

# A Distributed Optimal Scheduler Without ILP Overhead

Stephen Blythe  
Department of Computer Science  
Southern Illinois University - Edwardsville  
Edwardsville, IL 62026 - 1656  
sblythe@siue.edu

## Abstract

This paper presents an alternative to ILP based schedulers that still guarantees the discovery of valid design points. The proposed methodology is shown to improve on the run time behavior of contemporary ILP solutions, and a distributed version of the methodology is described. Details of this methodology are then presented, along with examples that show its strengths and weaknesses. Empirical evidence is given to demonstrate the methodology's success.

**Keywords:** Distributed Computing, Electronic Design Automation, Scheduling, Optimization

## 1. Introduction

The field of High-Level Synthesis (HLS) has become increasingly important in the design automation of electronic systems over the past several years. Often referred to as *silicon compilation* [8], design automation refers to the translation of a behavioral description such as a source code program or a dataflow graph into an equivalent electronic circuit (see Figure 1). High-Level Synthesis refers to an abstraction that breaks the design automation problem into more manageable high-level procedures such as scheduling dataflow operations into clocked control steps, allocation of appropriate hardware resources to the overall design, mapping operations to specific resources, and many other problems. An example of scheduling a dataflow graph is given in Figure 2. This example uses the dataflow graph found in Figure 1 and shows a schedule that utilizes 6 control steps and has a resource allocation of 3 multipliers, 1 ALU (Arithmetic Logic Unit, capable of addition, subtraction, and comparison operations), and 1 unit dedicated to addition. Note that more computationally expensive modules (i.e. resources) are often spread across multiple control steps, a process known as *multi-cycling*.

An interesting problem that High-Level Synthesis must solve is thus the following, often referred

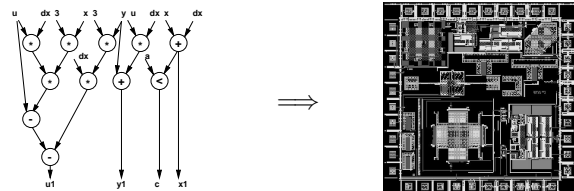


Figure 1: The Basic HLS Problem.

to as the *scheduling* problem: Given a resource allocation (number of modules of every given type) and a schedule length (number of control steps), can all the operations within a behavioral description be placed into a (set of) control steps without exceeding resource allocation in any given control step while still enforcing all data dependencies found in the behavioral description? Unfortunately, this problem is known to be NP-complete, so much prior work has been on attempting to find good heuristics for the scheduling problem[3][12]. Unfortunately, such techniques cannot guarantee finding a solution. However, it has long been one of the primary goals of High-Level Synthesis and design automation to produce optimal high level designs. Only relatively recently has the problem been solved optimally[9]. Much of this work has been built around Integer Linear Programming (ILP) based solutions [5]; unfortunately these solutions can become unbearably time consuming[2]. They are, after all, solving an NP-complete problem (scheduling) by rewriting it as another NP-complete problem (Integer Programming) and then solving that problem. Furthermore, ILP solvers that are viable for scheduling problems other than the smallest in size tend to be expensive commercial products such as AMPL/CPLEX[7] <sup>1</sup>.

An approach that is often employed when solving such large scale problems is that of distributed systems, in which many loosely coupled worksta-

<sup>1</sup>A single *educational* license for AMPL/CPLEX costs approximately \$1500(US); other types of licenses, such as commercial versions, cost significantly more.

tions are utilized to solve a single large scale problem. Unfortunately, the expense of commercial ILP solvers does not usually allow distributed processing solutions, as each node in a distributed system that solves ILP problems requires a (legal) copy of and/or license for the commercial ILP solver. In lieu of this, our methodology utilizes a customized scheduler that can be easily distributed to a theoretically unlimited number of workstations with no licensing fees at any node.

The work in this paper outlines an optimal scheduler that utilizes the well structured format underlying typical ILP formulations and maps them to an optimal exploration methodology whose performance represents a significant enhancement over ILP based run-times as discussed in Section 2. The methodology is then extended by distributing the workload across numerous workstations. The process for generating this distribution is presented in Section 3, along with a discussion of what would make a good distribution in this context. Further empirical results demonstrating that the developed exploration methodology runs efficiently and scales well to larger problems is presented in Section 4.

## 2. The Basic Scheduler

Before considering the distributed version of the scheduler, it is important to understand the nature of the equivalent non-distributed scheduler. While this scheduler is derived from existing ILP formulations of the scheduling problem, its actual methodology in finding solutions represents a more efficient search of the corresponding potential solution space. Specifically, the formulation represents potential schedules through a binary vector, with a unique location in the vector for each possible control step that each operation in an input behavioral description could be scheduled into<sup>2</sup>. Should an element in this vector be 1, its corresponding operation is to be scheduled into the corresponding control step; otherwise (i.e. if the element is

<sup>2</sup>Some ILP representations use positive integer ranges for elements instead of binary values; such formulations attain similar run-time performance to the binary vector approach.

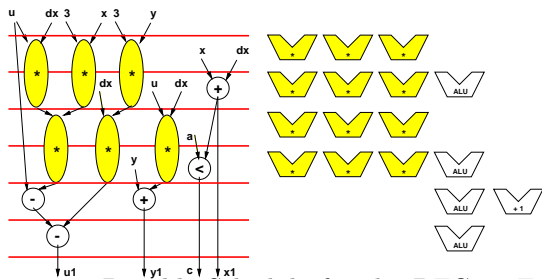


Figure 2: Possible Schedule for the DFG in Figure 1.

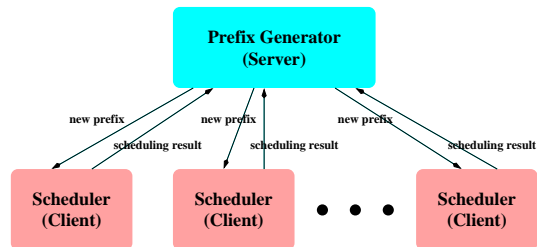


Figure 3: Distributed Scheduler Architecture

0), the operation is not to be scheduled into the corresponding control step.

In such models, the goal is then to find a binary vector that satisfies the scheduling constraints. For the scheduling problem, it would mean that each operation must be scheduled into exactly one control step, data dependencies between operations must be enforced, and operation utilization in any given control step must not exceed the resources allocated to the problem. ILP solvers determine the validity of a particular vector through a sequence of (possibly sparse) matrix multiplications, a process which we will show to be unnecessary in this specific problem. Furthermore, ILP solvers attempt to find a valid vector (i.e. a valid schedule in this problem) by enumerating all possible vectors via a “branch and bound” methodology in which all possible enumerations of the vector are generated until an acceptable vector is found. This leads to an exponential run-time (which is not surprising considering that the problem is NP-complete), due to the fact that the problem size is dictated by the number of elements in the vector and each element in the vector has exactly two possibilities. For example, [5] presents such an ILP formulation; it can be shown that the number of enumerations made by this ILP formulation is  $O(2^{df \cdot n})$  in the worst case, where  $n$  is the number of nodes in the data flow graph and  $df$  is the average number of possible control steps that an operation could possibly be scheduled into (i.e. a node’s *degrees of freedom*). However, this work goes on to demonstrate that subsets of their formulation hold certain mathematical properties that should result in improved run times over other existing methodologies. The work found therein also goes on to demonstrate the effectiveness of their formulation, and thus we build our methodology with respect to their ILP formulation.

Our customized methodology attempts to improve upon the ILP solution methodology by folding the three basic constraints mentioned above directly into the enumeration process. In particular, ILP solvers will (uselessly) enumerate possibilities such as scheduling the same operation more than once; obviously this results in an invalid schedule.

Our methodology instead avoids both the “branch and bound” problem and avoids considering such vectors through a more careful enumeration of design points (i.e. vector values). Enumerations are constructed by recursively assigning each operation to a single control step within its schedule interval (that is, between the first and last possible control steps it could be placed into due to data dependencies and number of allocated control steps) and then recursively doing the same for each of its successors. Should this result in an invalid vector at any point in time, the operation is simply re-assigned to the next possible control step as the recursion backtracks through the enumeration.

Furthermore, one can enforce data dependency constraints by simply modifying the schedule interval of operations that are data dependent on the current one. Such modification is possible due to the fact that placement of the current operation means that its data dependent successors may no longer be able to be scheduled in the full range of their schedule intervals. As a result, their schedule interval can temporarily be pruned to contain only those control steps that are not invalidated by the current operation’s placement. Upon backtracking through the recursion, these modified schedule intervals need to be restored as operation placements are restored. This is a simple matter of keeping a list of modifications at each operation’s point in the recursion to allow for such backtracking restoration.

The final constraint that the problem requires represents resource limits. Interestingly, this can be quite easily checked as we place an operation into the binary vector. All that is needed is to keep a count of the number of resources (of each type) utilized in each control step to this point in the recursion. Upon implicitly matching the two constraints as described in the prior paragraph, we only need to attempt to add 1 to the appropriate resource utilization count. If this value stays within the corresponding allocated resource constraint, the operation can be successfully scheduled in the current control step; otherwise, the assignment fails and would then either try the next control step in its schedule interval or, if the schedule interval has been exhausted, cause the recursion to backtrack to the prior operation.

In this manner, our methodology easily folds in the typical constraints found in ILP formulations without a need for the matrix multiplications that the ILP solution methodology requires. Furthermore, the number of enumerations considered is greatly reduced over the ILP methodology, although this number is still exponential in nature.

As a motivational example, some results when

Name	Operations	Delay (cs)
add	+	1
sub	-	2
alu	* /	3

Table 1: Module Library Utilized

Allocation	ILP	Solver	Dist.
1add,1alu,30cs	0.104	0.036	0.031
2add,1alu,29cs	0.050	0.014	0.009
2add,2alu,22cs	0.010	0.013	0.012
3add,4alu,20cs	0.009	0.004	0.003
1add,1alu,300cs	1366.314	0.008	0.007

Table 2: Run Times for EWF Benchmark (sec.)

utilizing the EWF benchmark and the module library found in Table 1 are presented in Table 2. Although a relatively small behavioral description (34 arithmetic operations), the EWF does indicate the success that our methodology has relative to commercial ILP solvers. The first four rows of Table 2 apply the tightest possible feasible time constraints for the given resource allocations, thus representing optimal results. In these cases our methodology performs comparably to the ILP solver<sup>3</sup>- noting that the problem is small enough that the distributed versions of the solver do not significantly aid in reducing the problem’s run time. The last (fifth) row gives a clear indication of a situation where an ILP solver can behave quite poorly. In this case, a very loose time constraint is given, resulting in many control steps and thus a very large binary vector to enumerate. The ILP solver simply builds this enumeration, while our solution quickly builds toward a solution instead of performing a blind “branch and bound” search. In fact, the solution is found with very little recursive backtracking and thus very few enumeration attempts.

While our methodology still exhibits exponential run-times in its worst case, the behavior is significantly better than that found in typical ILP schedulers, despite the fact that our methodology still *guarantees* to find solutions. Further details on the methodology and its analysis show that the number of enumerations explored by our proposed methodology is  $O(df^n)$ . Thus, our methodology’s number of enumerations explored will be significantly less than the number of enumerations found while utilizing an ILP solution to the same scheduling problem. This is illustrated in Figure 4, where the number of enumerations explored by both our methodology and the ILP methodology are plotted for varying sizes of behavioral descriptions (not-

<sup>3</sup>We utilized the AMPL/CPLEX solver[7].

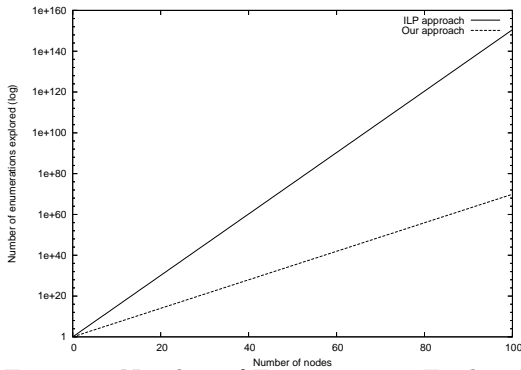


Figure 4: Number of Enumerations Explored

ing that the number of enumerations is logarithmically scaled!). It is also worth noting that the time spent at each enumeration is  $O(n)$  in the worst case for our methodology (due to enforcing data dependence with successor compute nodes), while the ILP formulation requires multiple matrix multiplications at each enumeration resulting in a worst case run time of  $O(df^2 \cdot n^3)$ <sup>4</sup> at each enumeration. Even though our methodology still can exhibit exponential behavior in its worst case (usually when there is no feasible scheduling solution to the specified constraints), the methodology does lend itself very well to distributed versions of the scheduler, as outlined in the following Section.

### 3. The Distributed Scheduler

The distributed version of our customized scheduler is outlined in Figure 3. The basic concept utilized in this system is to simply have one server node partially enumerate a vector (i.e. a schedule). When this is done, the remainder of the enumeration can be passed off to another (client) node to continue enumerating. The server can repeat this process and send another partial enumeration to another client, and so on. We denote each such partial enumeration as a *schedule prefix*. This process, although derived independently, represents an instance of the work outlined in [13] where a general methodology for distributing a branch and bound search for the purposes of solving NP-complete problems is given. However, a clear method of distributing the work is key to such distributed applications and, as is usually the case with such applications, depends upon the nature of the problem being solved.

Obviously, it is undesirable to have the server blindly send prefixes to nodes, as this could easily lead to an uneven distribution of the workload. In-

<sup>4</sup>note that sparse matrix multiplications *may* reduce this somewhat, although certainly not to the linear time of our methodology.

stead, prefixes are generated and kept at the server and distributed to clients as clients join the distributed system or complete a current enumeration based on a previous prefix. Should a client find a solution utilizing the new prefix, it simply notifies the server of the solution and the server subsequently communicates to all remaining clients that no further work should be performed. If the server exhausts all possible prefixes, then there is no more work to be distributed and each client that subsequently requests another prefix is told that the enumeration process has run to completion. Note that in this latter case, there was no solution found, meaning the posed problem is infeasible.

#### 3.1. Schedule Prefixes

Generating and representing the schedule prefixes are obviously of the utmost importance, as they represent the core unit of work that the distributed system is utilizing. Representing a prefix is somewhat simple and straightforward; all we need do is store the currently mapped control step for every operation found in the prefix. In other words, a simple list of (operation, control step) value pairs. The list is then the exact length of the prefix in size, namely the number of operations contained within the prefix, and is thus finitely bounded in size.

#### 3.2. Prefix Length

Determining the appropriate length of the scheduling prefix to utilize is also an important matter. A prefix that is too long will result in minimal amounts of computation at each compute node (client workstation) in the distributed system. This in turn would mean that compute nodes would have to request new prefixes more frequently, resulting in more network traffic. Unfortunately, such can result in network bottlenecks and thus be detrimental to the scalability of the distributed system to the extent that adding more nodes might actual *reduce* the overall performance of the distributed system.

At the other side of the spectrum, making the prefix size too small can also be a disadvantage. While certainly reducing the network communication overhead as a result of requiring more compute time between client prefix requests, the computations themselves may get bogged down in “subproblems” encountered along the way which would be better off being distributed appropriately. This is often a particular problem as the system starts up (but can also be encountered later as well), when solutions are more likely to be found. Solutions are more likely to be found early on during the enumeration process as the design of the scheduler is to first aggressively try to place operations as early as possible within their schedule interval. Should

all of this exploration be dedicated to one compute node, other compute nodes cannot be utilized to help find any potential solution(s). This effectively delays finding solutions due to an inappropriate distribution of the search space. It also limits the number of compute nodes that can be effectively utilized at any point in time; that is, if the number of prefixes to be generated is too low (because the prefix length is too small), that low number puts an upper bound on the number of nodes that could connect. Furthermore, a small number of prefixes to explore may result in one node getting “stuck” on a single long subproblem long after all other nodes have completed all remaining prefixes. An example of this is discussed in Section 4.

Several basic ideas for determining an appropriate prefix length are given below, along with a discussion of their strengths and weaknesses:

- **fixed prefix size** - the prefix length is simply a fixed number of operations, without concern to the nature of the problem being solved. That is, one might always choose to make the prefix contain exactly 10 operations from the input behavioral description. While simple in nature (and implementation), such a scheme could result in the exact problems described above. In essence, a fixed prefix size tends not to work well in practice due to variances in input behavioral descriptions and scheduling problem constraints. That is, the fixed size may perform well in one case and poorly in all others.
- **inverted fixed prefix size** - the prefix length is calculated in a reverse manner. Instead of specifying the number of operations found in the prefix, the number of operations left out of the prefix is specified. The goal here is to fix the size of the problem to be fed into the compute nodes in the distributed system, thus avoiding some of the problems discussed above. Unfortunately, the number of nodes itself does not directly dictate the size of the subproblem being solved; the number of enumerations to be explored is what will dictate such. Factors other than the number of operations, such as the size of corresponding schedule intervals, have a much greater impact on the number of enumerations to explore. As a positive, this method of computing prefix size is also somewhat trivial to implement.
- **scaled prefix size** - the prefix length is simply calculated as a (fixed) percentage of the number of operations in the input behavioral

description. While also a temptingly simple measure of possible compute work, the number of nodes alone still does not dictate the number of enumerations that need to be performed (as discussed with the inverted fixed prefix size method). Thus, this measure also performs poorly in many cases.

- **scaled inverted prefix size** - the prefix length is calculated from an approximation of corresponding subproblem sizes, measured in terms of total enumerations possible. In this case, one must first decide upon the number of enumerations one wishes to have each compute node explore and then determine the prefix length that would result in subproblems of (approximately) this size. For any given prefix size, the number of enumerations in its corresponding subproblem can be approximated by taking the size of the schedule interval for each node that is not in the prefix (and is therefore part of the subproblem to be solved) and multiplying these values together. It is important to note that doing such really just gives an upper bound on the number of enumerations possible, as the scheduling methodology may prune many of these enumerations before they are explored. One can iteratively search through the various prefix sizes until the desired subproblem size has been found. While slightly more complex than other methods of determining prefix size, this method gives the most measurable (and thus useful) means of finding an appropriate prefix size. The method also results in different prefix sizes for different scheduling problem formulations over the same behavioral description when the same upper bound on number of enumerations is used, unlike the previous relatively static methods.

Each of the above methods can be used to successfully generate prefixes, but only the scaled inverted prefix method is truly applicable in general practice, so that is the method we will choose to employ for the remainder of this paper. In any case, each method still has a parameter that would guide (if not outright choose) the size of our prefix, and it is necessary to choose an appropriate parameter value therein. Interestingly, the concept of *scalability* in distributed systems comes into play in helping to determine this value, as discussed below.

### 3.3. Scalability

Scalability refers to the idea that adding compute nodes to a distributed system should always improve its run time performance, preferably linearly

Tolerance Parameter	Prefix Length	Number of Nodes		
		2	3	4
$10^{30}$	6	11.1	7.5	5.5
$10^{29}$	7	8.5	5.8	4.3
$10^{28}$	8	12.2	8.1	6.2

Table 3: Exhaustive EWF Search Times (mins.)

in the number of compute nodes. As discussed previously, one of the primary reasons that distributed systems fail to deliver full scalability is communication overhead [6]; prefix length certainly has the capability of impacting this in our scheduler, as described previously. In our context, we want to reduce communications when the number of nodes increases, meaning that as the number of compute nodes in the distributed system increases significantly, prefix length should decrease to cut down on communications overhead.

Even these guidelines do not determine an appropriate “enumeration upper bound” when using the scaled inverted prefix method. This value should be roughly the same for a distributed system of a fixed number of compute nodes of equal processing power (noting that our methodology will still work when compute nodes do not have the same processing power) no matter what the behavioral description or scheduling constraints are. That is, once an appropriate value has been determined for a given distributed system, it need not be changed again. Unfortunately, the best method for finding such a value is trial and error. In our case, each compute node is a 3.2GhZ Pentium 4 with Hyper-threading technology running the Linux operating system (Fedora Core 4). Due to limited resource availability <sup>5</sup>, we have only been able to dedicate 4 machines to our distributed system, and they are connected by a 100Mb hub. Experiments showed that a subproblem size of roughly  $10^{29}$  possible enumerations worked well, keeping the network communications hub at about 10% capacity without having one compute node dominate the system. Table 3 shows our distributed system run times for various different enumeration size parameters for the EWF benchmark when run with an infeasible scheduling constraint, namely a minimal resource allocation (1 unit of each type) and 28 control steps.

One should also note the linear improvement in run time as the number of nodes increases. This demonstrates that our distributed approach is truly scalable, a property highly desired in distributed systems. It is worth noting that as the number of

<sup>5</sup>the author’s host institution did not have the resources to dedicate more than 4 compute nodes.

Nodes	Allocation	ILP	Solver	Dist.
32	66cs	8.817	0.005	0.005
64	87cs	–	0.013	0.009
128	202cs	–	0.025	0.019
256	417cs	–	0.058	0.051
512	792cs	–	0.152	0.117

Table 4: Run Times for Pseudo-Random Behavioral Descriptions (secs.)

Tolerance Parameter	Number of Nodes		
	2	3	4
$10^{30}$	54.5	54.5	54.5
$10^{24}$	50.9	33.8	25.2

Table 5: Exhaustive Enumeration Times (mins.)

compute nodes increases, one may wish to increase the enumeration size parameter being utilized. As discussed previously, failure to do so may result in increased communication overhead and thus a reduction in the efficiency of the distributed system.

## 4. Further Results

So far, we have only presented results for the EWF example. It is important to show that our methodology scales well as the size of the behavioral description increases. In Table 4, we present results for our methodology on pseudo-random graphs of exponentially increasing size containing +, -, /, and \* operations. In each case, the resource allocation is minimal (i.e. 1 resource of each type), as is the latency constraint for this resource allocation. Note that for graphs with 64 or more nodes, comparison with ILP run times is exceedingly difficult, as the ILP ran for two hours and was manually halted in each case (before finding a solution). Again, this is an indicator of the effectiveness of our methodology, which found solutions in under 1 second in each case, even with larger behavioral descriptions.

Also of interest is further proof that our method scales well when full enumerations must occur. To demonstrate this, we perform an experiment similar to the “full enumeration” experiment summarized previously in Table 3, but using the randomly generated behavioral description with 32 nodes. Results from this experiment can be found in Table 5. In this case, a time constraint that cannot be satisfied (27 control steps) is specified, meaning that all possibilities must be enumerated en route to deciding that the requested design is infeasible. Interestingly, one can see that the methodology scales well in one case and not in another. When the tolerance parameter is  $10^{24}$ , the resulting work scales quite well. However, the poor choice of  $10^{30}$  in this case shows the danger of a bad choice. In this particu-

lar instance, what has happened is that one of the prefixes generated early on results in a very time consuming problem that should have been broken down into several subproblems. Instead, one node spends all of its time computing the result corresponding to that prefix, and the other client nodes complete all of the remaining prefixes long before this single prefix has its result computed. Thus, no matter how many nodes we add to our system, the run time will never be improved in this case! Nonetheless, with a proper choice of prefix length, it is not unreasonable to expect further run-time improvements as the number of nodes in the distributed system increases.

## 5. Conclusions & Future Work

We have presented a new scheduling algorithm based on the components found in ILP models of the scheduling problem that avoids the overhead found in ILP solvers, while still *guaranteeing* that solutions will be found. We have also shown that the scheduler lends itself quite well to building a distributed scheduling algorithm. Lastly, we have discussed the basic structure and customizable parameters involved in our distributed system and how they can be utilized to ensure our methodology can scale well to a much larger distributed system. Along these lines, one future avenue of research would be to utilize users on the Internet at large to help solve specific scheduling problems, as is done with NASA's SETI program [1] or the search for Mersenne prime numbers [10].

While the scheduler itself currently only considers time and resource constraints, other subproblems or constraints such as power consumption [4] or temperature [11] can easily be folded into the methodology. This can be accomplished during our enumeration by testing these constraints in addition to the others constraints discussed in this paper. Depending on the nature of the constraint, it may only be possible to test it at the leaves of the recursion, when a possible design has already been fully enumerated, in which case the new constraint will have little impact on the run time of the methodology. In other cases, it may be possible to apply the new constraint while an enumeration is being recursively built; in this case, the new constraint may well aid in pruning the search space, thus effectively improving the performance of our methodology.

Another possible improvement would be to modify the nature of the problem our methodology solves itself. Currently, only a decision problem version of the scheduling problem is solved, wherein a potential solution space is explored to determine

whether or not it contains a solution. Instead, research into extending our solution methodology so that it finds the best solution (as measured by a user specified metric) in a solution space may be possible. The scalable nature of our methodology is likely to aid in this pursuit.

## References

- [1] D. W. Aaron Parsons. SETI Instrumentation. [http://setiathome.ssl.berkeley.edu/~aparsons/papers/2003-10-29\\_Arecibo\\_Presentation.pdf](http://setiathome.ssl.berkeley.edu/~aparsons/papers/2003-10-29_Arecibo_Presentation.pdf).
- [2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ILP Versus Specialized 0-1 ILP: an Update. In *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, pages 450–457, San Jose, California, Nov. 10-14 2002. IEEE Computer Society Press.
- [3] A. B. Barskiy. Minimizing the Number of Computing Devices Needed to Realize a Computational Process within a Specified Time. *Engineering Cybernetics (USSR)*, 17:59–63, July 1968.
- [4] A. Chang and W. J. Dally. Explaining The Gap Between ASIC And Custom Power: A Custom Perspective. In *Proc. of the 42nd ACM/IEEE Design Automation Conf.*, pages 281–284, Anaheim, California, June 13-17 2005. IEEE Computer Society Press.
- [5] S. Chaudhuri, R. A. Walker, and J. E. Mitchell. Analyzing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem. *IEEE Transactions on VLSI Systems*, 2(4):456–471, Dec. 1994.
- [6] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Concepts and Design*. Pearson Education / Addison-Wesley, Essex, England, 2001.
- [7] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Wadsworth, Pacific Grove, CA, USA, 1993.
- [8] D. D. Gajski, editor. *Silicon Compilation*. Addison-Wesley, Reading, MA, USA, 1988.
- [9] C. H. Gebotys and M. I. Elmasry. Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis. In *Proc. of the 28th ACM/IEEE Design Automation Conf.*, pages 2–7, San Francisco, California, June 17-21 1991. IEEE Computer Society Press.
- [10] GIMP. Great Internet Mersenne Prime Search Home Page. [www.mersenne.org](http://www.mersenne.org).
- [11] R. Mukherjee, S. O. Memik, and G. Memik. Temperature-Aware Resource Allocation and Binding in High-Level Synthesis. In *Proc. of the 42nd ACM/IEEE Design Automation Conf.*, pages 196–201, Anaheim, California, June 13-17 2005. IEEE Computer Society Press.
- [12] P. G. Paulin and J. P. Knight. Force Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Transactions on Computer-Aided Design*, 8(6):661–679, June 1989.
- [13] P. Sanders. Parallelizing NP-Complete Problems Using Tree Shaped Computations. [cite-seer.ist.psu.edu/sanders99parallelizing.html](http://ciseer.ist.psu.edu/sanders99parallelizing.html).