Digital Circuit Synthesis and Optimization

Leo Adberg
EECS ‘20
UC Berkeley
ladberg@berkeley.edu

ABSTRACT

The DCSO (Digital Circuit Synthesis and Optimization) project is a simulator and optimizer for digital logic. The purpose of DCSO is to make logical optimizations that current compilers and synthesizers cannot make and to make them in a transparent fashion. The DCSO RTL (register transfer level) language is designed to be similar in appearance to Verilog, though supports only a handful of features that make simulation and optimization easier, and therefore is not ready to be used in real hardware design yet. The DCSO RTL code is compiled into a logic graph (directed and acyclic, representing one clock cycle), which can be simulated and optimized over. The optimization is done by splitting the logic graph into “gadgets”, which consist of a set of nodes with defined inputs and outputs to the gadget. The optimizer creates a mapping of functionally identical gadgets and attempts to swap out each gadget with matching gadgets of lower cost.

1 INTRODUCTION

My first real experiences with digital circuits started when I took EECS 151 (Introduction to Digital Design and Integrated Circuits) at UC Berkeley. In the class, we designed a RISC-V CPU from scratch to run on a Xilinx PYNQ FPGA using Xilinx’ Vivado Design Suite, a monolithic, proprietary, and opaque piece of software. The summer after that, I spent an internship working on a hardware simulator that compiled from Verilog to x86 for fast incredibly fast simulation in software.

In both those experiences, I ran up against the limits of existing digital circuit synthesis and optimization. Existing optimizers, while smart, employ mostly rule-based reductions and pre-programmed algebraic optimizations. Because of these optimization methods, the programmer often has to work with the optimizer if they want to produce the best output. I chose to pursue DCSO because I wanted to learn how to design an optimizer myself and see what novel, better ways there are of optimizing digital circuits.

1.1 Relation to Programming Languages/Systems

Digital circuits and software are at a high level the same thing, simply with different execution models. Both do pre-defined operations on binary data, but circuits run operations in parallel while software runs operations serially. Because of this, the same optimization techniques that a compiler (or synthesizer) runs on software can also be done on hardware[1]. For example, constant folding/propagation, dead subexpression elimination, and parallelization are all optimizations that work equally well on hardware and software.

One of the differences that comes up in hardware and software is in loop execution. In software, a loop is simply another block of code that takes in an input and produces an output. Loops can be optimized and verified using preconditions, postconditions, and identifying loop invariants. Because the preconditions and postconditions uniquely identify the functionality of a block of code, loops can be unrolled, eliminated, or otherwise modified in some way that changes the number of iterations as long as the preconditions and postconditions are still met. These optimizations are not simply transferable to hardware, where the concept of a loop is much different. In hardware, the execution of a digital circuit finishes in a finite (and usually constant) amount of time because the underlying logic graph is acyclic. A true, generalized loop can only be implemented over multiple clock cycles (each clock cycle is a full evaluation of the logic graph). When designing hardware, it is usually assumed that code can not be reordered from one clock cycle to another, so optimization of a loop is not included in DCSO.

1.2 Goals of Optimization

In digital design, circuits are usually optimized to minimize cost and silicon space (which are usually tightly correlated), execution time (which is often add odds with cost/space), or some weighted combination of the two. Hardware at the smallest level can be parallelized to accomplish a shorter execution time at the expense of cost/space, which is different from software which usually requires parallelization to occur at a higher level (e.g. thread-level parallelism, machine-level parallelism) and is thus out of scope for most software optimizers.

[1] In the digital design world, “hardware” can often refer to the code used to generate digital circuits (e.g. Verilog code), and not the physical silicon itself. In this paper, “hardware” will always refer to the digital circuit (aka a directed graph of logical operations) and not a real CPU.

DCSO is intended to optimize circuits according to an arbitrary cost function without any pre-programmed rules or reduction knowledge, allowing it to optimize in ways that wouldn't be obvious otherwise or wouldn't have occurred to programmers. I will show examples of DCSO's optimizations and compare with existing optimizers.

2 DCSO DESIGN

DCSO is designed to take in an input in the form of DCSO RTL, a language similar to Verilog, or constructed using an embedding in C++, which is structurally equivalent to the RTL. The RTL code is type checked and compiled into a direct, acyclic graph of logical operations. The logic graph can be executed directly to simulate the circuit or can be optimized over.

2.1 DCSO RTL Grammar

The DCSO RTL grammar is defined as follows:

```
Program: decls stmts ;
decls: decl decls | ;
decl: type var [size] ;
type: reg | wire | input | output ;
var: [a-z]+ ;
size: int ;
stmts: stmt stmts | ;
stmt: IF (expr) { stmts } ELSE
{ stmts } ENDIF | var = expr ';' ;
expr: var | size'int | expr[int] |
expr & expr | expr '|' expr | expr ^
expr | expr + expr | expr . expr ;
```

This results in code like the following, which counts the current iteration and performs a basic operation on the inputs:

```
input x[2];
input y[2];

output z[2];
output iter[4];

reg it_reg[4];
wire xysum[2];

xysum = x + y;

if (x[0] == 0) {
  z = x[1] . y[0];
} else {
  z = xysum;
}

it_reg = it_reg + 4'1;
iter = it_reg;
```

Note the Verilog-like syntax of integer literals which must specify a bit length before the apostrophe. All variables must be declared before the body of the program. As you can see, the DCSO RTL has four types of variables (`input`, `output`, `reg`, and `wire`). A `reg` (short for register) is the only type that carries state from cycle to cycle. `input` variables are read only while `output` variables are write only. `wire` and `reg` variables can be read and written. Values written to a `wire` are available when reading the same cycle (a synchronous assignment), while values written to a `reg` are only available the next cycle (an asynchronous assignment). Each variable is also assigned a size, meaning the number of bits that it carries. Variables can only be assigned to expressions of the same size, and arithmetic operations (`+`, `&`, `|`, `^`) can only operate on inputs of the same size. The size of an expression can be modified using concatenation (the `.` operator) or subscripting to access an individual bit (the `[]` operator). An `IF` statement takes a size 1 expression as its input.

A valid Program must assign to every writeable variable and each clause of an `IF` statement must assign to the same set of variables. Nested `IF` statements are allowed. Variables cannot be assigned multiple times (excluding `IF` statements where they can of course only take one path).

One important note is that I have included addition as a primitive operation in the DCSO RTL, making it appear to be on-par with the bitwise operators. In real digital design, bitwise operators are all implemented trivially using the corresponding logic gates, but implementing an addition is not as easy. There multiple ways to implement an addition and each carry with it design considerations (e.g. ripple adder, carry-select adder). In real digital design, these considerations would need addressing, but for now I have left it as a primitive to show how optimizations can be implemented in ways that are easier to read than bitwise operations.

In the current state of DCSO, there is no parser. This means that code must be written in the C++ embedding, which is structurally the same but does not look as clean. Any RTL code in this paper has been manually translated to C++ to run (which is a relatively straightforward task).

2.2 Logic Graph Compilation

The RTL is compiled to a logic graph such that every expression in the RTL corresponds to a node in the graph. `IF` statements are represented as MUX (multiplexer) nodes, which take as input the condition and the two possible values and output the selected value. The inputs to the graph consist of every `input` variable and the state of all the `reg` variables as set by the last clock cycle (initialized to 0). The outputs of the graph consist of every `output` variable and the future state of every `reg` variable (reading and writing are two separate nodes to accommodate the asynchronous assignment). Because of the construction, the logic graph is both directed and acyclic.

2.3 Logic Simulation

Given the logic graph, simulation is fairly easy. All inputs to the graph are set and then the values of all output nodes are read. The value of a node is cached for the current cycle and is calculated by recursively reading the values of the incoming edges and performing the node's operation on them. This means that the simulation runs in time linear with the number of nodes needed to calculate the outputs, no nodes are calculated more than once, and no "dead" nodes (not upstream of an output) are calculated.

2.4 Optimization

The optimization is the most interesting part of DCSO and the only thing that sets it apart from any other digital design application. Of course, that means it is also the hardest part to design and is unfortunately in a usable but bare-bones state at the moment. At a high level, the optimizer splits the logical graph up into "gadgets". Each gadget consists of one or more nodes and the incoming and outgoing edges of those nodes. The optimizer attempts to replace gadgets in the graph with functionally equivalent but lower cost gadgets.

2.4.1 ***Gadget Finding***. Gadget selection is an interesting and open-ended problem. As gadgets can overlap and be of any size, there are many possibilities to choose from. One could choose every possible subset of connected nodes to be a gadget, but this results in an exponential number of gadgets which is obviously infeasible for any reasonably sized example. DCSO currently chooses every node and all its upstream dependencies as gadgets, which results in a linear number of gadgets and catches most obvious optimization opportunities, but it has the downside of missing opportunities in the middle of the logic graph. For example, one could write[2]:

```
    out = (((foo + bar) + baz) + (2 - baz));
```

This can be optimized to:

```
    out = (foo + bar) + 2;
```

Hypothetically, unless the optimizer manages to do the whole reduction in one step (which is harder to do), this requires an intermediate reduction of:

$$(x + y) + (z - y) \rightarrow (x + z)$$

This intermediate reduction is seemingly obvious, but without creating a gadget in the middle of the logic graph it wouldn't be possible.

2.4.2 ***Gadget Generation***. Due to the design of DCSO, gadgets can only be reduced to other gadgets. Because of this, limiting gadgets to only those existing in the logic graph significantly reduces possible routes for optimization, to the point of uselessness in a small example. To fix this, we can generate new gadgets by adding in new nodes with no output (aka "dead code"). These gadgets won't affect the logic graph or even be evaluated in simulation, yet can be used as a library of possible reductions by the optimizer. Hypothetically, this library could be generated randomly, intelligently, or in some brute force fashion. So far, DCSO does not generate this library, but it is possible to add dead nodes to the logic graph manually which will then be used for reductions.

2.4.3 ***Gadget Matching***. Once a set of gadgets is identified, they must be matched to each other to perform the reductions. This can be done in a few ways, each with their own upsides and downsides. First, the input space of each gadget can be brute forced to generate a truth table for the gadget. The truth table is then hashed and stored in a hash map so that any functionally identical gadgets can be easily matched to it. This is what DCSO currently does, but there are other matching methods that can be used in tandem for better optimization. Another method is to hash gadgets structurally. Currently in DCSO, each node calculates its string representation using its inputs' representations recursively, so this could be hashed to represent structure. Structural hashing would be a useful addition to truth table hashing because in a real-world example with large inputs it would be impossible to generate a truth table with 64 state bits. However, that gadget could be structurally identical to a gadget with only 16 state bits, which could be truth table hashed and reduced, allowing the 64 bit gadget to match and reduce as well.

3 PERFORMANCE COMPARISON

It would be ideal to compare DCSO with existing EDA (electronic design automation) tools such as Cadence, Synopsys, or Vivado, doing so is painful to the point where I will exclude them from this comparison. None of the tools are supported on Mac, most don't officially support Ubuntu, and all are only available through trials I would have to request. In addition, once installing the tools, building up small test cases and evaluating optimizer performance is non-trivial.

Instead, I will compare with software compilers which, with the help of godbolt.org, are incredibly easy to use as a reference for highly performant optimizers. This also has the added benefit of proving that DCSO, though designed for digital circuits, can also optimize software in novel ways. To keep the tests fair, I only use branchless code and write software code that is equivalent to the hardware code, down to sizes of inputs.

I also have to give Clang and GCC credit: most attempts to fool them were futile, and I am only showing the few examples I found where one of them failed at optimization. I am running both compilers with "-O3" and they were incredibly smart at optimizing most expressions I threw at them.

3.1 Expression using Equality Operator

This example assigns two variables to the parity (even-ness in this case) of x using two different methods, and then returns whether they are equal. Of course, because both variables represent the same thing, the equality check should always return true.

Software code:

[2] Yes, I know subtraction isn't an operation in the DCSO RTL. It's almost as easy to do in a digital circuit as addition so bear with me as it makes this example nice. Assume integer literals include the relevant bit length as well.

```
bool test(char x) {
    char a = (x + 1) & 1;
    char b = (x & 1) == 0;
    return a == b;
}
```

Hardware code:

```
input x[8];
wire a[8];
wire b[8];
output ret[1];

a = (x + 8'1) & 8'1;
b = 7'0 . ((x & 8'1) == 0);
ret = a == b;
```

As you can see, both inputs are functionally the same and use the same sized inputs and outputs (8 bits and 1 bit, respectively). I chose 8 bits because it's small enough that DCSO can optimize using truth table hashes, but large enough that there exists a native C type so compilers should be able to optimize for it too.

The resulting GCC output failed to optimize the code:

```
test(char):
        lea     eax, [rdi+1]
        not     edi
        and     eax, 1
        and     edi, 1
        cmp     al, dil
        sete    al
        ret
```

However, Clang was able to fully optimize the code:

```
test(char):
        mov     al, 1
        ret
```

Of course, DCSO was able to also fully reduce the circuit:

```
Reducing:  (0 . ((x & 1) == 0))
into:      ((x + 1) & 1)

Reducing:  (((x + 1) & 1) == ((x + 1)
& 1))
into:      1

*** Cycle 1 ***
ret = 1 = 1
```

For comparison, here is the DCSO output with optimization turned off:

```
*** Cycle 1 ***
ret = 1 = (((x + 1) & 1) == (0 . ((x
& 1) == 0)))
```

I suspect the reason GCC had trouble with this example was due to the equality operator, as most other expressions consisting of only bitwise operations were able to be optimized fully.

3.2 Bitwise Operations

The previous example was easy for DCSO because the final result was simple and therefore easy to find, but in this example the final result is not simply a number. It also turns out that Clang is even smarter than I thought. It took a while to find an expression that I knew *could* be optimized more than Clang was able to.

Software code:

```
char test(char x) {
    return ((x ^ 254) + 2) + x;
}
```

Hardware code:

```
input x[8];
output ret[8];

ret = ((x ^ 8'254) + 2) + x;
```

The GCC output:

```
test(char):
        mov     eax, edi
        xor     eax, -2
        lea     eax, [rax+2+rdi]
        ret
```

The Clang output:

```
test(char):
        mov     eax, edi
        xor     al, -2
        add     eax, edi
        add     al, 2
        ret
```

While the GCC output looks pretty short, upon further inspection you can see that both compilers simply did the three algebraic operations as described in the code. The "correct" optimization involves recognizing that the expression is equal to "(x << 1) & 1", which for comparison would result in this slightly faster assembly:

```
test(char):
        lea     eax, [rdi + rdi]
        and     al, 2
        ret
```

Unfortunately, DCSO also fails at this task because it doesn't know the correct expression and therefore can't add it to the gadget library. However, we can give both DCSO and Clang an assist. I gave both programs the correct answer as a hint and saw if they picked up on it:

```
char test(char x) {
    char a = (x & 1) << 1;
    return ((x ^ 254) + 2) + x;
}
```

Clang chose to emit the same assembly as before, removing the dead code instead of learning from it. DCSO added the dead node to its library and was able to verify equality with the original expression, replacing it due to the lower cost:

```
Reducing:  (((x ^ 254) + 2) + x)
into:      (x[0] . 0)

*** Cycle 1 ***
ret = 2 = (x[0] . 0)
```

This example isn't meant to show that DCSO is better than Clang (it isn't yet), but simply that if a comprehensive enough gadget library was generated, the gadget-based reduction method would provide decent results.

4 RELATED WORK

There are many other ways of performing optimization of digital logic not covered by DCSO. One of the most promising methods is using a SAT solver to generate and verify possible reductions as in [1]. At a high level, DCSO effectively serves the same purpose as a SAT solver, but uses pre-defined methods of hashing gadgets to verify correctness instead of asking a SAT solver to evaluate it. This has benefits and drawbacks, namely that it can be faster and more transparent than a SAT solver but might miss functional equivalence between gadgets due to the chosen hashing methods.

The Berkeley Verification and Synthesis Research Center (BVSRC) makes *ABC: A System for Sequential Synthesis and Verification* [2] which targets the same goals as DCSO through rewriting AIGs (And-Inverter Graphs, a way of representing digital logic with only a few primitives). Unfortunately, I just learned about ABC and although it is open source, I can't comment on how it performs optimizations because documentation is lacking and I haven't dug through the source yet. In the future, I would like to compare it with DCSO like I did earlier with GCC and Clang.

In this paper only two basic methods for checking functional equivalence of two gadgets. In reality, this is a big problem that is extensively covered in literature. For example, [3] describes current state-of-the-art methods of verifying functional equivalence and provides much faster methods. However, these methods (and most others) rely on checking equivalence between each pair of circuits at a time, and thus matching N gadgets takes $O(N^2)$ time no matter how fast the comparison is done. Using a hash-based method allows for matching N gadgets in $O(N)$ time, which allows a significantly larger set of possible gadgets and therefore possible reductions.

5 CONCLUSION

I have presented DCSO (Digital Circuit Synthesis and Optimization), a framework for optimizing digital circuits whose techniques carry over into optimization of programming languages and systems. DCSO is still in its early stages and has much work to be done to make it useful, but it shows promise.

I believe that more research into digital circuit and program optimization through gadget-based hash matching should be done. It allows for nearly limitless optimization potential in linear time, which is a huge improvement over current methods. Hash matching also relies on decent hashing algorithms to be chosen. Using multiple hashing algorithms concurrently is always beneficial, so I am interested in seeing what other possibilities there are in addition to truth table and structural hashing.

ACKNOWLEDGMENTS

The DCSO project was developed for CS 263 at UC Berkeley over the Fall 2019 semester. Many thanks to Alvin Cheung for teaching a great class!

REFERENCES

[1] Tobias Welp, Smita Krishnaswamy, and Andreas Kuehlmann. 2012. Generalized SAT-sweeping for post-mapping optimization. In Proceedings of the 49th Annual Design Automation Conference (DAC '12). ACM, New York, NY, USA, 814-819. DOI: https://doi-org.libproxy.berkeley.edu/10.1145/2228360.2228507

[2] Berkeley Logic Synthesis and Verification Group, "A System for Sequential Synthesis and Verification". https://people.eecs.berkeley.edu/~alanmi/abc/

[3] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. 2006. Improvements to combinational equivalence checking. In Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06). ACM, New York, NY, USA, 836-843. DOI: https://doi.org/10.1145/1233501.1233679