005.

[10] E. Gamma *et al*. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, USA, 1994.

[11] D. Garlan and M. Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[12] A. B. Kahng and Q. Wang. A faster implementation of aplace. ISPD '06, pp. 218-220, New York, NY, USA, 2006.

[13] M. Kleinhans *et al.* VLSI placement by quadratic programming and slicing optimization. IEEE Trans. on CAD, 10(3): 356-365, 1991.

[14] J.-M. Lin and Y.-W. Chang. Tcg-s: orthogonal coupling of p*-admissible representations for general floorplans. DAC '02, pp. 842-847, 2002.

[15] S. MacDonald *et al*. From patterns to frameworks to parallel programs. Parallel Computation, 28(12):1663-1683, 2002.

[16] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, USA, 2004.

[17] A. N. Ng *et al*. Solving hard instances of floorplacement. ISPD '06: pp.170-177, 2006.

[18] M. Pan and C. Chu. Fastroute 2.0: A high-quality and efficient global router. Asia South Pacific-DAC '07: pp. 250-255, 2007.

[19] P. Spindler and F. M. Johannes. Fast and robust quadratic placement combined with an exact linear net model. In ICCAD '06: pp. 179-186, 2006.

[20] T. Taghavi, *et al.* Dragon2006: blockage-aware congestion-controlling mixed-size placer. ISPD '06, pp. 209-211, 2006.

[21] N. Viswanathan *et al*. Rql: global placement via relaxed quadratic spreading and linearization. DAC '07, pp. 453-458, 2007.

[22] N. Viswanathan, M. Pan, and C. Chu. Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. Asia South Pacific-DAC '07, pp. 135-140, 2007.

[23] H. Murata *et al*. VLSI Module Placement Based on Rectangle-Packing by the Sequence Pair, IEEE Trans.on CAD 15(12), pp. 1518-1524, 1996.

[24] J. Cong, J. Fang, and K. Khoo. DUNE: a multi-layer gridless routing system with wire planning. ISPD'00, pp.12-18, 2000.

[25] E. Carlson, R. Rutenbar. A Scanline Data Structure Processor for VLSI Geometry Checking, IEEE Trans. on CAD 6(5), pp. 780-794, September 1987

[26] J. Mitra, P. Yu, D. Pan. RADAR: RET-aware detailed routing using fast lithography simulations, DAC'05, pp. 369-372, 2005.

[27] C. Chiang *et al.* Fast and efficient phase conflict detection and correction in standard-cell layouts. ICCAD'05, pp. 149-156, 2005.

[28] N. Saluja and S. Khatri. A robust algorithm for approximate compatible observability don't care (CODC) computation. DAC'04, pp 7-11, 2004.

[29] C. Leiserson and J. Saxe. Retiming Synchronous Circuitry, Algorithmica, vol.6, pp.5-35, 1991.

[30] K. Keutzer. DAGON: technology binding and local optimization by DAG matching. DAC'87, pp. 341-347, 1987.

[31] E. Lehman *et al.* Logic decomposition during technology mapping, IEEE Trans. on CAD 16(8):813-834, 1997.

[32] A. Ramalingam *et al.* An accurate sparse matrix based framework for statistical static timing analysis. DAC'06, pp. 231-236, 2006.

[33] S. Devadas, *et al.* Computation of floating mode delay in combinational circuits: practice and implementation, IEEE Trans. on CAD 12(12):1924-1936, 1993.

[34] N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli. Graph algorithms for clock schedule optimization. ICCAD'92, pp. 132-136, 1992.

[35] P. Goel and B. Rosales, PODEM-X: An automatic test generation system for VLSI logic structures. DAC'81, 1981.

[36] H. Fujiwara. FAN: A Fanout-Oriented Test Pattern Generation Algorithm, Proc. Int'l Symp. Circuits and Systems, pp. 671-674, 1985.

[37] S.-T. Cheng, R. Brayton. Synthesizing multi-phase HDL programs, Verilog HDL Conference, pp. 67-76, 1996.

[38] S. Devadas, K. Keutzer, S. Malik, Delay computation in combinational logic circuits: theory and algorithms, ICCAD'91, pp. 176-179, 1991.

[39] E. Goldberg, M. Prasad, and R. Brayton. Using SAT for combinational equivalence checking. DATE'01, pp. 114-121, 2001.

[40] S. Malik *et al.* Logic Verification Using Binary-Decision Diagrams in a Logic Synthesis Environment. ICCAD'88, pp. 6-9, 1988.

[41] A. Cimatti, *et al.* NuSMV 2: An OpenSource Tool for Symbolic Model Checking, CAV'02, pp. 359-364, 2002.

[42] A. Kolbi, J. Kukula, R. Damiano, Symbolic RTL simulation, DAC'01, pp. 47-52, 2001.

[43] L. Nagel. SPICE2: A Computer Program to Simulate Semiconductor Circuits, Memorandum No. ERL-M520, University of California, Berkeley, May 1975.

[44] S. Srivastava and J. Roychowdhury. Interdependent Latch Setup/Hold Time Characterization via Euler-Newton Curve Tracing on State-Transition Equations, DAC'07, pp. 136-141 , 2007.

[45] A. Mishchenko, S. Chatterjee and R. Brayton, DAG-aware AIG rewriting: a fresh look at combinatorial logic synthesis, DAC'06, pp. 532-535, 2006.

[46] R. Brayton *et al*. Logic Minimization Algorithms for VLSI Synthesis, Kluwer, Boston, 1984.

[47] T.-C. Chen and Y.-W. Chang. Modern floorplanning based on fast simulated annealing, ISPD'05, pp. 104-112, 2005.

[48] C. Fiduccia, R. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions, DAC'82, pp. 175-181, 1982.

[49] M. Hsueh. Symbolic layout and compaction, PhD Thesis, University of California, Berkeley, 1980.

[50] T. Larrabee. Test pattern generation using Boolean satisfiability, IEEE TCAD, 11, 1, pp. 4 - 15, 1992.

[51] V. Krishnaswamy, P. Banerjee. Actor Based Parallel VHDL Simulation Using Time Warp, Parallel and Distributed Simulation 1996, pp. 135-142.

[52] A. Stam. A Framework for Coordinating Parallel Branch and Bound Algorithms, LNCS Coordination Models and Languages, pp. 523-532, 2002.

[53] W.-M. Hwu *et al*. Implicitly parallel programming models for thousand-core microprocessors, DAC'07, pp.754 – 75.

[54] Anderson, E., *et al.,* 1999 LAPACK Users' Guide (Third Ed.). Society for Industrial and Applied Mathematics.

[55] Lumsdaine, A., *et al.,* Challenges in Parallel Graph Processing. Parallel Processing Letters, 17(1):5--20, 2007.