The P2NC Project
Or How Much Parallelism is There?

Yosi Ben-Asher* Gady Haber†

June 22, 2000

1 Abstract

- We consider the problem of speeding up the execution of sequential programs by means of parallelization. Good solutions to this problem are essential to the possibility of keep producing computers that are faster than those available today.
- Modern processors relay on Instruction Level Parallelism (ILP) as a way to speedup the execution of sequential programs. However, experimental studies suggest that ILP might be limited.
- The P2NC compiles sequential programs into boolean circuits and then executes them is parallel using a probabilistic algorithm.
- It is based on the observation that parallelism is best exposed when programs are compiled into circuits.

*Computer Sci. dep. Haifa University, Haifa.

†I.B.M research lab, Haifa

The goals of the P2NC project are:

- To produce upper bounds to the amount of parallelism that can be potentially extracted from actual programs using ILP techniques.
- Evaluate the potential to improve ILP involved with direct compilation of programs to circuits.
- The probabilistic algorithm used by P2NC can lead to a new type of CPU which is not based on the Von-Newman architecture. Moreover, fast evaluation times of this algorithm may suggest ways to attack the well known circuit evaluation problem, namely, constructing a fast circuit that can evaluate polynomial circuits in logarithmic number of parallel steps.

An initial version of P2NC is currently being implemented and we already have some preliminary results.

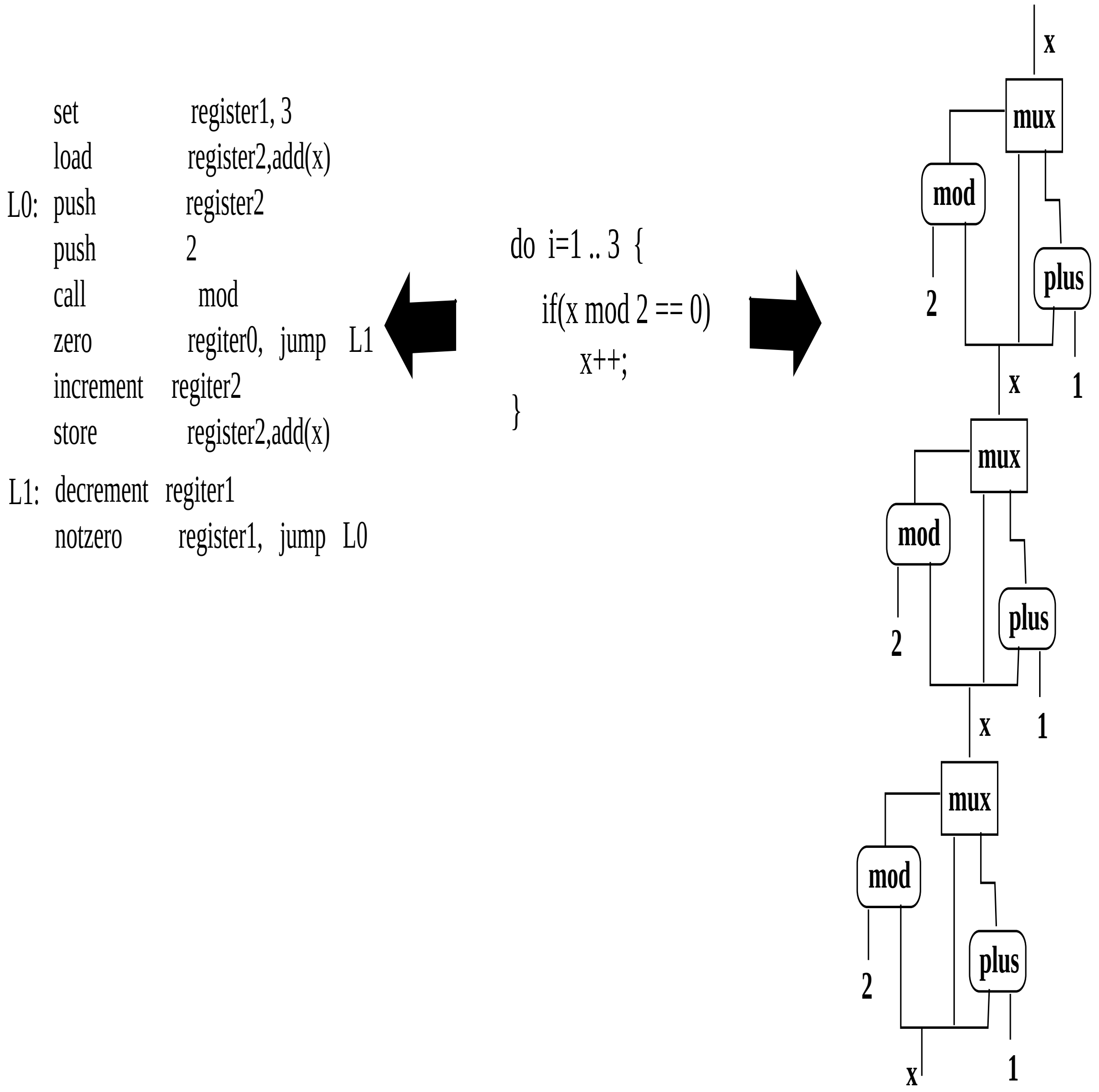


Figure 1: *Compiling into circuit versus compiling to assembler instructions.*

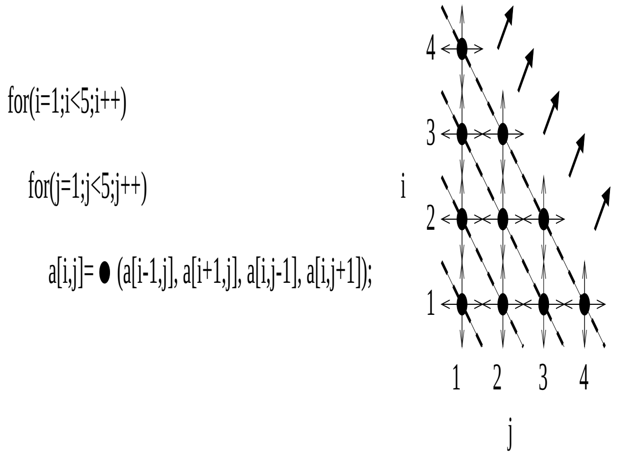
P2NC is a tool which transforms sequential loop based programs into algebraic circuits and then to boolean circuits. We say that a circuit $C^r(I)$ is a "translation" of a program r executed with input I if:

- for every bit value computed in the run of $r(I)$ there is a unique gate in $C^r(I)$ containing this value.
- The inputs of every node in $C^r(I)$ are either data-bits of I or internal nodes of $C^r(I)$.
- The size of $C^r(I)$ is less than $O(f(n))$ where $f(n)$ is the maximal number of steps executed by r on any input of size n.

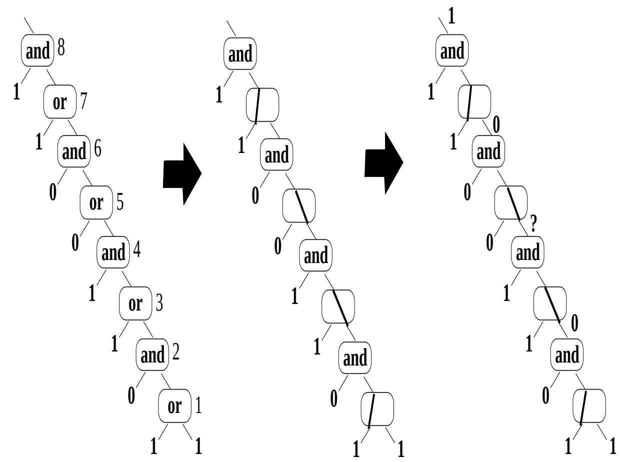
The resulted boolean circuits are evaluated by a probabilistic parallel algorithm. This stands in contrast to the Von-Newman architecture wherein programs are compiled to a sequence of RAM machine instructions which are executed by an interpreter implemented in hardware (namely the CPU). Figure 1 demonstrates the possibility to directly compile a program into algebraic gates $('+','mod','mux')$ versus the usual way of compiling into a small fixed set of assembler instructions.

Apart from executing the circuit, P2NC also accumulates statistics regarding the potential for parallelism encapsulated in the circuit. There are two parameters:

degree of parallelism (DP)- is the ratio between the size of the circuit and its depth. We will show that ILP techniques can not do better than DP as any path in the circuit represents real dependences in the program.



Potential speedup (PS)- measures the ability to speedup the execution of a program using probabilistic methods to evaluate the circuit. PS is the ratio between the depth of the circuit (which is the minimal time for evaluating the circuit) and the number of iterations needed by the probabilistic algorithm.



Why P2NC

It is estimated that within the next five years hardware manufactures will be able to increase the number of silicon gates within a factor of 100.

Speeding up the execution of sequential programs can be viewed as an effort laying between two ends:

- Let c be an encoding of a circuit of size $n^{O(1)}$ and $I = \{0,1\}^n$ an input to c. The Circuit Evaluation PROBLEM (CVP) [**?**] is whether for any n, c, i, there is a circuit C with depth of $\log^{O(1)} n$ that can evaluate c (i.e., $C(c, I) = c(I)$. CVP is a p-complete problem [**?**] and even its approximation is known to be hard, thus we look for practical versions of CVP circuits.
- Practical effort to develop a compiler+hardware techniques (known as ILP) that can reorder the instructions of a sequential program so that they can be executed in parallel using a CPU with several processing units inside the CPU. The main problem here is to overcome input/output/control dependencies between instructions. However, as will be shown there are indications that ILP is limited in its ability to extract parallelism.

The approach

From one hand, had the CVP problem been solved, then hardware manufactures would have replaced the CPU by such a circuit. On the other hand, if ILP techniques can really extract any desired amount of parallelism, then, obviously CVP is solved: just transform c to a program, compile it, and execute it on a multiple processing units CPU. Consequently, we believe that both ends should be combined to form one solution, capable to extract more parallelism than both these ends.

This is exactly what the P2NC project is all about, compiling sequential programs into circuits and then evaluating them by circuits that increase the amount of parallelism beyond what is exposed by the compilation.

We intend to conduct a sequence of experiments measuring the DP and PS of sequential programs for various CVP probabilistic algorithms and compilation techniques to increase ILP. There are for possible results:

Low DP and low PS values will indicate that the P2NC idea of combining CVP and ILP can not be used to speedup sequential programs.

High DP and low PS values will indicate that ILP might be sufficient and that perhaps compilation to circuits can be useful just for speeding up sequential programs.

Low DP and high PS values will indicate that fast evaluation techniques of circuit are crucial to speeding up the execution of sequential programs.

High DP and high PS values will indicate that both directions should suffice and there should be no special effort to combine them.

Hypothetically, a practical system based on the P2NC tools will include a compiler from high level languages to circuits, a CPU that is working using a probabilistic mechanism to evaluate circuits (e.g., like the one used in figure **??**).

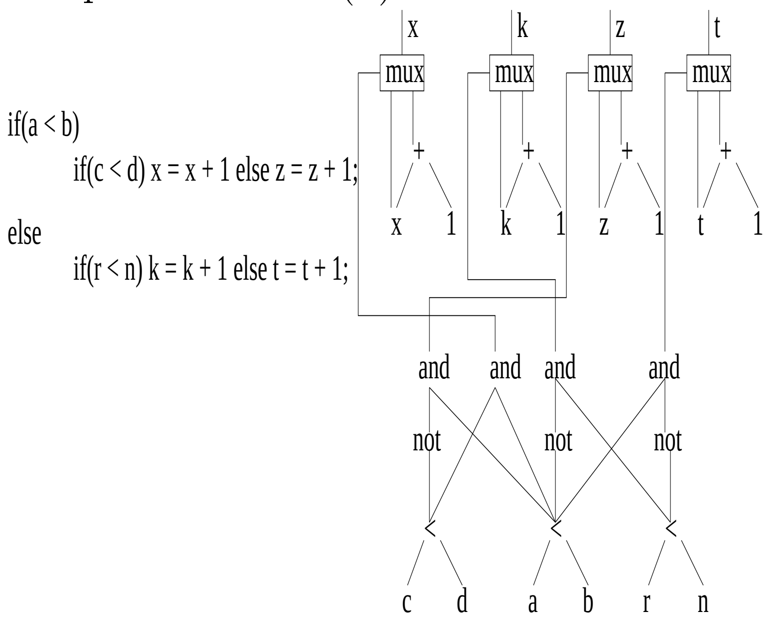
ILP is "exposed" and bounded by $C^r(I)$

Loop Unrolling, as P2NC constructs $C^r(I)$ by unrolling all the loops in the original program. Even while-loops are unrolled using the $break - statment$.

Compiler/hardware scheduling, where the compiler/hardware groups independent instructions. Exposed by the parallel evaluation of $C^r(I)$.

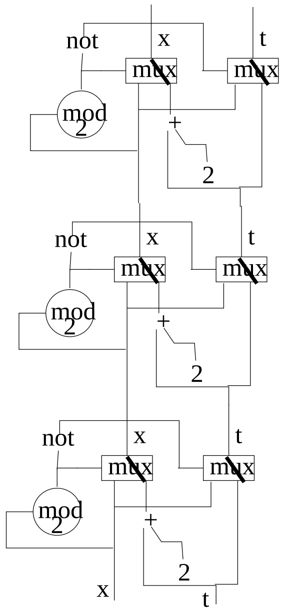
False memory dependencies - of load/store instructions which potentially can access the same memory location when in fact they are not. This is automatically exposed in $C^r(I)$ since gates are connected according only if their values are really used.

Eager evaluation - is the possibility of executing several execution paths in parallel as long as they do not affect the memory. Exposed by the parallel evaluation of independent subcomponents of $C^r(I)$.



Branch Prediction- speculatively the hardware can evaluate instructions of different iterations based on predictions which branches shall be taken. The probabilistic evaluation algorithm is designed so that it can use guesses to increase parallelism.

```
for( i=0; i < 3; i++)
   if(x mod 2 == 0) t = x;
   else  x = t + 2;
```



Pipeline parallelism- is not relevant when circuits are used as stages like "fetching" and "decoding" of assembler instructions are not required by $C^r(I)$.

Measuring the potential of ILP

Like P2NC, these results are usually upper bounds to the actual speedup of ILP since they can relay on run-time information which is not known to the hardware or the compiler (e.g., perfect knowledge regarding branch prediction).

- Earliest studies [**?**] in 1970 showed speedup factor of $2-3$ however this results where made without using branch prediction techniques. A well known and deep study of 18 programs was made by Wall [**?**].
- Wall categorized hardware abilities into seven categories (ranging from "stupid" to "perfect") according to abilities such as: loop unrolling size, window size knowledge of branch predictions, and knowledge regarding false memory dependencies. His results show that speedup of 2 to 60 ranging from the stupid to the perfect model with average of $7-13$ was obtained.
- A different study (a bit earlier) [**?**] studied the effect of different combinations branch-prediction and eager evaluation assuming perfect knowledge of memory dependencies in the trace. The results where quite high for programs with high degree of parallelism (e.g., matrix multiplication), however the harmonic mean was 158 for the ideal model (called "oracle") and up to 40 for the less optimal models.
- Finally, a more recent study [**?**] studied the effect of infinite window size where independent instructions are selected from the whole trace and executed in parallel. A significant increase in the results (harmonic mean speedup of 163 compared to 63) was obtained by eliminating the false memory dependencies caused by changing the stack-pointer in various function calls.

Well do we have a conclusive result?

2 The DOL language

DOL is a restricted form of an imperative language, it is specially designed to produce an efficient form of algebraic circuits which is the reason for some of its anomalies. The fact that DOL is general enough to simulate Trying machine immediately follows from the fact that Oblivious Turin machine (moving in the same direction until it hits the end of input) is general and hence its automaton can be simulated in DOL.

1. No pointers, only arrays can be used, $A[A[i]]$ is not allowed.
2. No if-statements or function calls, boolean operators are used instead.

$$
\begin{array}{ll}
 & C[i] = Exp; \\
if(exp)\ A[i] = 5; & A[i] = (5 \wedge C[i]) \vee (\neg C[i] \wedge A[i]); \\
else\ A[i] = A[i] + 1; & A[i] = ((A[i] + 1) \wedge \neg C[i]) \vee (A[i] \wedge C[i]);
\end{array}
$$

3. Input/output through special arrays.
4. The ranges of the do-loops are constant expressions.
5. break statement are supported:

$$
\begin{array}{l|l}
i = 0; & \\
while\ (i \leq 100)\ \{ & i\ := \ < 0:0, 0:1, 0:0 >; \\
\quad if(A[i] < B[i]) & do\ i\ 100\ \{ \\
\qquad break; & i = |< 1:1, 0:0, x+y-i-1 : A[i] \leq B[i] > \\
\quad i = i + 1; & \} \\
\} &
\end{array}
$$

6. $+, -, *, /, max, min, ==, \neq, <, \leq, >, \geq, \%(mod), \vee, \wedge, \neg, mux$

Flattening nested do-loops:

$i = |\ < 0:0, 0:1, 0:0 >;$
$1:\ do\ i =\ 20\{$
$\quad A[i] = i;$
$\quad j = |\ < 0:0, 0:1, 0:0 >; \qquad \Longrightarrow$
$\quad 0:\ do\ j =\ 10\ \{$
$\quad\quad B[j] = A[i] + B[j-1];$
$\quad\quad F[1] = B[j] \leq A[i];$
$\quad\quad j = |\ < 1:1, 0:0, 10-j : F[1] >;$
$\quad \}$
$\quad A[i] = A[i] + B[i];$
$\quad i = |\ < 1:1, 0:0, a[3]-i : (A[i] < A[i+1]) >;$
$\}$
$ii = |\ < 0:0, 0:1, 0:0 >;$
$i = |\ < 0:0, 0:1, 0:0 >;$
$1:\ do\ ii =\ 20 * 10\ \{$
$\quad C[0] =\ (ii\ \%\ 10\ ==\ 0);$
$\quad A[i] =\ (A[i] \wedge \neg C[0]) \vee (i \wedge C[0]);$
$\quad j = |\ < 0:0, 0:(ii\ \%\ 10\ ==\ 0), 0:0 >;$
$\quad B[j] = A[i] + B[j-1];$
$\quad F[1] = B[j] \leq A[i];$
$\quad j = |\ < 1:1, 0:0, 10-j : F[1] >;$
$\quad A[i] =\ (A[i] \wedge \neg C[0]) \vee ((A[i] + B[i]) \wedge C[0]);$
$\quad i = |\ < 1:(ii\ \%\ 10\ ==\ 0),$
$\quad\quad\quad 0:0, 20-i : (A[i] < A[i+1]) >;$
$\quad ii = |\ < 1:1, 0:0, 10-j : F[1] >;$
$\quad ii = |\ < 0:0, 0:0, (20-i)*10 : (A[i] < A[i+1]) >;$
$\}$

Flattening two consecutive do-loops:

$$
\begin{array}{l}
i = |\ \ < 0:0, 0:1, 0:0 >; \\
1:\ \ do\ i =\ 20\{ \\
\quad A[i] = A[i-1] - 1; \\
\quad F[1] = A[i] < B[i]; \\
\quad i = |\ \ < 1:1, 0:0, 20-i : F[1] >; \quad \Longrightarrow \\
\} \\
A[k] = B[k] + A[k]; \\
j = |\ \ < 0:0, 0:1, 0:0 >; \\
2:\ \ do\ j =\ 10\ \{ \\
\quad B[j+k] = A[i-1] + A[j+1]; \\
\quad j = |\ \ < 1:1, 0:0, 10-j : (B[i] \neq 0) >; \\
\} \\
i = |\ \ < 0:0, 0:1, 0:0 >; \\
j = |\ \ < 0:0, 0:1, 0:0 >; \\
ii = |\ \ < 0:0, 0:1, 0:0 >; \\
1:\ \ do\ ii =\ 30\{ \\
\quad C[1] = (ii\ \ \leq\ 20); \\
\quad A[i] = (A[i] \wedge \neg C[1]) \vee ((A[i-1] - 1) \wedge C[1]); \\
\quad F[1] = (F[1] \wedge \neg C[1]) \vee ((A[i] < B[i]) \wedge C[1]); \\
\quad A[k] = (A[k] \wedge (ii \neq 20)) \vee ((B[k] + A[k]) \wedge (ii == 20)); \\
\quad B[j+k] = (B[j+k] \wedge C[1]) \vee \\
\qquad\qquad ((A[i-1] + A[j+1]) \wedge \neg C[1]); \\
\quad i = |\ \ < 1:(i \leq 20), 0:0, 20-i : F[1] \wedge (i \leq 20) >; \\
\quad j = |\ \ < 1:(ii > 20), 0:0, \\
\qquad\qquad 10-j : ((B[i] \neq 0) \wedge (ii > 20)) >; \\
\quad ii = |\ \ < 1:1, 0:0, 20-i : F[1] \wedge C[1] >; \\
\quad ii = |\ \ < 0:0, 0:0, 10-j : \\
\qquad\qquad ((B[i] \neq 0) \wedge \neg C[1]) >; \\
\}
\end{array}
$$

Note that all the updates of index variables are (except the skip) depends on constants and on the loop-index of the flattened program. In addition, the skip affects the index ii of the flattened loop, thus, skipping iterations in one of the original loops is correctly translated to a skip on the outer loop index (ii).

Flow chart of the P2NC project

1. Read DOL program and flat it to the AC tree representation of one iteration.
2. $T = 1$, read P, generate initial array file $Afile$ with zero values.
3. Call $get\_chunk(T, P, Afile)$, to build the algebraic circuit for this chunk as described in figure 3 including substitution/folding of constant array/index values (this fills the input values of the chunk's circuit). Note that gates are tagged with output array locations for storing output values.
4. The resulted AC circuit is transformed to a boolean circuit.
5. Evaluate the boolean circuit using the AND/OR restrictions.
6. Identify the most early "break" node in the circuit (a node tagged with break) which has been evaluated to true and set the start value of T for the next chunk accordingly (default is $T = t + P$).
7. Substitute tagged output values from the circuit to the suitable array locations provided that the iteration index of these nodes is less equal than the iteration index of the earliest break-node of the previous round.
8. Go to stage three until T has reached to its maximal value.

```
i=0;
ii=0;
do ii 10000000  {
   A[i] = (A[i+1]+A[i-1])/2;
   i =| <1:1, 0:(ii % 10 == 0),0:0>;
   ii=| <1:1, 0:0,  1000:(A[ii%100] > IN[i])>;
}
```

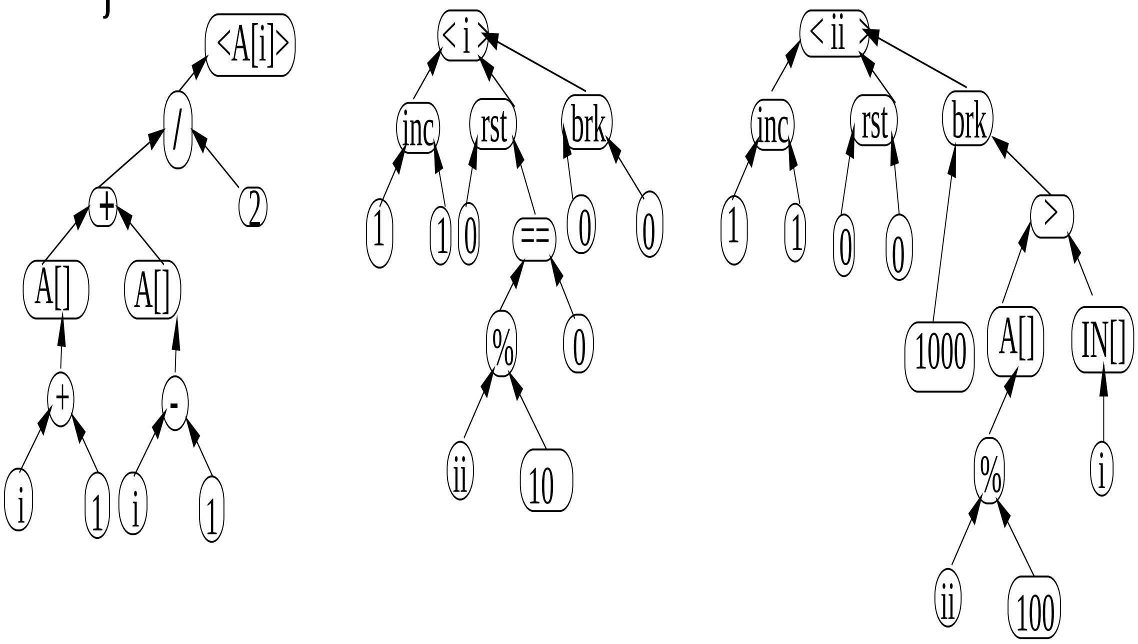


Figure 2: *Building the initial ac-circuit out of a flattened loop.*

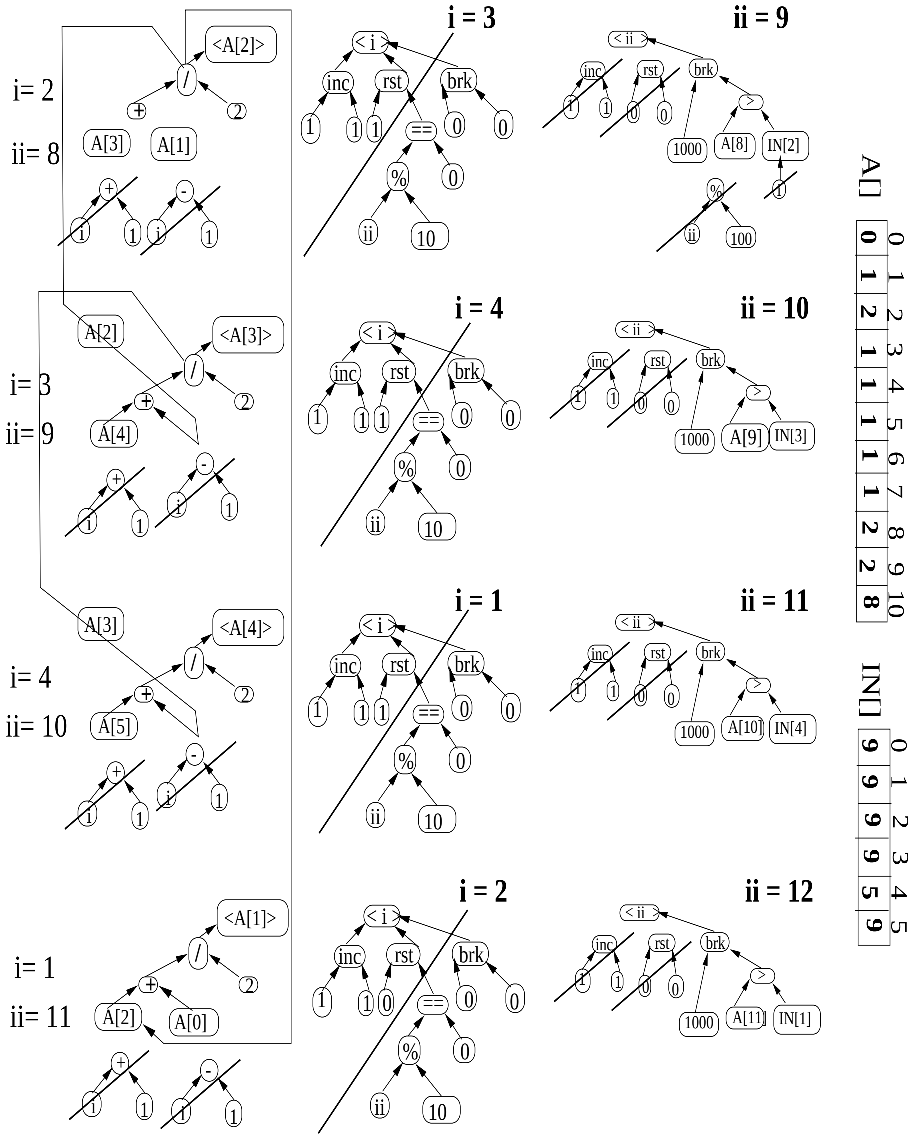


Figure 3: *Creating the circuit for the whole chunk.*

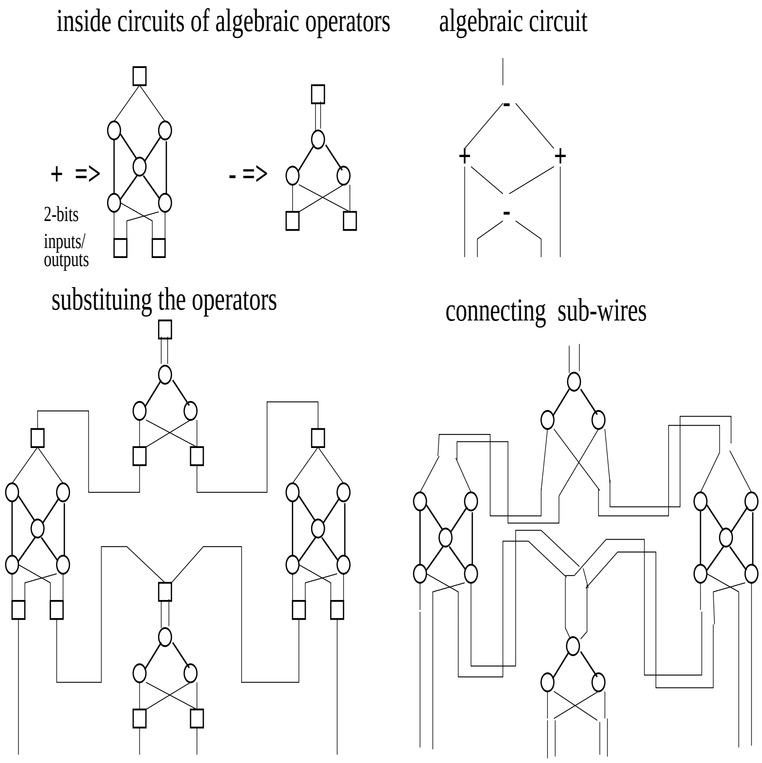


Figure 4: *substituting algebraic circuits.*

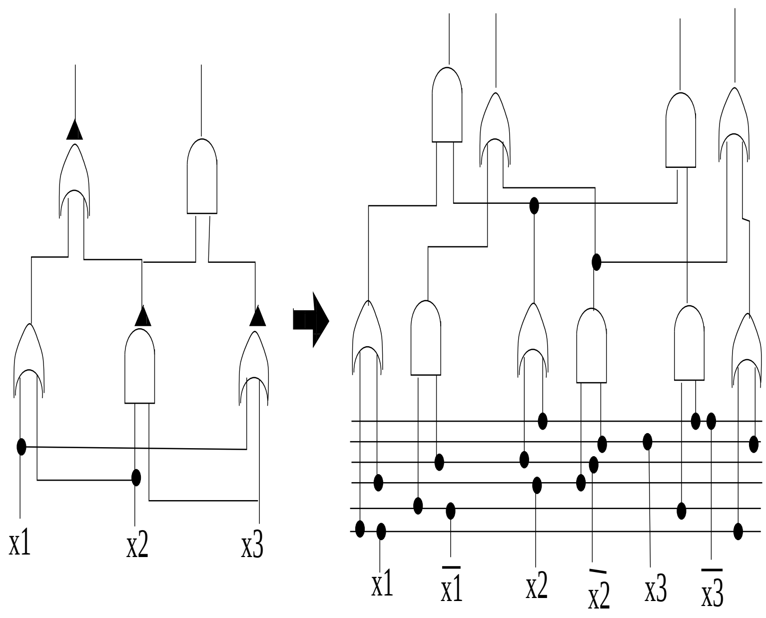


Figure 5: *Pushing the NOT-gates to the inputs.*

The and/or restriction algorithm

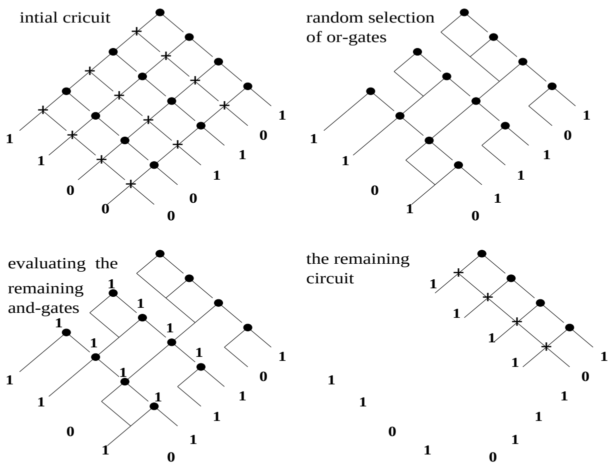


Figure 6: *Or restriction of a mesh like circuits $+/\cdot$ are OR/AND gates.*

1. In random, connect one of the inputs of every OR-gates to its output (thus temporarily removing the OR-gates from the circuit).
2. Mark by $'1'$ the output of every AND-gate whose “leaves” are all marked $'1'$. This can be efficiently done using simple parallel connectivity like algorithm.
3. Restore the last deleted or bypassed or gate.
4. Evaluate and mark each gate whose inputs values are marked.
5. If there are un-marked gates then goto the first step.

3 Compilation track (I64)

In here we attempt to create optimized code for executing on regular existing processors with super-scalar and pipeline units.

We observe that it is possible to create a generic code for one chunk of size p if all the indexing functions are linear, e.g., $(i+j)*2-5$ is linear since it can be transformed to $2*i+2*j-5$ so that the base value of every chunk can be added to the compiled chunk. For example

$$\begin{array}{ll} for\ i = T \ldots\ T + P\ do & for\ i = 1 \ldots\ P\ do \\ \quad A[i] = A[i-1] + A[i-1]; & \quad A[T+i] = A[T+i-1] + A[T+i-1] \end{array}$$

Thus, generic code for one chunk

$$for\ j = 1 \ldots PdoA[T+i] = A[T+i-1] + A[T+i-1]$$

can be created and augmented in

$$\begin{array}{l} for\ k = 0 \ldots \frac{n}{P}\ do \\ \quad T =\ k \cdot \frac{n}{P}; \\ \quad for\ j = 1 \ldots PdoA[T+i] = A[T+i-1] + A[T+i-1]; \} \end{array}$$

1. Set P to be the pipeline depth times the number of super-scalar units.

2. Read DOL program and flat it to the AC tree representation of one iteration as described in figure 2.

3. Generate a compile code of the flattened DOL for P iterations use the base T to change linear indexing functions so that they will depend on T alone (+ the explicit index in the chunk which is known at this stage).

4. Perform topological sort and reordering of the chunk's statements so that independent statements are grouped together.

5. Use the AC representation of the chunk's code to perform register allocation.

6. Augment the chunk's code in a suitable loop.

7. Generate Merced code and execute it.

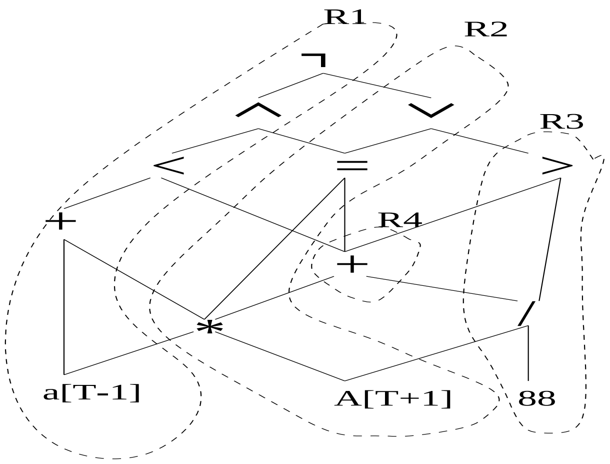


Figure 7: *Register allocation in a given AC.*