Hardware compilation of non-deterministic choice

Matthew Naylor and Simon Moore

***Abstract*—We present a high-level synthesis (HLS) compiler for a simple imperative language extended with *non-deterministic choice*, and apply it to three combinatorial search problems: N-queens, optimal Golomb rulers, and propositional satisfiability. The resulting circuits, running on a medium-sized FPGA, are 10-100 times faster than corresponding software implementations running on a modern PC. This work represents a step towards HLS of programs that exploit irregular, dynamic parallelism.**

I. Introduction

Non-deterministic choice is a program construct that specifies, at some point in a program, two alternative execution paths. Unlike an if-then-else construct, the choice is not resolved by evaluating a boolean condition. Instead, the program may proceed by executing any *either* alternative. A program may now have have more than one possible result. To illustrate, consider the following program, where S_0 `?` S_1 denotes non-deterministic choice between statements S_0 and S_1, and S_0 `;` S_1 denotes sequential composition.

```
x := 1 ; (x := x+1 ? x := x+2) ; x := x*x
```

This program has two possible results, one in which the final state contains $\texttt{x} = 4$ and the other in which it contains $\texttt{x} = 9$.

Non-deterministic choice is particularly useful when implementing search algorithms, where a program must consider various paths that may lead to a solution. It allows a choice to be specified without providing low-level details about how the choice is implemented. Indeed, the choice can be implemented in various ways. For example, sequentially, by executing the first path and, if that fails, backtracking and executing the second. Or both choices can be explored in parallel, by process creation akin to UNIX `fork()`. This dual interpretation opens up the possibility of *automatic parallelisation*: program performance can be improved by introducing more processors, assuming a sufficient amount of non-determinism is present.

Although non-deterministic choice does not entail any costly synchonrisations between parallel processes, it still poses challenges for parallel computing. First, the overhead of process creation can easily outweigh the benefit of parallel execution, especially for short-lived processes. Second, the workload must be efficiently balanced across hundreds or even thousands of processors to acheive a high degree of scalability.

In this paper, we respond to these challenges by compiling non-deterministic programs down to programmable hardware, customised for efficient process creation and load-balancing. We first introduce our source language – *Wildfire* – by example, presenting programs to solve three well-known search problems. After that, we describe the main compilation rules that map Wildfire programs to FPGA circuits. Finally, we analyse the performance of the resulting circuits, and discuss the strengths and weaknesses of the approach.

II. Example 1: N-queens

N-queens is a classic programming problem often used in teaching, and makes for a good introduction to the Wildfire language. The problem is to place N queens on an $N \times N$ chess board such that no queen is attacked by any other queen. We consider a particular version of the problem in which the goal is to count the number of solutions.

One of the most elegant ways to solve this problem is to use a bit-vector encoding, which has been proposed independently by Richards [1] and Zongyan [2]. The basic idea is to visit each row of a chess board, one at a time, and to use three N-bit vectors to track the squares on the current row that are under attack due to: (1) a left diagonal attack, (2) a right diagonal attack, and (3) a column attack. Let us call these three vectors `l`, `r`, and `c`. The squares on the current row on which a queen may safely be placed can be computed simply by `~(l|r|c)`. Now suppose that we wish to place a queen at posistion `choice` on the current row, where `choice` is a one-hot bit-vector. We can place the queen and move to the next row by the following three assignments.

```
l  :=  (l|choice) << 1 ;
r  :=  (r|choice) >> 1 ;
c  :=  (c|choice)
```

All that remains is to explore all valid choices of queen placement on each row. Usually this is achieved using recursion, but if instead we use non-deterministic choice then we enable a simple route to automatic parallelisation. Here is the full N-queens solver encoded in Wildfire:

```
0  -- Number of queens & dimensions of board
1  const N = 18
2
3  -- State of squares on the current row
4  var safe   : bit(N) -- Safe squares
5  var l      : bit(N) -- Attacked (left diag)
6  var r      : bit(N) -- Attacked (right diag)
7  var c      : bit(N) -- Attacked (column)
8  var choice : bit(N) -- Chosen square
9
10 safe := ~0 ; -- Initially, all squares safe
11 while safe != 0 do
12   -- Isolate the first hot bit in safe
13   choice := safe & (~safe + 1) ;
14   -- Place a queen here & move to next row
15      ( l := (l|choice) << 1
16     || r := (r|choice) >> 1
17     || c := c|choice
18      ; safe := ~(l|r|c) )
19   -- Or, do not place a queen here
20   ?  ( safe := safe & ~choice )
21 end ;
22 -- Fail unless every column has a queen
23 if c != ~0 then fail end
```

Notice that Wildfire supports fork-join parallel composition of the form S_0 `||` S_1. It binds tighter than sequential composition, so the statement S_0 `||` S_1 `;` S_2 is equivalent to

$$(S_0 \texttt{ || } S_1) \texttt{ ; } S_2$$

that is, perform S_0 and S_1 in parallel and, after both have finished, perform S_2.

To define non-deterministic choice, it is helpful to consider the program as, initially, running in a *single world*. When we execute S_0 `?` S_1, we *fork the world*: in one world we proceed by executing S_0 and, in the other, by executing S_1. Any statement following a non-deterministic choice is executed in both worlds, as captured by the following property.

$$(S_0 \texttt{ ? } S_1) \texttt{ ; } S_2 \;=\; (S_0 \texttt{ ; } S_2) \texttt{ ? } (S_1 \texttt{ ; } S_2)$$

A second Wildfire construct related to non-determinsitic choice is the **`fail`** construct. Whereas S_0 `?` S_1 creates a new world, **`fail`** destroys the world in which it is executed. It is referred to as the *identity* of `?` because it satisifies the following properties.

$$S \texttt{ ? } \textbf{fail} \;=\; S$$
$$\textbf{fail} \texttt{ ? } S \;=\; S$$

The result of a non-deterministic program is therefore a *multi-set* containing the final state of every non-failing world. When running a Wildfire program on FPGA, one of the outputs emitted by default is the size of this set, which in this example is the number of solutions to the N-queens problem.

In our N-queens program, we use non-deterministic choice (line 20) to explore the possibility of both placing and not placing a queen on a particular square. If we are ever in a world where there are no safe squares on the current row, and we have not yet placed N queens, then we **`fail`** (line 23).

III. Example 2: optimal Golomb rulers

Let us move to a new example, finding *Golomb rulers*, which is perhaps of greater practical value [3] than N-queens. A Golomb ruler of length n with m marks is a ruler in which the distance between any two marks is distinct and the maximum distance between any two marks is n. For example, the ruler with marks at positions $0, 1, 4, 6$ is a 4-mark Golomb ruler of length 6. The shortest possible Golomb ruler with m marks is called an *optimal Golomb ruler* of order m (OGR-m).

Again, a neat way to solve this problem is to use a bit-vector encoding, as proposed by McCracken [3]. The idea is to represent a `ruler` as a bit-vector with each 1 or 0 denoting the presence or absence of a mark at each position. The set `dist` of all distances measureable by the ruler can also be represented as a bit-vector. The key property relating these two bit-vectors is that the new distances measurable after prefixing a 1 to a given ruler is equal to the ruler itself. And the key question is: if we prefix a 1 to a given ruler, does the new ruler measure any distances that were also measurable by the old ruler? This can be answered by simply computing the value of `ruler&dist`. If the result is zero, then we can prefix a mark to the ruler by computing `(ruler<<1)|1` and update the distance set to `dist|ruler`.

The above reasoning leads to a simple loop that starts with a zero-length ruler and lengthens it by one unit on each iteration, either by prefixing a 1 (adding a mark) or by prefixing a 0 (not adding a mark). A mark can only be added when the new measurable distances are distinct from the old ones. When it is valid to add a mark, we use non-deterministic choice to consider the possibility of both adding and not adding it.

```
0  -- Find rulers with NumMarks and MaxLength
1  const NumMarks  = 14
2  const MaxLength = 126
3
4  -- Number of bits needed to represent ruler
5  const N = MaxLength + 1
6
7  -- Program state
8  var ruler : bit(N) = 1 -- Positions of marks
9  var marks : bit(5) = 1 -- Number of marks
10 var dist  : bit(N) = 0 -- Distances measured
11
12 while marks != NumMarks do
13   -- Fail if end of ruler reached
14   if msb(ruler) == 1 then fail end ;
15   -- Can a mark be placed here?
16   if (ruler & dist) == 0 then
17       -- Put a mark here
18       (marks := marks + 1 ||
19        dist  := dist | ruler ;
20        ruler := (ruler << 1) | 1)
21       -- Or, do not put a mark here
22     ? (ruler := ruler << 1)
23   else
24     ruler := ruler << 1
25   end
26 end
```

This Wildfire program can be used to find Golomb rulers of length n with m marks, or to prove that one does not exist. Running the program repeatedly, starting with $n = 1$ and incrementing n each time, OGR-m can be found for a given m.

IV. Example 3: propositional satisfiability

For our third and final example, let us consider the propositional satisfiability (SAT) problem. Given a propositional formula, typically in conjunctive normal form (CNF), the problem is to determine if there exists an assigment to the variables that satisfies the formula. CNF is a logical form consisting of a set of conjuncts (also called clauses), where each conjunct is a set of disjuncts (also called literals), and each disjunct is a variable with either positive or negative polarity. For example:

$$(v_0 \vee v_1 \vee v_2) \wedge (\neg v_0 \vee \neg v_1) \wedge (v_1 \vee v_2)$$

A classic SAT solving algorithm from 1962, upon which many modern solvers are still based, is the DPLL algorithm [4]. The three main elements of DPLL are: (1) a branching heuristic, for choosing the next variable to be assigned; (2) a search strategy, that explores assignments of truth values to variables; and (3)

an inference rule known as *unit propagation*. Unit propagation uses the simple observation that, if a clause exists containing exactly one unassigned variable, then the value of that variable can be inferred. For example, in the above formula, if $v_1 \mapsto 0$ then, due to the third clause, we can infer that $v_2 \mapsto 1$.

We have implemented a DPLL-based solver in Wildfire, shown in Appendix A. It uses Wildfire's non-determinstic choice construct to implement the search strategy, and a static variable order, specified as part of the input formula[1], for branching. Since CNF formulae can be large, involving many variables, our solver makes extensive use of Wildfire *arrays*. For example, we represent the mapping from variables to values using the array

```
var vars : VarId -> Value
```

where `VarId` represents a unique variable id bounded by the maximum number of variables supported (`2^LogMaxVars`)

```
type VarId = bit(LogMaxVars)
```

and `Value` represents the value of a variable:

```
enum Value = Unbound | Zero | One | End
```

A variable's value is either unassigned (`Unbound`), or a truth value (`Zero` or `One`), or a sentinel marking the end of the array of variables (`End`).

V. Compilation scheme

All Wildfire constructs except for non-deterministic choice and `fail` are compiled in a standard way according to Page and Luk [6]. However, whereas Page and Luk compile a program to a single program instance in hardware, we compile a program to *multiple* program instances – as many as can fit on the target FPGA. We refer to each program instance as a *processor*. Processors are connected according to an arbitrary topology specified at compile-time. Specifically, the topology is defined by $\mathcal{N}(p)$, which maps a processor id p to a set of neighbouring processor ids.

Now, to execute a statement S_0 `?` S_1 on processor p, one of the following steps is taken.

1) If a neighbouring processor $q \in \mathcal{N}(p)$ is idle, then the state of p is copied to q. Subsequently, p proceeds by executing S_0 and q proceeds by executing S_1. In this case we say that p *spawns* S_1.
2) If no neighbour of p is idle, then p proceeds sequentially by executing S_0 and when that terminates (either by reaching the end of the program or by executing `fail`), p backtracks, using a stack to undo all state changes since S_0 started, and then proceeds by executing S_1.

One potential drawback of this scheme is that opportunities for parallelism can be lost. For example, suppose that a processor p has no idle neighbours and therefore executes S_0 `?` S_1 using the sequential (backtracking) method. Now suppose, moments after this decision is made and before S_1 has started, a neighbour q becomes idle and is ready for new work. In principle, q could start executing S_1 but in the above scheme the opportunity is lost. This is not a major concern when programs contain large amounts of non-determinism, such as the ones presented in this paper.

[1]We use SatZoo's algorithm for producing static variables orders [5].

Acquiring and releasing locks Each processor is associated with a lock indicating whether or not that processor is busy (locked) or idle (unlocked). To execute S_0 `?` S_1 on processor p, we first try to acquire the lock of each neighbour of p in parallel. If one or more locks are sucessfully acquired then the processor corresponding to one of them is instructed to start executing S_1 and the other acquired locks are immediately released. If no locks are acquired then S_0 `?` S_1 is handled sequentially. The process of acquiring neighbouring locks (and releasing the unused ones) is a single-cycle operation. When a processor terminates and has no backtracking to do, its lock is released and it becomes available for new work.

Copying state When processor p spawns a statement S_1 on a neighbouring processor q, the *live* state of p must be copied to q. First, all of p's register variables that are *live-in* to S_1 are copied to q in a single clock cycle. Second, all *read-write* arrays, which are implemented using dual-port block RAMs, are also copied to q in parallel. Array copying is generally less efficient than register copying because the array data must be accessed sequentially. For simplicity, we use one block RAM port to implement array accesses by the *program*, and the other to *copy* data between processors during spawning. Copying can be optimised by making the copying port wider than the program port. For example, the program port of the `vars` array in our SAT solver is only 2 bits wide but the compiler uses a copying port that is 16 bits wide, so 8 array elements can be copied per cycle. In general, we make the copying port as wide as possible without increasing the number of block RAMs needed to implement the array.

Note that we distinguish between *read-write* arrays and *read-only* arrays. Read-only arrays do not need to be copied, so both block RAM ports are available for program access and a single read-only array can be shared by two processors. In fact, the compiler permits each read-only port to be shared by any number of processors using an arbiter. Consequently read-only arrays can generally be much larger in size than read-write arrays before running out of block RAM resources. This is very useful in our SAT solver because we can store a large propositional formula using a read-only array. The degree of sharing is specified using an `opt` declaration in the source program. For example, `opt ProcessorsPerROM = 8` instructs the compiler to share each read-only array between 8 processors.

Saving and restoring state In the case where all neighbouring processors are busy, S_0 `?` S_1 is handled sequentially by backtracking. To enable backtracking, all register variables live-in to S_1 are pushed onto a processor-local stack, along with a pointer to S_1. When execution of the first path finishes, backtracking is achieved by by popping the stack, and the second path is then executed as if first path never had. The amount of data that can be pushed-to or popped-from

	Procs	Speedup	PIU	ALMs	BRAM
18-queens	1	1×	6%	0.001%	0.001%
Torus	625	455×	70%	83%	11%
Butterfly	480	500×	76%	63%	8%
OGR-14	1	1×	9%	0.003%	0.001%
Torus	196	200×	86%	71%	8%
Butterfly	144	148×	89%	52%	6%
SAT-c85	1	1×	6%	0.002%	0.001%
Torus	484	114×	45%	63%	23%
Butterfly	388	281×	59%	50%	18%

Fig. 1. Performance of Wildfire-generated hardware on a DE5-NET FPGA for various numbers of processors arranged using torus and butterfly topologies. PIU stands for Peak Interconnect Usage. SAT-c85 and 18-queens both clock at 200MHz whereas OGR-14 clocks at 150MHz.

the stack on each clock cycle depends on the stack width, which is controlled by an `opt` declaration. For example, `opt StackWidth = 40` indicates a 40-bit wide stack. If the total amount of live data at a choice point exceeds the `StackWidth` then multiple clock cycles will be needed to save or restore it. Consequently, for programs involving only register variables and no array variables, the overhead of sequential backtracking is greater than the overhead of parallel spawning. This opens the possibility of super-linear speed-ups.

Backtracking on array variables is handled a bit differently. On every array assigment $a[i] := x$, the old value of $a[i]$ is saved onto a stack associated with that array. When the path terminates, all array assignments since the last choice point are undone by popping the stack.

VI. Performance analysis

Figure 1 shows the performance of Wildfire hardware on a DE5-Net FPGA board for three benchmark problems: (1) **18-queens** counts the number of solutions to the 18-queens problem; (2) **OGR-14** proves that there is no Golomb ruler of 14 marks with a length of 126 or less; and (3) **SAT-c85** proves that the formula `sgen1-unsat-85-100` (a standard crafted benchmark), with 85 variables and 540 literals, is unsatisfiable.

Two different network topologies are considered: *torus* and *butterfly*. The maximum path length between any two processors is $O(\sqrt{n})$ in the torus but only $O(log(n))$ in the butterfly [7], allowing work to diffuse through the network more quickly. Indeed, Figure 1 shows that performance scales better using the butterfly. On the other hand, each processor in the torus requires only short connections to its neighbours, reducing pressure on the FPGA interconnect. Consequently, more processors can be packed onto an FPGA while maintaining a respectable clock frequency.

The only application that does not scale perfectly is SAT-c85. We attribute this to the use of array variables which are costly to copy, hence the overhead of process creation starts to outweigh the overhead of backtracking. To combat this, we could attempt to synthesise a wider copying port but in general that can result too many BRAMs being used.

Figure 2 compares Wildfire programs running on FPGA against like-for-like software adaptations of the Wildfire programs running on a PC. These software adaptations are written in C and use recursion rather than non-deterministic choice to implement the search algorithm. The biggest performance gap arises in the SAT-c85 example, which we attribute to intensive array access. On FPGA, the combined memory bandwidth of the parallel block RAMs exceeds the memory bandwidth of the PC, sustaining higher performance. Note, however, that our simple DPLL-based solver lacks many important features found in modern SAT solvers, such as non-chronological backtracking, activity-based branching, and clause learning. Many of these features exploit sequential execution, and are not straightforwardly parallelised. On this particular problem instance (SAT-c85), our Wildfire-generated DPLL solver outperforms MiniSAT, a modern solver, by a factor of $15\times$, but there are many other problem instances for which it is significantly slower.

	PC (s)	FPGA (s)	Speedup
18-queens	498.0	9.5	52×
OGR-14	389.0	18.4	21×
SAT-c85	2661.1	28.0	95×

Fig. 2. Wall-clock performance of Wildfire-generated hardware on a DE5-NET FPGA and corresponding PC-based implementations on a single core of an Intel Core i7-6770HQ processor.

VII. Conclusion and future work

High-level synthesis of irregular parallel programs can yield excellent results. We have seen this in the context of non-deterministic choice, a high-level programming construct that allows a simple form of automatic parallelisation. FPGAs are an attractive target for such programs because they can be tailored for fast process creation and load balancing. On the negative side, we have relied heavily on parallel on-chip memory resources, which are limited in terms of capacity. Even if their capacities are increased, the cost of copying larger memories during process creation can hinder performance scaling. As a result, the compiler works best on fairly low-memory applications. Two possible avenues for future work are: (1) to extend the Wildfire compiler to target multi-FPGA systems, and (2) to develop a finite-domain constraint solver using Wildfire and use as a backend for the standard MiniZinc language. The Wildfire compiler and example applications are freely available and open-source.

References

[1] M. Richards, *Backtracking Algorithms in MCPL using Bit Patterns and Recursion*, Tech Report 433, University of Cambridge, July 1997.

[2] Q. Zongyan, *Bit-vector Encoding of N-queen Problem*, SIGPLAN Notices, vol. 37, num. 2, February 2002.

[3] A. Dollas, W. T. Rankin, D. McCracken. *A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler*, IEEE Transactions on Information Theory, vol. 44, January 1998.

[4] M. Davis, G. Logemann, D. Loveland. *A machine program for theorem proving*, in Communiations of the ACM, vol. 5, 1962.

[5] N. Eén and N. Sörensson. *An Extensible SAT-solver*, Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003.

[6] I. Page and W. Luk, *Compiling Occam into Field-Programmable Gate Arrays*, Abingdon EE&CS Books, 1991.

[7] W. F. Burton and R. M. Sleep, *Executing Functional Programs on a Virtual Tree of Processors*, Functional Programming and Computer Architecture (FPCA), 1981.

APPENDIX: SAT SOLVER IN WILDFIRE

Here is a basic SAT solver with unit propagation, assuming a static variable ordering, coded in Wildfire.

```
-- Up to 128 variables
const LogMaxVars = 7
-- Up to 1024 literals
const LogMaxLits = 10
-- Propagation stack depth of 512
const LogStackDepth = 9

-- Types
type VarId      = bit(LogMaxVars)
type LitId      = bit(LogMaxLits)
type StackIndex = bit(LogStackDepth)
enum Value      = Unbound | Zero | One | End
struct Lit      = { neg         : bit(1)
                  , id          : VarId
                  , next        : LitId
                  , endOfClause : bit(1)
                  , finalLit    : bit(1) }

-- Array of literals in CNF
var lits  : LitId -> Lit   = "lits.mif"
-- First clause containing given variable
var first : VarId -> LitId = "first.mif"
-- Mapping from variables to values
var vars  : VarId -> Value = "vars.mif"
-- Temporary unit propagation stack
-- (The => operator denotes an array that
--  is never live at a choice point)
var stk   : StackIndex => VarId

-- Registers
var solved      : bit(1)
var picked      : bit(1)
var sat         : bit(1)
var lit         : Lit
var unit        : Lit
var val         : Value
var v           : VarId = 1
var w           : VarId
var i           : LitId
var j           : LitId
var sp          : StackIndex
var unbound     : bit(2)

-- Solver
while ~solved do
 -- Pick unbound variable
 picked := 0 ;
 while ~picked do
   val := vars[v] ;
   if (val == Unbound) | (val == End)
     then picked := 1 else v := v+1 end
 end ;

 -- Have all variables been assigned?
 if val == End then solved := 1 else
   -- Non-deterministic choice
   val := Zero ? val := One ;

   -- Assign variable and push to stack
   vars[v] := val || stk[0] := v || sp := 1 ;

   while sp != 0 do
     -- Pop stack
     w := stk[sp-1] ; sp := sp-1 ||
     -- Lookup first clause containing w
     i := first[w] ;

     -- Loop over each clause containing w
     while i != ~0 do
       -- Loop over each literal in clause
       j := i || lit := 0 ||
       unbound := 0 || sat := 0 ;
       while ~lit.endOfClause do
         lit := lits[j] ;
         val := vars[lit.id] ;
         if lit.id == w then
           i := lit.next end ||
         if (val == One) & ~lit.neg |
            (val == Zero) & lit.neg then
             sat := 1 end ||
         if (val == Unbound) &
            (unbound != 2) then
           unbound := unbound+1 ||
           unit := lit
         end || j := j+1
       end ;

       -- Fail if clause is unsatisfiable
       if ~sat & (unbound == 0)
         then fail end ;

       -- Unit propagation
       if ~sat & (unbound == 1) then
         vars[unit.id] :=
           cond(unit.neg, Zero, One) ||
         stk[sp] := unit.id ; sp := sp+1
       end
     end
   end
 end
end
```