Fast Compiler-Driven Hierarchical Logic Simulation

Björn Schieffer

Fachbereich 14 – Informatik, Universität des Saarlandes
Postfach 15 11 50, 66041 Saarbrücken, Germany
email: schieffer@cs.uni-sb.de

Abstract

Hierarchy can be used to make compiler-driven logic simulation available for very large circuits where the size of a flat simulation routine is too big. But due to many parameter evaluations, hierarchical simulators have an unacceptable runtime. In this paper, it is shown how to reach the performance of a flat simulator using hierarchical code but avoiding the expensive information transport with parameters.

Introduction

Many logic simulators are *event-driven* [AS87, BHKS91]. This means that they only calculate the assignment of a signal when it could have changed. Unfortunately, the management of such an event is more expensive than the calculation of many signal assignments. So, event checks are not always better than *brute force* assignment calculation – depending on the structure of the circuit and the simulated input pattern. If, for each circuit considered, a simulation routine is created and compiled to speed up the expensive simulation, it is called *compiler-driven* simulation. A hierarchical extension for event-driven logic simulation is proposed in [WS92]. This paper examines the hierarchical extensions of compiler-driven simulators.

Hierarchical simulation routines are very short compared to flat ones, i.e. the flat routine of a 256 bit carry look ahead adder takes 15378 lines of code in contrast to 396 lines for the hierarchical one. But due to many parameter evaluations the hierarchical adder routine is 30 times slower than the flat one. In this paper, a simulator with hierarchical code and nearly the speed of the flat simulator (only 1.3 times slower in the case of the 256 bit adder) is presented.

1 Flat Or Hierarchical Solutions

The problem of logic simulation is to determine the assignment of the primary outputs of a given circuit for a given input pattern. A compiler-driven logic simulator goes straight forward through the cells of the circuit in topological order while calculating the signal assignments. The following routine $flat_1$ is the simulation routine for the full adder in figure 1.

$flat_1(i_1, i_2, i_3)$
signals $i_1, i_2, i_3, s_1, s_2, \cdots, s_9, o_1, o_2$;
begin
$s_1 = i_1$ **and** i_2;
$s_2 = i_1$ **nand** i_2;
$s_3 = i_1$ **nand** s_2;
$s_4 = s_2$ **nand** i_2;
$s_5 = s_3$ **nand** s_4;
$s_6 = s_5$ **and** i_3;
$s_7 = s_5$ **nand** i_3;
$s_8 = s_5$ **nand** s_7;
$s_9 = s_7$ **nand** i_3;
$o_2 = s_8$ **nand** s_9;
$o_1 = s_1$ **or** s_6;
return(o_1, o_2);
end;

Now, we are looking for a *good* extension of this algorithm which makes it usable for hierarchical circuit descriptions. The cells of a hierarchical circuit are basic cells – as in flat circuits – or box cells. These are instances of subcircuits. The description of such a subcircuit is called module.

A hierarchical simulator [App88, WM89, Mau91] can look like this: For each module, a simulation routine is generated. The assignment of the outputs of each cell is determined in topological order. The calculation of box cells is done by calling the simulation routine of the module describing the box cell.

Therefore, we need a topological ordering for all modules. In contrast to the flat circuit, this is **not** equivalent to the circuit being combinational. There can be *virtual feedbacks* [Bee92] – i.e. cycles in the hierarchical circuit representation – which vanish in a disintegration of the circuit. The dotted line in circuit B of figure 2 connects one output of cell A with one of its inputs, but the output is independent from this input. So, there exists a topological ordering of the basic cells of the flat circuit. (first the right and-gate then the left one.) The feedback of the dotted line is only virtual. The size of the minimal disintegration without virtual feedbacks can be exponentially in the size of the circuit [KS91]. The search for the minimal disintegration is a NP-complete problem [WM89, KS91]. But we can restrict to circuits with no virtual feedbacks, because the feedback deletion is easy in practical designs.

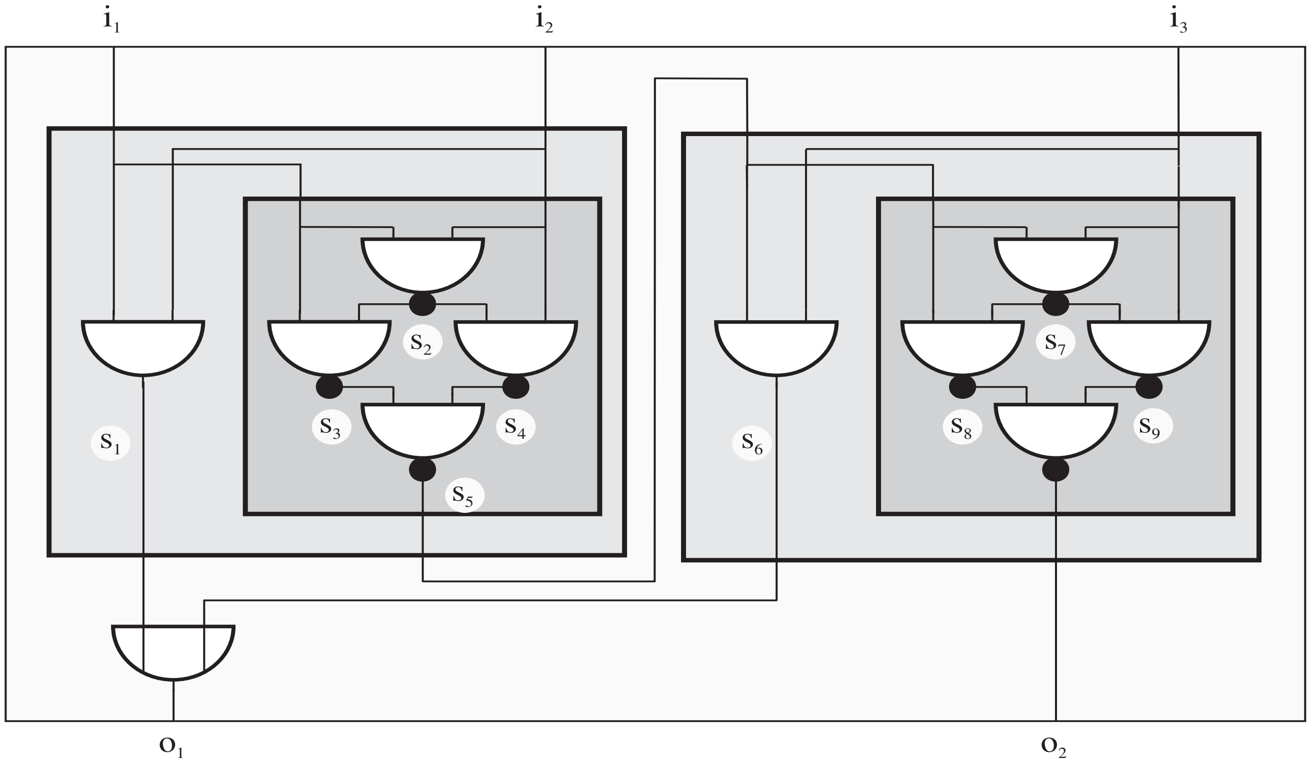

Figure 1: flat description of a full adder

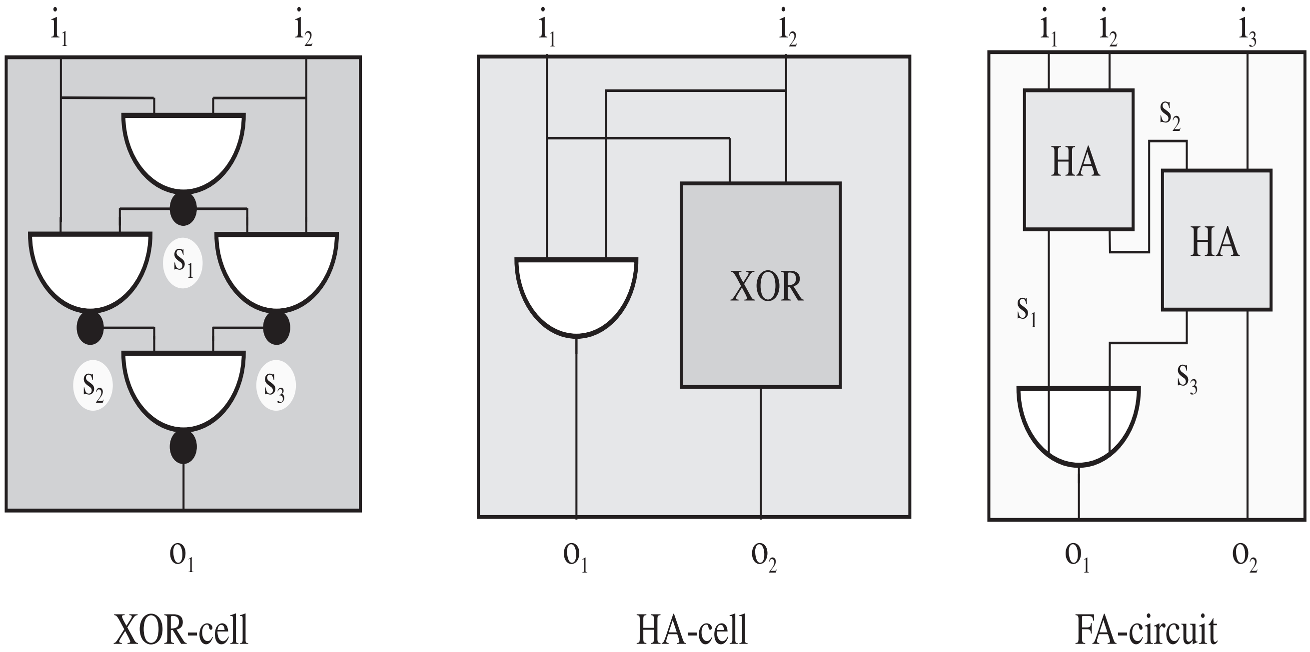

Figure 3: hierarchical description of a full adder

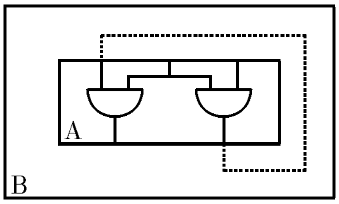

Figure 2: virtual feedback

If we have the hierarchical description of the full adder in figure 3 the simulation routines may look like this:

$xor_1(i_1, i_2)$
signals $i_1, i_2, s_1, s_2, s_3, o_1$;
begin
    $s_1 = i_1$ **nand** i_2;
    $s_2 = i_1$ **nand** s_1;
    $s_3 = s_1$ **nand** i_2;
    $o_1 = s_2$ **nand** s_3;
    return(o_1);
end;

$ha_1(i_1, i_2)$
signals i_1, i_2, o_1, o_2;
begin
    $o_1 = i_1$ **and** i_2;
    $o_2 = xor_1(i_1, i_2)$;
    return(o_1, o_2);
end;

$fa_1(i_2, i_2, i_3)$
signals $i_1, i_2, i_3, s_1, s_2, \cdots, s_3, o_1, o_2$;
begin
    $(s_1, s_2) = ha_1(i_1, i_2)$;
    $(s_3, o_2) = ha_1(s_2, i_3)$;
    $o_1 = s_1$ **or** s_3;
    return(o_1, o_2);
end;

To rate hierarchical simulators we first need a measure for their quality.

S_1: hierarchical signal assignment
$flat_1$ needs the assignment of all the signals $s_1, s_2, \cdots s_9$ from figure 1. In contrast to this, fa_1, ha_1 and xor_1 only use the signal assignments corresponding to figure 3. After the simulation of a box cell, the local signals are dropped. This can be regarded as the implementation of a stack for the signal assignment.

S_2: hierarchical circuit representation
Not only the *dynamic* memory needed for the signal assignment, but also the *static* memory space to store the description of the circuit can be either hierarchical or flat. In our example, the circuit description is given as part of the simulation code. Surely, a good simulator should only use hierarchical circuit representations[1].

S_3: linear runtime
This is very important because of the size of VLSI-circuits.

The conditions S_1 and S_2 are fulfilled by the hierarchical simulator. The following theorem shows, why it violates condition S_3.

Definition 1 *A circuit is said to be* reasonable *iff it contains no* package modules *– that are modules containing only one box cell and no basic cells. Only basic cells may be packed in a module.*

Theorem 1 *Let n be the number of signals in the disintegration of an arbitrary circuit H. Then, the runtime of the hierarchical simulator is not limited by a function of n. But if H is reasonable, it has runtime $\Theta(n^2)$.*

For the sake of shortness we do without the complete proof. But take the circuit family H_k^{buffer} for $k = 1, 2, 3 \cdots$ in figure 4 as an example for quadratical runtime. The disintegration of H_k^{buffer} consists of k parallel buffers and therefore it has $2k$ signals.

$buffer_1(i_1)$
signal i_1;
begin
    return(i_1);
end;

$buffer_2(i_1, i_2)$
signals i_1, i_2, s_1, s_2;
begin
    $s_1 = buffer_1(i_1)$;
    $s_2 = buffer_1(i_2)$;
    return(s_1, s_2);
end;

...

$buffer_k(i_1, i_2, \cdots, i_k)$
signals $i_1, i_2, \cdots, i_k, s_1, s_2, \cdots, s_k$;
begin
    $(s_1, s_2, \cdots s_{k-1})$
        $= buffer_{k-1}(i_1, i_2, \cdots i_{k-1})$;
    $s_k = buffer_1(i_k)$;
    return$(s_1, s_2, \cdots, s_k)$;
end;

[1] In fact, compilers can have severe problems to translate the flat simulation code for the purpose of fault simulation where the code is embedded in a switch statement [DG87].

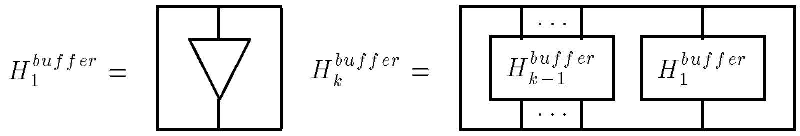

Figure 4: buffer circuit

The runtime t of $buffer_k$ is $\Theta(k)$ for parameter evaluation plus the runtime of $buffer_{k-1}$ plus runtime of $buffer_1$ which is of constant size:

$$\left.\begin{array}{r} t(1) = \Theta(1) \\ t(k) = \Theta(k) + t(k-1) + t(1) \end{array}\right\} \Rightarrow$$

$$t(k) = \Theta(k^2)$$

The path between the leftmost input of the circuit and its corresponding buffer crosses $k-1$ module borders. Each crossing corresponds to a parameter in a procedure call. Since the size of the whole circuit is only $2k$, this is very much for a single transport. To get hierarchical algorithms with linear runtime we demand in the next section that a single transport has only constant costs.

2 Mixed Solutions

Since we have seen the hierarchical simulator not to fulfill condition S_3, we want to design an algorithm doing so. Unfortunately, we have to give up either condition S_1 or condition S_2. At the end of this section we will see why this is the case.

All simulators compute the assignments of all signals. Differences only lie in the storage of the signal assignment and their transport from the point of creation – the source – to the points of usage – the access.

Let us visualize the operations necessary to perform the information flow in the different algorithms by arrows. An arrow in the direction of the information flow illustrates a write operation from the source of the arrow to its destination. An arrow against the flow describes a read operation. Figure 5 illustrates the information flow in the flat simulator. Computing the right gate needs the output value of the left gate given by a single read operation.

The claims for the algorithms can now be formulated in terms of arrows:

P_1: **hierarchical signal assignment:**
the realization of the signal assignment as a stack implies a *visibility* of instances of cells. A cell instance does not *see* all others but only its ancestors, i.e. all instances on the path from the root to itself in the tree description of the circuit. In our example, the two xor-cells do not see each other while they see the and-gates of their corresponding half-adder-cell.

The destinations of the arrows have to be visible from the sources because a read or write operation to the destination is made there.

This means that the arrows can not show *into* boxes. At best, they show to the border of a box.

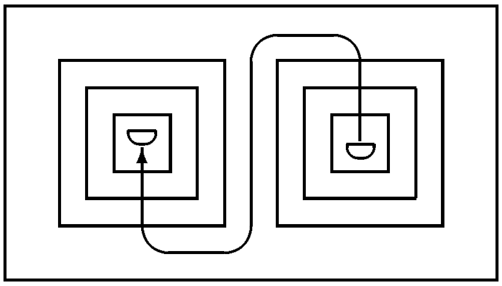

Figure 5: transport in a flat simulator

P_2: **hierarchical circuit representation:**
All arrows together build a representation of the circuit. For a hierarchical storage, the arrows of a module have to be independent of its instances. This means that the arrows may not *leave* modules because the *outside* of a module depends on the instance, but the inside does not. In our example, we only want single code for the two half-adder-cells. Thus, the code has to be independent of the two instances.

P_3: **linear runtime:**
The sum of all arrows in a figure has to be linear in the number of signals of the disintegrated circuit. A sufficient criterion for this is that all transports from a source to an access are built of a constant amount of arrows.

Figure 6 shows the transport in the hierarchical simulator. At each module border, the value of a signal is propagated by a copy operation as a procedure parameter. All arrows stay in their module. Thus the conditions P_1 and P_2 hold. Unfortunately, condition P_3 is violated: We have seen in the last section that in the circuit family H^{buffer}, the length of such a trans-

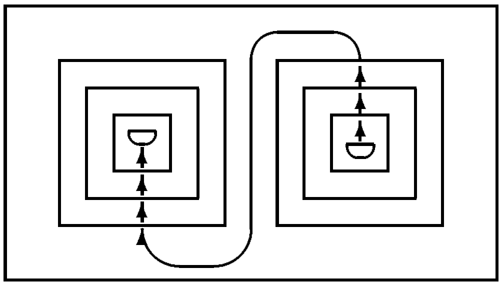

Figure 6: transport in a hierarchical simulator

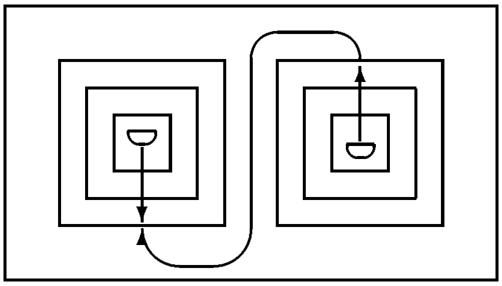

Figure 7: fast transport via visible signals

port can reach the size of the circuit and the sum over all transports can be quadratical in it.

During a closer look at the last figures, you see that each signal value first has to pass some module outputs and then some module inputs. Surely, each of this two sequences can be empty, but there will never be an output after an input in the signal transport. This observation is only true if the circuit does not contain any *feed through*. These are signals from an input to an output of a module. From the point of simulation, such signals seem to be stupid, but they are very useful to get a regular design of parametric circuits [BHK+87]. In the following, we only consider circuits with no feed throughs. This is justified in the ease of deleting feed throughs for the purpose of simulation [KS91].

The above observation allows us to find algorithms fulfilling condition P_1 in parallel with P_3 and condition P_2 in parallel with P_3.

2.1 Hierarchical signal assignment

We are looking for an algorithm with a stack oriented signal assignment and constant costs for each signal value transport. The problem is the mutual *invisibility* of the source and any drain of each signal. This prevents us from writing the signal value from the location of the source to the location of the drain. The solution to the problem lies in the unique existence of a signal instance visible from the source as well as from the drain. The source can write its value to it and the drain can read it from here later on. In figure 7, condition P_1 holds because the arrows do not point inside a module. Each transport is built up of three arrows so that condition P_3 holds.

As you can see in the following listing $flat_2$ for our full-adder example, the stack can be precomputed so that there is no need for memory allocation and stack pointer movement. To illustrate the content of the stack we use the symbols $s_1, s_2, \cdots, s_9$ and o_1, o_2 from figure 1.

$flat_2(i_1, i_2, i_3)$
signals i_1, i_2, i_3;
stack of signals sp;
begin
  $sp[2] = i_1$ **and** i_2;
  $sp[4] = i_1$ **nand** i_2;
  $sp[5] = i_1$ **nand** $sp[4]$;
  $sp[6] = sp[4]$ **nand** i_2;
  $sp[3] = sp[5]$ **nand** $sp[6]$;
  comment
    stack:

0	1	2	3	4	5	6	7
		s_1	s_5	s_2	s_3	s_4	

    In sp[2] and sp[3] there are the outputs
    of the first half adder.
    s_2, s_3 and s_4 are not needed any longer,
    so that sp[4], sp[5] and sp[6] may be reused.
  end of comment;
  $sp[4] = sp[3]$ **and** i_3;
  $sp[5] = sp[3]$ **nand** i_3;
  $sp[6] = sp[3]$ **nand** $sp[5]$;
  $sp[7] = sp[5]$ **nand** i_3;
  $sp[1] = sp[6]$ **nand** $sp[7]$;
  $sp[0] = sp[2]$ **or** $sp[4]$;
  comment
    stack:

0	1	2	3	4	5	6	7
o_1	o_2	s_1	s_5	s_6	s_7	s_8	s_9

  end of comment;
  return$(sp[0], sp[1])$;
end;

2.2 Hierarchical circuit description

We now describe how to use a hierarchical circuit description – that means hierarchical code – for a logic simulator with linear runtime. In figure 8, you find the arrow description of its signal transports. Again, we have three arrows all starting at an intermediate level. The first and last one point at the source and drain

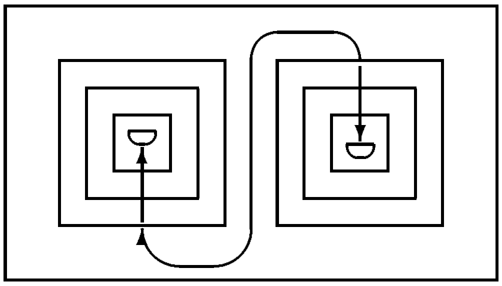

Figure 8: fast navigation in hierarchical description

of the signal while the second one connects both. The circuit description given by a representation of the arrows is hierarchical because, according P_2, the arrows only point inside of modules. Each transport consists of three arrows so that condition P_3 holds. Pointing inside the modules, the heads of the arrows are not visible from the starting points and condition P_1 is violated.

We use a flat field to store all the signal assignments equipped with a pointer for the start address of the box cell currently worked at. As the local addressing is identical for all instances of one module, we can use the same code for different instances of a module. Before starting the simulation of a box cell, the assignments of all inputs – like the one at the source of the third arrow – are pushed to their accesses. xor_2, ha_2 and fa_2 are the resulting routines for our full-adder example with the following addressing in the flat signal field.

FA	0	1	2	3	4	5	6	7	8	…
HA_1	0	1	2	3	4	5	6	7	8	
			XOR_1	0	1	2	3	4	5	
	i_1	i_2	s_1	i_1	i_2	s_2	s_3	s_4	s_5	…

…	9	10	11	12	13	14	15	16	17	18
HA_2	0	1	2	3	4	5	6	7	8	
			XOR_2	0	1	2	3	4	5	
…	s_5	i_3	s_6	s_5	i_3	s_7	s_8	s_9	o_2	o_1

$xor_2(sp)$
signal pointer sp;
begin
  $sp[2] = sp[0]$ **nand** $sp[1]$;
  $sp[3] = sp[0]$ **nand** $sp[2]$;
  $sp[4] = sp[2]$ **nand** $sp[1]$;
  $sp[5] = sp[3]$ **nand** $sp[4]$;
end;

$ha_2(sp)$
signal pointer sp;
begin
  $sp[2] = sp[0]$ **and** $sp[1]$;
  $xor_2(address\ of\ sp[3])$;
end;

$fa_2(i_2, i_2, i_3)$
signals i_1, i_2, i_3;
field of signals sp[0] $\cdots$ sp[18];
begin
  $sp[0] = sp[3] = i_1$;
  $sp[1] = sp[4] = i_2$;
  $ha_2(address\ of\ sp[0])$;
  $sp[9] = sp[12] = sp[8]$;
  $sp[10] = sp[13] = i_3$;
  $ha_2(address\ of\ sp[9])$;
  $sp[18] = sp[2]$ **or** $sp[11]$;
  return$(sp[18], sp[17])$;
end;

2.3 Is there an optimal algorithm?

We postulated three demands for a *good* hierarchical logic simulator: **hierarchical signal assignment**, **hierarchical circuit description** and **linear runtime**. For each pair of conditions, an algorithm can be found that fulfills both. What we need for total contentment is an algorithm fulfilling all three conditions.

Theorem 2 *There is no logic simulator for combinational circuits with hierarchical signal assignment, hierarchical circuit representation and linear runtime in the circuit size.*

To prove this theorem, we use the signal value transport conditions in terms of arrows from the previous section. According to condition P_1, arrows may not point into a module and according to P_2, not out of it. The combination of these conditions implies that arrows do not cross borders of modules. Thus we have to use the inputs and outputs of each module to propagate the signal values. But this is a contradiction to the third condition P_3 because the sum over all arrows can increase unlimited for arbitrary circuits and – we have seen this for the circuit family H^{buffer} – quadratically for *reasonable* ones. □

3 Practical Application

These results influenced the hierarchical version of the compiler-driven fault simulator COFS [HSS92]. Since, for fault injection, it is necessary to keep all signal engagements (or to recompute them), the algorithm with hierarchical circuit description and linear runtime has been chosen. Only the generated simulation routines are considered in the following measurements.

Figure 9 and 10 show the length of the simulation routines and the runtime of 100 000 simulations for four circuits. These are a 16 and a 256 bit carry-look-ahead adder, the mantissa handling of a 24 bit floating point adder and a 64 bit partial multiplier, all designed with Cadic [BHK+87]. They demonstrate that it is possible to get nearly the runtime of a flat simulator with the enormous code reduction effected by hierarchy.

Acknowledgements

I want to thank Prof. Dr. Dr. h.c. mult. Günter Hotz for his encouragement and support and Dr. Uwe Sparmann for helpful comments and discussions.

References

[App88] A. Appel. Simulating digital circuits with one bit per wire. *IEEE Transactions on Computer-Aided Design*, pages 987–993, 1988.

[AS87] Kurt J. Antreich and Michael H. Schulz. Accelerated fault simulation and fault grading in combinational circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):704–712, September 1987.

[Bee92] John F. Beetem. Hierarchical topological sorting of apparent loops via partioning. *IEEE Transactions on Computers*, pages 607–619, 1992.

[BHK+87] B. Becker, G. Hotz, R. Kolla, P. Molitor, and H.G. Osthof. Hierarchical design based on a calculus of nets. In *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC87)*, pages 649–653, June 1987.

[BHKS91] B. Becker, R. Hahn, R. Krieger, and U. Sparmann. Structure based methods for parallel pattern fault simulation in combinational circuits. In *Proceedings of the 2nd European Design Automation Conference (EDAC91)*, 1991.

[DG87] W. Daehn and M. Geilert. Fast fault simulation for combinational circuits by compiler driven single fault propagation. In *Proceedings 1987 International Test Conference*, pages 286–292, 1987.

[HSS92] J. Hartmann, B. Schieffer, and U. Sparmann. Cell oriented fault simulation. In *Proceedings of the European Simulation Multiconference 92*, pages 424–429, 1992.

[KS91] R. Kolla and B. Serf. The virtual feedback problem in hierarchical representations of combinational circuits. *Acta Informatica*, 28:463–476, 1991.

[Mau91] P. Maurer. Scheduling blocks for hierarchical compiled simulation of combinational circuits. *IEEE Transactions on Computer-Aided Design*, pages 184–192, 1991.

[WM89] Zhicheng Wang and Peter M. Maurer. Scheduling high-level blocks for functional simulation. In *Proceedings of the 26st ACM/IEEE Design Automation Conference*, pages 87–90, 1989.

[WS92] H. Wittmann and B. Seiss. Hierarchische Fehlersimulation in kombinatorischen Schaltungen. In *ITG-Fachbericht 122, Rechnergestützter Entwurf und Architektur mikroelektronischer Systeme*, pages 159–166, 1992.

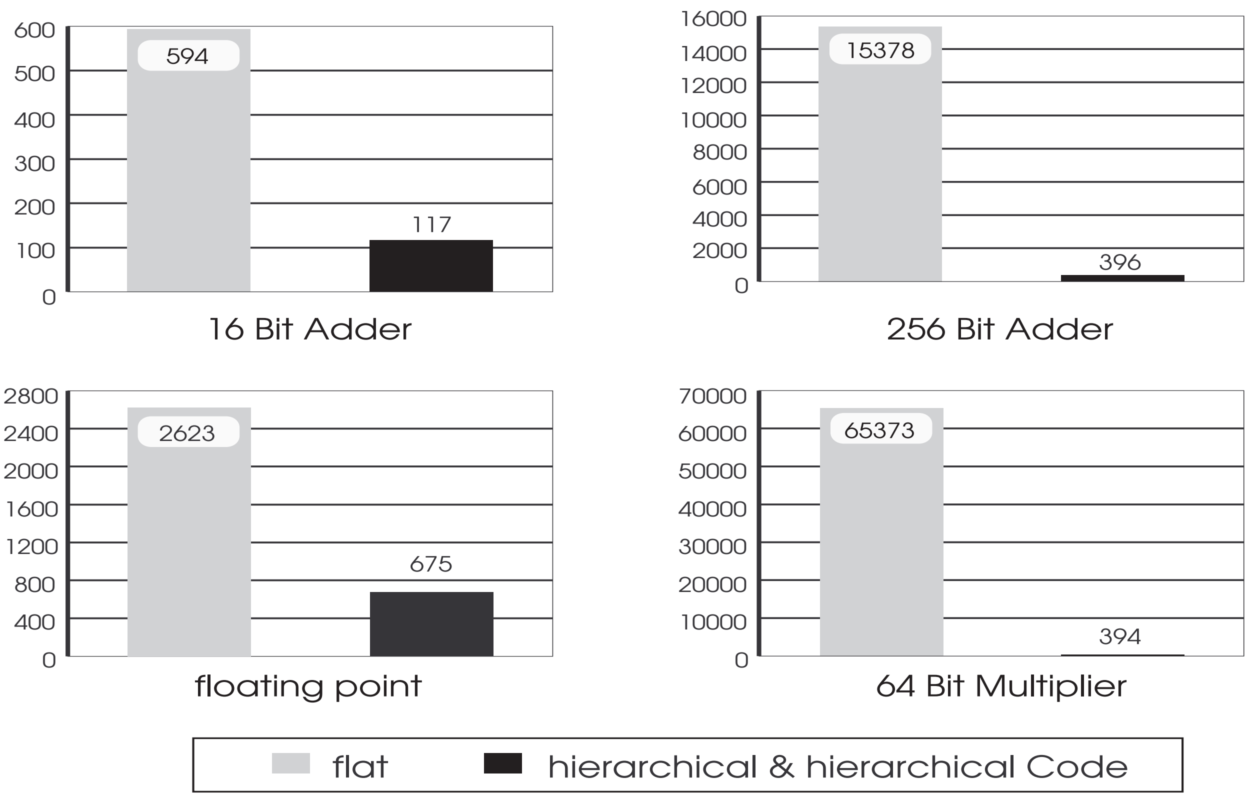

Figure 9: lines of code in simulation routine

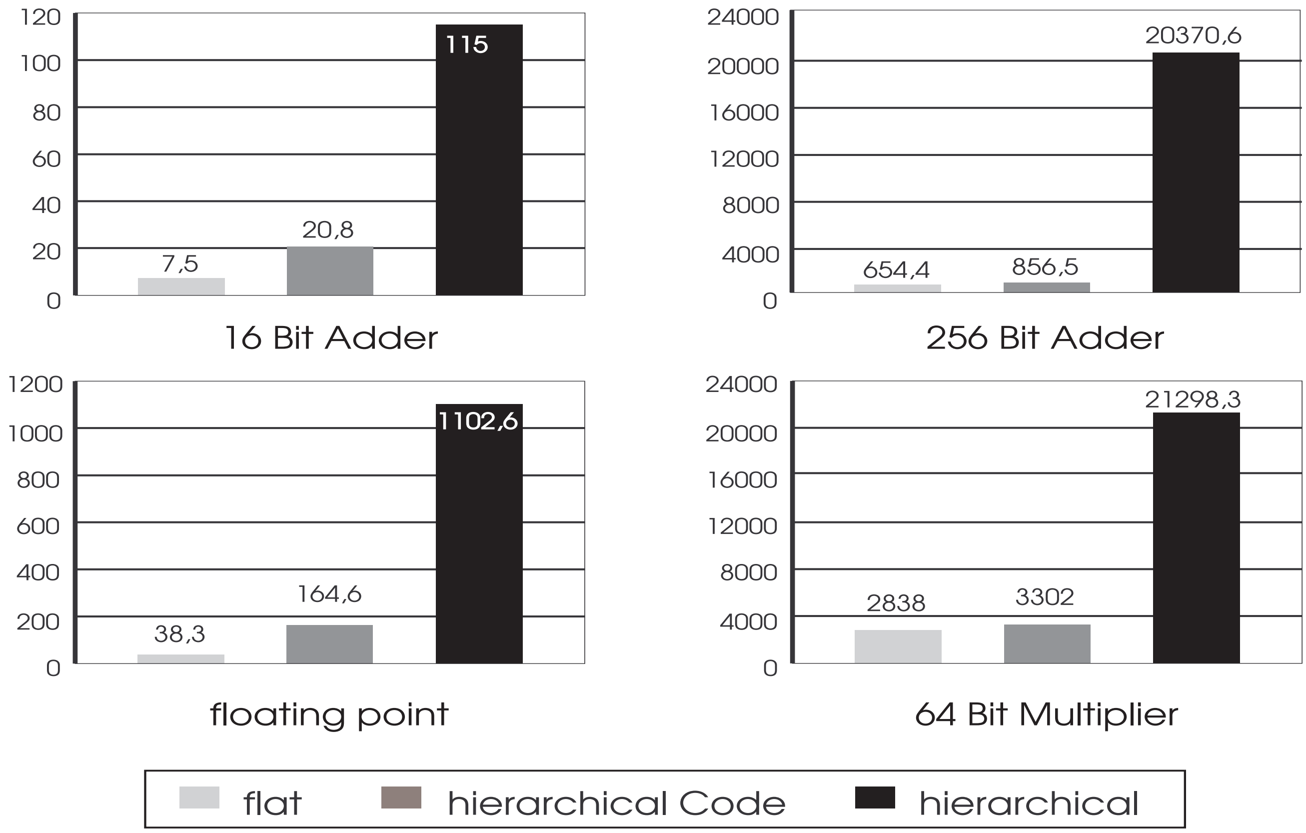

Figure 10: runtime of 100 000 simulation runs in sec