Improvement of Memory Management for the Optimization of Integrated Circuit Simulators

Timothée Bossart, Alix Munier, Anna Robert, Francis Sourd
Laboratoire d'informatique de Paris VI, 4 place Jussieu, 75005 Paris
{timothee.bossart,alix.munier,francis.sourd}@lip6.fr

Abstract

This paper describes two simple mathematical models for the minimization of the memory access of a cycle-based simulator. An integrated circuit can be viewed as a directed acyclic graph G. The problem consists then to build a graph order on the vertices, compatible with the relation order induced by G, and minimizing the cost function.

We first noticed that, experimentally, the two cost functions we propose are correlated. Since the number of gates of a circuits may be important, we then focus our study on simple greedy algorithms. We firstly test two very simple intuitive strategies and show experimentally that they do not behave better than a random numbering of the vertices. Then, we tried a more sophisticated heuristic and we show that a speed up of near 20% *may be achieved in real conditions.*

1 Introduction

Simulation is a crucial challenge for the design of integrated circuits [4]. In very few words, a cycled-based simulator can be viewed as a computer program that reads a file that contains the physical description of an integrated circuit — a VHDL file for example — and produces, in a so-called *compilation phase*, an *executable simulation code* that simulates the behavior of the circuit. Then, the *test phase* consists in running the executable code on a large number of benchmarks.

As the executable code that simulates the circuit is run a very large number of times — some test phases may last several days — producing an optimized code is of practical interest to significantly reduce the length of the test phase. The generated simulation code has a very special structure. On the one hand, there is no loop nor branching instructions. On the other hand, there are a great number of instructions and of variables to deal with in each block of code. Conventional compilers, such as `gcc`, are not devised to optimize such a code.

In this paper, an integrated circuit will be seen as a set of *logical gates* (such as AND, OR, NOT…) interconnected through wires represented by a directed acyclic graph (see figure 1(a) and 1(b)). The value of the output of a gate is directly derived from its inputs so that the role of the code simulating the circuit is to sequentially compute the values of all the wires in order to compute the output. Clearly, since all the inputs must be calculated in order to compute the output, the values of the wires must be computed in a topological order (*e.g.* [2]) induced by the digraph (see figure 1(c)). Conversely, any topological order of the digraph yields a different code so that our problem is to find a topological order that produces the fastest code.

The main difficulty in building the model is to find an estimate for the speed of the code. In fact, given the great number of variables induced by a large integrated circuit, memory management is of key importance in order to use different cache levels at best. The two models proposed here aim at minimizing total memory access time.

In Section 2, we present the model and the two score functions. We also briefly discuss on their correlation. Section 3 is devoted to the presentation of a new heuristic to solve the practical problem. The validation process is detailed, and we present some experimental results. In conclusion, some insights about further validations are presented, and the relevance of our work is discussed.

2 Models

Let $G(V,A)$ be a directed acyclic graph representing the dependence between the variables of the simulation code. Each vertex of V is associated with a logical gate or an input of the logical circuit. An arc $(u,v) \in A$ corresponds to a wire. We set $n = |V|$ and $m = |A|$. $\forall v \in V$, we denote by $\Gamma^+(v)$ (*resp.* $\Gamma^-(v)$) the set of successors (*resp.* predecessors) of v.
Since any possible code is represented by a numbering of the nodes of G, a *feasible solution* of the problem is formally described by a bijection φ that maps V to $\{1,\cdots,n\}$

and satisfies the constraint $\forall a = (u,v) \in A, \varphi(u) < \varphi(v)$. φ is called a *graph ordering function* or a *graph order*.

For example, let us consider the logical circuit pictured by figure 1(a). The associated graph is pictured by figure 1(b). The figure 1(c) corresponds to a feasible code. The corresponding graph order is $\varphi(v_1) = 1$, $\varphi(v_2) = 2$, $\varphi(v_3) = 3$, $\varphi(v_4) = 5$, $\varphi(v_5) = 6$, $\varphi(v_6) = 4$, $\varphi(v_7) = 7$ and $\varphi(v_8) = 8$.

Given a graph ordering function φ, let $C(\varphi, u)$ denote the *total cache access cost* for the variable u. This cost represents the sum of the access costs to search in memory the variable u. The total cost of the order φ on G will then be defined in the following way:

$$C(\varphi, G) = \sum_{u \in V} C(\varphi, u)$$

The problem is to find an order φ that minimizes this objective function. In the rest of this section, two models are proposed to evaluate by two different ways the cache access costs $C(\varphi, u)$. In both these models, $C(\varphi, u)$ is a deterministic function that only depends on φ and u. This hypothesis can be criticized because cache policies may be randomized and the real access times depends of numerous other parameters such as the cache size, the operating system (and its settings), the memory state when the simulation code is run, the programs that are concurrently run and many others. However, as the simulation code is run many times — eventually on different machines — we are interested in minimizing its average running time. So, we are going to assume that $C(\varphi, u)$ represents something like a "mean" cache access cost.

We observe that the expression of the total cost $C(\varphi, G)$ deliberately ignores the time spent at computing the value of the output of the logical gate once the input are read. In fact, this time is assumed to be constant. So, the total time for running the simulation code is the sum of a constant computation time and a cache access time depending on φ. Only this second value is minimized.

In the first model, the estimation of $C(\varphi, u)$ is based on the number of instructions executed between to successive usage of the variable u. The second model is more complex since it keeps track of all the memory moves.

2.1 Directed Sum Cut

This model is based is based on the observation that φ induces a numbering of the lines of the simulation code (see Figure 1(c)): $\varphi(v_i)$ is the number of the line at which v_i is created. Let us consider $u \in V$ and the elements from $\Gamma^+(u) \cup \{u\}$ ordered following the sequences $v_0, v_1, \ldots, v_{|\Gamma^+(v)|}$ with $v_0 = u$ and $\varphi(v_0) < \varphi(v_1) < \ldots < \varphi(v_{|\Gamma^+(v)|})$. The first access to u is made when v_1 is computed. The cost for reading u is then $f(\varphi(v_1) - \varphi(u))$. After this computation, both the variables u and v_1 are at the top of the memory stack so that when u is accessed by the second successor v_2 of u, the access cost is $f(\varphi(v_2) - \varphi(v_1))$. As a consequence, the total cost related to the access to variable u is

$$C(\varphi, u) = \sum_{i=1}^{|\Gamma^+(v)|} f(\varphi(v_i) - \varphi(v_{i-1}))$$

In order to simplify the model, we assume that the cache access cost function is simply the identity function. Under this assumption, we get:

$$C(\varphi, G) = \sum_{u \in V} \left(\left(\max_{(u,v) \in A} \varphi(v) \right) - \varphi(u) \right)$$

For the example of figure 2, the total DSC cost is equal to 9. This model is proved in [1] to have the same score function as the directed version of SUMCUT [3]. For the following, we will therefore write:

$$C(\varphi, G) = \mathrm{DSC}(\varphi, G)$$

2.2 Uniform Cost Stack

The UCS model (for *Uniform Cost Stack*) is intended to represent the load costs of variables which are stored during the execution of a program. This model is clearly an extension of the well-known model of Sethi presented in [8] for the register allocation problems.

The memory is seen as a stack, on which two operators are available:

- *LD (α)* moves the value α stored in the stack to the top (of the stack). The cost of this operation is proportional to the number of variables stored between α and the top before the move.
- *OP ($\alpha_1, \cdots, \alpha_k$)* applies a commutative operator to $\alpha_1 \cdots \alpha_k$. It is supposed that $\alpha_1 \cdots \alpha_k$ have been previously moved to the first k levels of the stack. The result is then moved to the top of the stack. Since the cost of an operation is supposed to be constant, we set it equal to zero.

For example, let us consider a graph $G(V,A)$ pictured by figure 2. The ordering function corresponds to the numbering of the nodes. Then, we can deduce a list of operations on the stack to evaluate the vertices of the graph following the numbering function (see figure 3). The cost of an execution is then the summation of the costs of the moves: each of them is associated with an arc $(x,y) \in A$.

For our example of figure 3, the arcs are valued with the corresponding cost. We finally get a total UCS cost of 9.

For general graphs, the code generation associated with graph order φ is more complicated: indeed, if a vertex $u \in V$

has several predecessors, we have to decide in which order they will be loaded in the stack before u to minimize the cost. It is proved in [1] that a simple (polynomial) strategy can be developed to get the best code for a given order φ.

2.3 Correlation between DSC and UCS

It is proved in [1] that UCS and DSC are polynomial for out-trees and coincide in this case. These criteria are not obviously correlated theoretically for general graph structures.

Experimentally, we observed a linear correlation of these two criteria for a great number of graphs generated randomly [7]. For example, figure 4 shows the values of both scores upon several random orderings of the precedence graph of an IEEE 64 bit multiplier designed with GenOptim [6].

This point allowed us to develop common heuristics for the two models, and to test them simultaneously.

3 Practical resolution

The aim of this section is to present a heuristic to build an efficient graph order and to present some experimental results: it is proved in [1] that DSC and UCS are both NP-hard for some particular graph structures. Moreover, the number of vertices n may be important for the practical applications. So, in order to obtain efficient solutions, we must develop heuristics with a very low complexity (*i.e.* a complexity of $O(n)$). Firstly, we present the heuristic that we developed for DSC and UCS. Then, we introduce our validation process. In the third part, we show experimentally that this heuristic improves the solution usually implemented.

3.1 Ordering heuristics

Heuristics currently in use in simulators such as CASS [5] only use a random topological order. At each step i, this algorithm numbers a vertex chosen randomly among a set $L(i)$ composed by the vertices whose predecessors are all numbered.

The simplest strategies derived from this algorithm are to bias the choice of the current vertex: let S be the set of the *source vertices* of G, $S = \{u \in V, \Gamma^-(u) = \emptyset\}$. $\forall u \in V$, the level of a vertex u will be defined as the number of arcs of the longest path from S to u. Then, a depth-first (*resp.* width-first) strategy consists in choosing a vertex of maximum (*resp.* minimum) level.

We tested these two strategies against random, with poor results. This lead us to develop a new heuristic intended to take more into account the inner structure of our models, their main common point being the fact that in both cases vertices must not be computed too long after the computation of their predecessor.

For our new heuristic, the candidate set C is composed of vertices whose numbering is urgent in terms of memory management. At each step $i \in \{1, \cdots, n\}$ of the algorithm, vertex u of minimum level is chosen out of the set of candidate nodes $C \subset V$. The set of the predecessors of u not yet numbered is denoted P, and thus there are 2 cases:

if $P = \emptyset$
  $\varphi(u) \leftarrow i$
  $i \leftarrow i + 1$
  $C \leftarrow C - \{u\} \cup \Gamma^+(u)$
else
  $C \leftarrow C \cup P$
endif

The candidate set C is initialized with a vertex $u \in S$. The algorithm orders every vertex because G is connected. The second case is treated at most $(n-1)$ times because it adds at least a vertex to the candidate set. Furthermore, at each iteration the predecessors of the chosen vertex are scanned, so the number of operations is at most $O(2nm)$. The complexity is $O(n)$ for practical applications since the mean degree of the vertices is bounded. An example is shown in figure 6.

3.2 Validation process

Methods which are usually applied in order to evaluate the quality of a heuristic are often hard to use on graph ordering problems, mainly because it is difficult to find good lower bounds for the criterion. For this reason, we decided to compare our results to a random topological order. This comparison was made possible by the implementation of a simulation environment described in figure 7.

First a precedence graph is generated from a netlist. For statistical measure purposes, a pseudo-random graph generator was also designed. The method is very simple. The vertices are numbered from 1 to n, and a bounded number of arcs (u, v) are pseudo-randomly generated. In order to forbid cycles, the condition $u < v$ is added for every arc.

Then, a simple program is generated in order to simulate the behavior of the circuit whose dependence graph is the generated graph. For this, we suppose that all the operators are identical, and the input data is generated randomly. In other words, our aim is not to simulate the semantic of the integrated component, but only the memory movements which occur during its simulation. The simulator is then compiled with a classical compiler such as `gcc`, and the

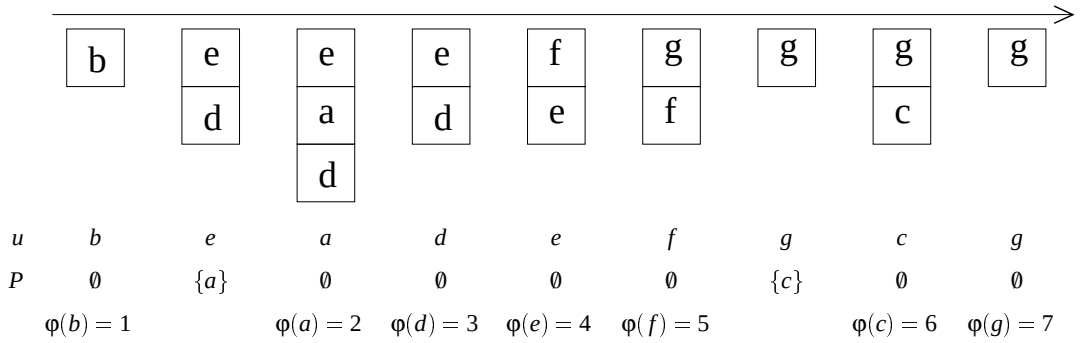


Figure 6. Evolution of the candidate set during the computation of the graph shown in figure 5

execution duration is measured. The UCS and DSC scores are also computed from the graph.

Up to a sufficient graph size, two factors prevent to make a good analysis of the results. On the one hand, the execution time is too small to be correctly measured by a computer. On the other hand, the low number of variables guarantees that all the execution can be made within low levels of the memory, which are very fast. For these reasons, our measures were made with graphs containing at least 10^3 vertices.
We also used the following method in order to magnify the memory defaults: the simulator is designed to handle memory blocks of parameterizable size. During the experiments which are presented here, this size was within `2000*sizeof(int)` and `20000*sizeof(int)`. Moreover, this also reflects the fact that high level simulators often manage heavy data structures.

3.3 Experimental results

Table 1 shows the execution time speed-ups obtained with our heuristic against a random order for two precedence graphs of operators computed by GenOptim [6]: an IEEE 64 bit divider and an IEEE 64 bit multiplier. Although the scoring method is deterministic, the execution time is actually the mean duration of several executions upon convergence, in order to reduce the side effects of the system under which the simulator is running. In fact, the variance can sometimes be more than twice the mean value. The convergence is empirically reached after 100 executions.
It was empirically observed for a great number of pseudo random graphs and netlist generated graphs that the new heuristic improves both UCS and DSC scores with regards to a random ordering. The results of table 1 are coherent with these observations. This improvement is also true for mean execution times.

		Divider	Multiplier
Vertices		35078	3109
Arcs		76023	7923
UCS score	random	82252692	1515454
	heuristic	80510609	1329766
Speed up		2%	12%
DSC score	random	68394731	883191
	heuristic	66682900	751626
Speed up		3%	15%
Execution time (ms)	random	59461	7972
	heuristic	50037	6431
Speed up		16%	19%

Table 1. Execution time speed up

4 Conclusions

This work has proved that the study of a theoretical model of memory management can result in a practical improvement of the processing time required to simulate and test integrated circuits. Indeed, we have formulated two combinatorial optimization problems in graphs and we have proposed a heuristic algorithm to solve them. This algorithm can be used to generate a simulation code that efficiently manages the memory. This heuristic algorithm has been experimentally proved to significantly decrease the average execution times of the simulation code.

As this study settles the usefulness of a specific algorithm to generate the simulation code, the next step of our work is to completely integrate our method into a real simulation environment. Such an environment is clearly more complex than the simplified experimental platform we have built for our tests, so we expect we will have to refine our theoretical model in order to optimize this integration. Maybe, we will have to improve and modify our heuristic

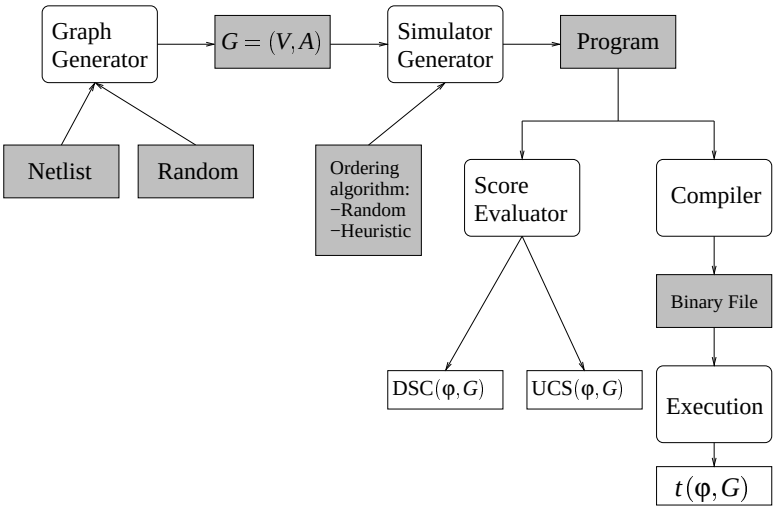


Figure 7. Validation process for the models

so that it fits a compiler or an operating system at best. At the end, the objective of reaching the same speed-up of 15% for a real simulation code is an exciting challenge.

References

[1] T. Bossart, A. Munier, and F. Sourd. Two models for the optimization of integrated circuit simulators. Technical report, Laboratoire d'Informatique de Paris VI, 2003. in submission.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[3] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

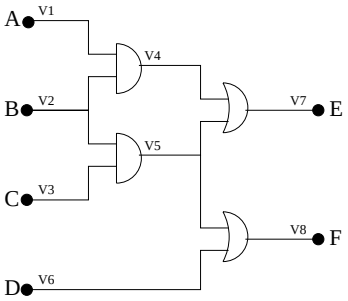
[4] J. B. Gosling. *Simulation in the Design of Digital Electronic Systems*. Cambridge University Press, 1993.

[5] D. Hommais. *Une méthode d'évaluation et de synthèse des communications dans les systèmes intégrés matériel-logiciel*. PhD thesis, University of Paris VI, 2001. In French.

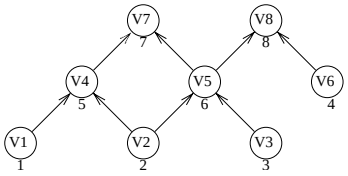
[6] A. Houelle. *GenOptim : un environnement d'aide à la conception de générateurs de circuits portables optimisés en performance et en surface*. PhD thesis, University of Paris VI, 20 June 1997. In French.

[7] A. Robert. Modélisation, analyse et optimisation d'un simulateur de circuits. Master's thesis, University of Paris VI, 2003. in French.

[8] R. Sethi. Complete register allocation problems. *SIAM J. Computing*, 4(3):226–248, September 1975.



(a)



(b)

$V_1 \longleftarrow A$
$V_2 \longleftarrow B$
$V_3 \longleftarrow C$
$V_6 \longleftarrow D$
$V_4 \longleftarrow \mathrm{AND}(V_1, V_2)$
$V_5 \longleftarrow \mathrm{AND}(V_2, V_3)$
$V_7 \longleftarrow \mathrm{OR}(V_4, V_5)$
$V_8 \longleftarrow \mathrm{OR}(V_5, V_6)$
$E \longleftarrow V_7$
$F \longleftarrow V_8$

(c)

Figure 1. Code generation for the simulation of a circuit

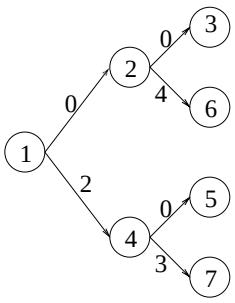


Figure 2. Sample numbered outtree with arc UCS scores

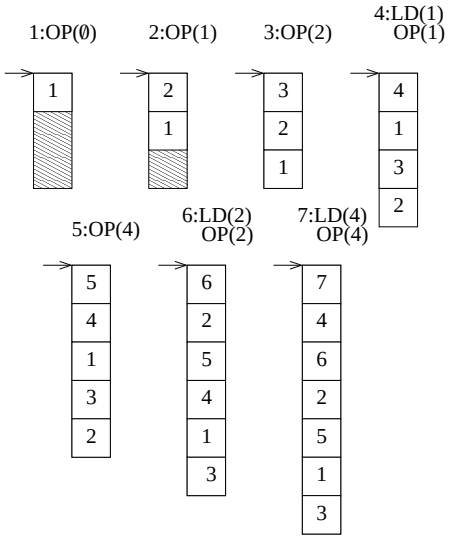


Figure 3. Evolution of the stack during the computation of the tree of figure 2

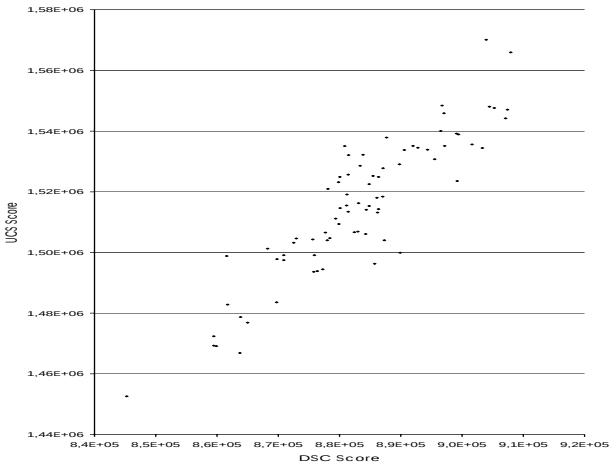


Figure 4. Correlation between DSC and UCS

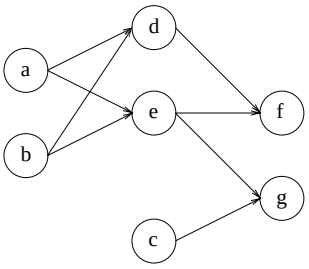


Figure 5. Sample graph ordered with the heuristic described in figure 6