The Power Compiler.

Accelerating C code using Multi-Cores for lower power and higher throughput

```
%./runPowerCompiler.sh  --prefix=. --resource=Resources.xml cic.cc
--output=mpi
1. ---> Generate LLVM IR
2. ---> Prepare POLLY
3. ---> Generate parallel loops
4. ---> Analyse parallel loops and modify IR correspondingly
5. ---> Unroll parallel loops and modify IR correspondingly
5. ---> Next to parallel loop : polly.loop_after
6. ---> Optimize IR : BalanceTree, ReduceBits
7. ---> Run scheduler
8. ---> Allocate resources
8. ---> Get BasicBlock frequencies
9. ---> Bind resources
10. ---> Generate Language Network
11. ---> Generate MPI
```

Who we are

- RUSHC : 20 years of development.
- Armenian team + US team (9 total)
- Andy Fox. Many projects:
  - Hw : Microsemi, Malleable, Basis, USL (Verilog, Serdes, Graphics)
  - Sw: Tabula, Actel, Carbon, Ausdia (llvm, hdl synthesis, sat-sim).

How we work

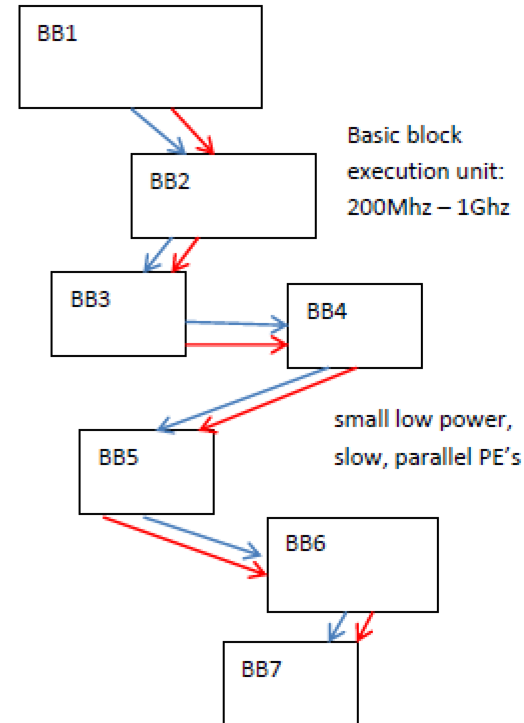
- Core technologies developed (HDL synthesis, verification, schedulers, llvm based compiler flows).
- License core technology and customize for client.
- Client owns customized source

What Power Compiler Does:

- Scheduling (SDC based scheduler) based on real instruction utilization (running real data)
- Matching (matching instruction sequences onto library).
- Balancing;Type strengthening
- Sw pipeline generation: 1 program -> multiple PE's.
- LLVM -> Optimization -> LLVM -> Network
  - Optimization written as "opt passes" (llvm in, llvm out).
- Generation of verific data structures.
- Generation of MPI/parallel C.

Power Reduction By Parallelization

```
void cic()
{
    //comb
    for(int i = 0; i < n; ++i){
        comb[i] = x[i] - delay_buffer[N/I - 1];
        for (int l = N/I - 2; l >= 0; --l) {
            delay_buffer[l + 1] =
delay_buffer[l];
        }
        delay_buffer[0] = x[i];
    }
    // zeros inserting
    int j = 0;
    for(int i = 0; i < n + n; i = i + 2) {
        y[i] = comb[j];
        y[i + 1] = 0;
        ++j;
    }
    //integrator
    for (int j = 0; j < 2 * n - 1; ++j) {
        y[j + 1] = y[j + 1] + y[j];
    }
}
```



Serial code on high speed processor

Massive Power Reduction

Massive increase in throughput

CPU: 1 GHz,

2.5W

Basic Block processing elements:
80mw.

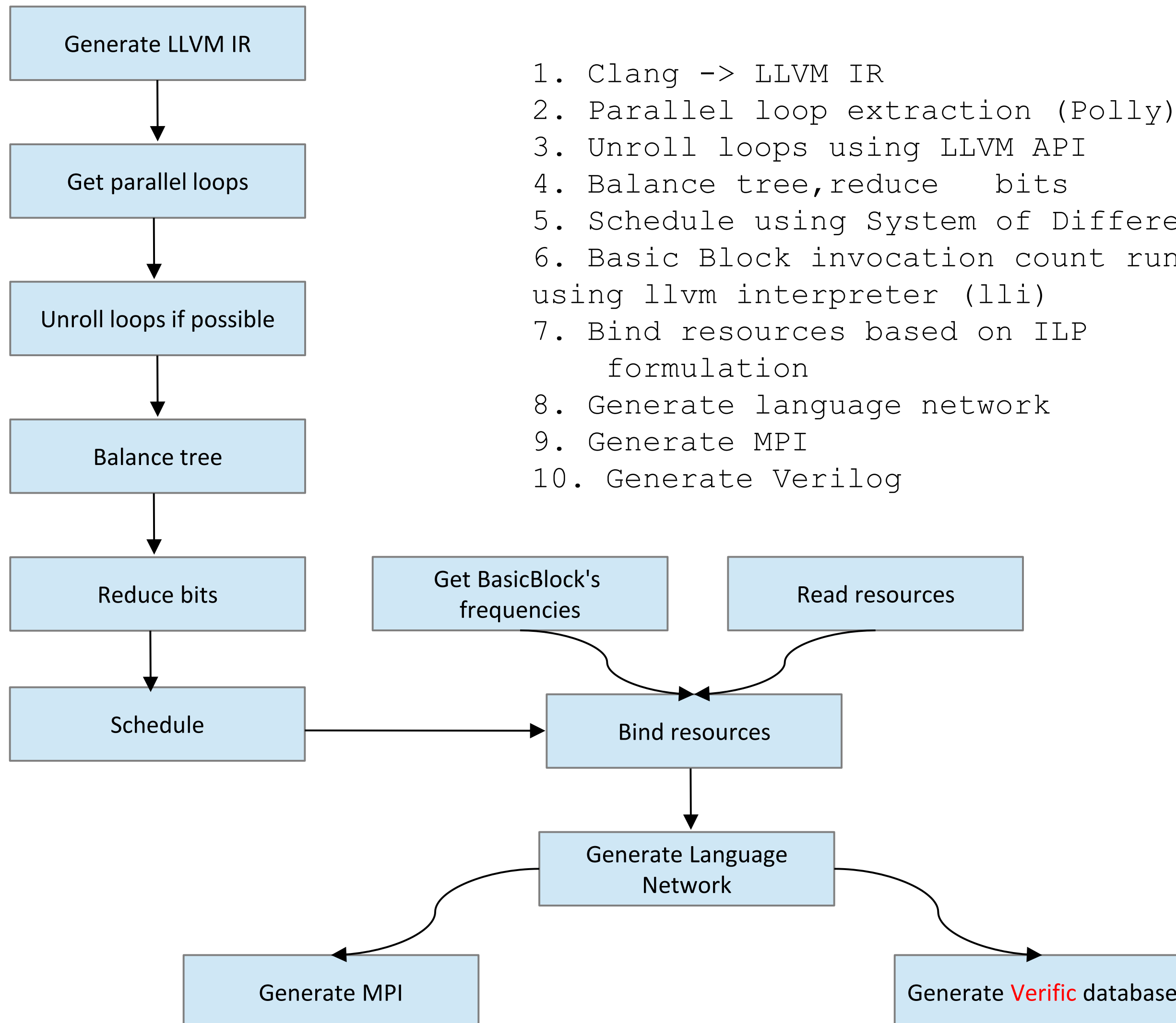
How Much Parallelism ? (cic)

Basic Block	Invocations	Number of instructions	Latency	Average number of instructions in parallel per cycle
BB1	10	21	7	3
BB2	1	6	5	1.2
BB3	20	14	5	2.8
BB4	1	6	5	1.2
BB5	39	9	6	1.5
BB6	1	2	1	2
BB7	1	1	1	1

Power Reduction (cic)

Basic Block	Basic Block frequency	Total power accelerator	Total power CPU	Average number of instructions in parallel per cycle
BB1	500	0.21	0.525	3
BB2	400	0.015	0.015	1.2
BB3	500	0.3	0.7	2.8
BB4	400	0.015	0.015	1.2
BB5	500	0.702	0.8775	1.5
BB6	200	0.003	0.005	2
BB7	200	0.003	0.0025	1

Flow



1. Clang -> LLVM IR
2. Parallel loop extraction (Polly)
3. Unroll loops using LLVM API
4. Balance tree,reduce bits
5. Schedule using System of Difference Constraints (SDC)
6. Basic Block invocation count running real data and using llvm interpreter (lli)
7. Bind resources based on ILP formulation
8. Generate language network
9. Generate MPI
10. Generate Verilog

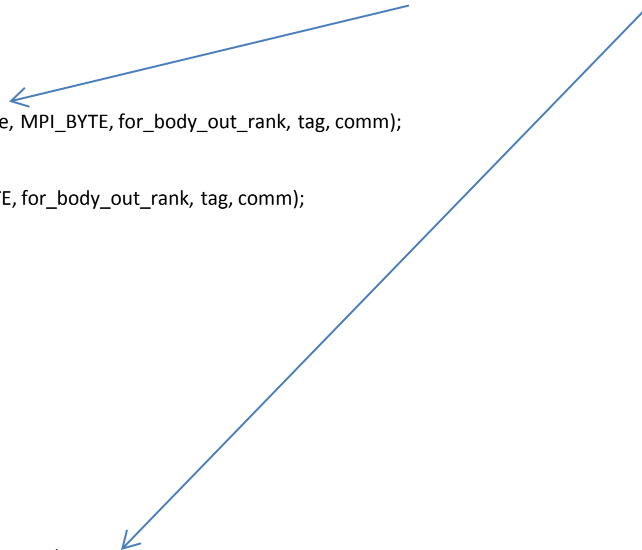
cic.cc Test case

```
void cic()
{
    //comb
    for(int i = 0; i < n; ++i){
        comb[i] = x[i] - delay_buffer[N/I - 1];
        for (int l = N/I - 2; l >= 0; --l) {
            delay_buffer[l + 1] = delay_buffer[l];
        }
        delay_buffer[0] = x[i];
    }

    // zeros inserting Parallel Loop Identified by Polly
    int j = 0;
    for(int i = 0; i < n + n; i = i + 2) {
        y[i] = comb[j];
        y[i + 1] = 0;
        ++j;
    }

    //integrator
    for (int j = 0; j < 2 * n - 1; ++j) {
        y[j + 1] = y[j + 1] + y[j];
    }
}
```

MPI output

```
int main(int argc, char* argv[]) {
   int numtasks, rank, rc, len;
   char hostname[MPI_MAX_PROCESSOR_NAME];
   rc = MPI_Init(&argc, &argv);
   if (rc != MPI_SUCCESS) {
      printf ("Error starting MPI program. Terminating.\n");
      MPI_Abort(MPI_COMM_WORLD, rc);
   }
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(hostname, &len);
    int y[40] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
   int comb[40] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
   int x[20] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
   int delay_buffer[5] = {0, 0, 0, 0, 0};
   typedef struct {
      int m_y[40];
   ..];
   } RAM_Struct;
   RAM_Struct RAM_obj;
   if (rank == 0) {
      // BBPE entry
      bool entry = 1;
        typedef struct {
         int m__pre67;
      } for_body_out;
      for_body_out for_body_out_obj;
   ..
      MPI_Send(&for_body_out_obj, for_body_out_size, MPI_BYTE, for_body_out_rank, tag, comm);
      for (size_t i = 0; i < 20; ++i) {
         RAM_obj.m_x[i] = x[i];
      MPI_Send(&RAM_obj, sizeof(RAM_obj), MPI_BYTE, for_body_out_rank, tag, comm);
   }
   else if (rank == 1) {
      MPI_Comm comm = MPI_COMM_WORLD;
      ...;
      bool is_first_time_ = true;
      MPI_Request req[2];
      do {
         int idx = -1;
         int RAM_rank = 0;
         int _pre68 = 0;
         for_body = 0;
         int tmp5 = 0;
         long indvars_iv_next62 = 0;
         if (! is_first_time_) MPI_Waitany(2, req, &idx, &status);
         MPI_Irecv(&entry_in_obj, entry_in_size, MPI_BYTE, entry_in_rank, tag, comm, &req[0]);
         MPI_Irecv(&for_body_in_obj, for_body_in_size, MPI_BYTE, for_body_in_rank, tag, comm, &req[1]);
```

Communication between Processes via MPI_Send/MPI_Recv



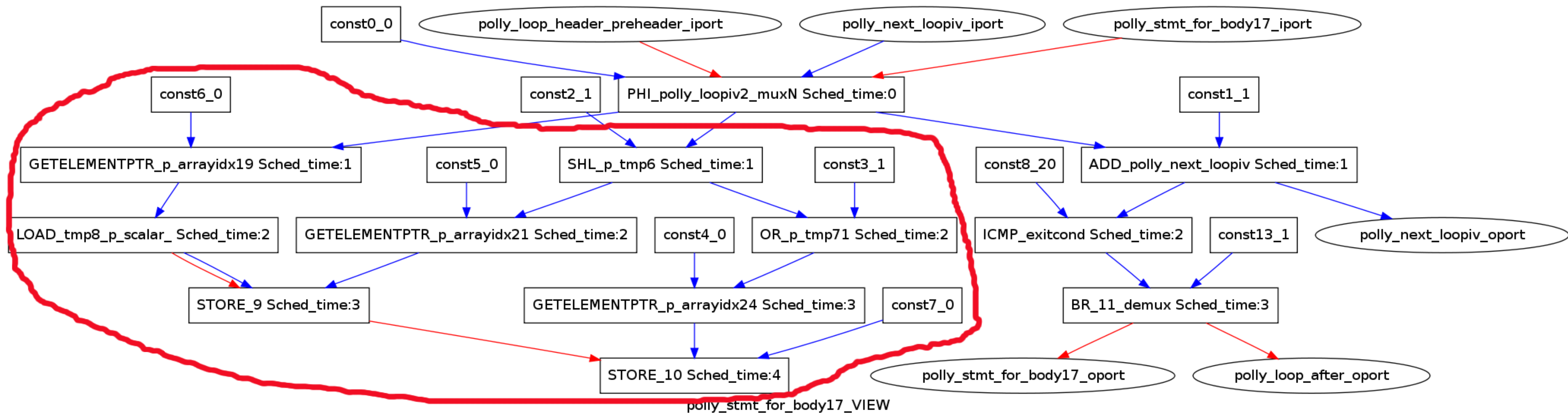
Verilog Output

FSM generated. Each instruction scheduled on a cycle, sharing, memory accesses..

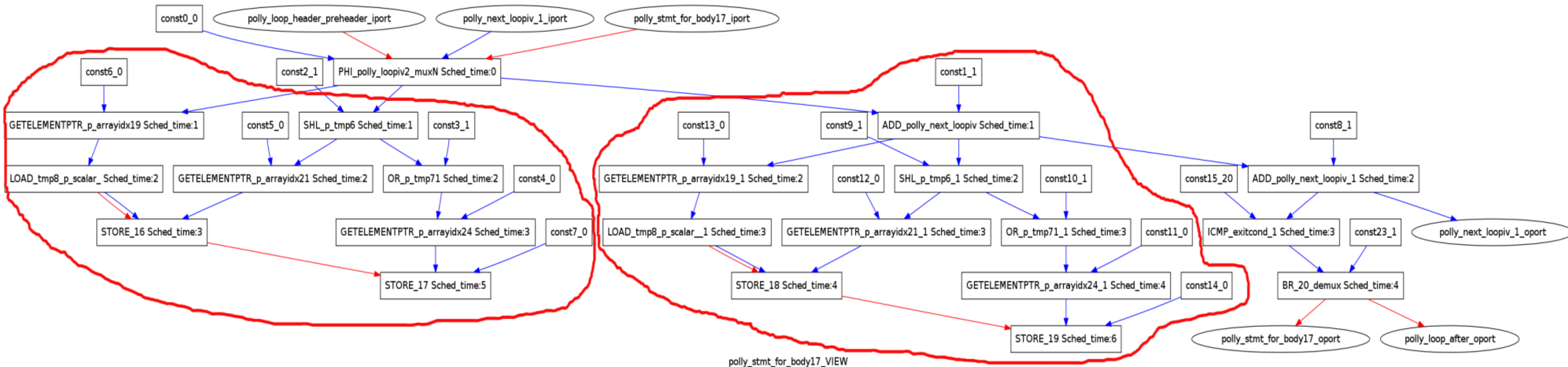
```
if(bbpe_state_reg_q == BBPE_RUNNING)
                begin
                    case (bbpe_cycle_count_q)
                    3'd0:
                        begin
                            begin: execution_unit_0_0
                                PHI_indvars_iv61_muxN_0_reg <= 64'd0 *
                                  in_entry + in_indvars_iv_next62 *
                                  in_for_body;
                                GETELEMENTPTR_arrayidx_0_reg <= 64'd0;
                                bbpe_mem_req_LOAD_tmp5_q <= 1'b1;
                                bbpe_mem_rw_LOAD_tmp5_q <= 1'b0;
                                bbpe_mem_addr_LOAD_tmp5_q <=
                                  GETELEMENTPTR_arrayidx_0_reg;
                            end //execution_unit_0_0
                        end //cycle 0
                    3'd1:
                        begin
                            begin: execution_unit_1_0
                                PHI_tmp4_muxN_0_reg <= in__pre *
                                  in_entry + in_tmp3 * in_for_body;
                                GETELEMENTPTR_arrayidx2_0_reg <= 64'd0;
```

Parallel Loop Extraction

Parallel loop



Unrolled parallel loop with factor 2 from CIC test

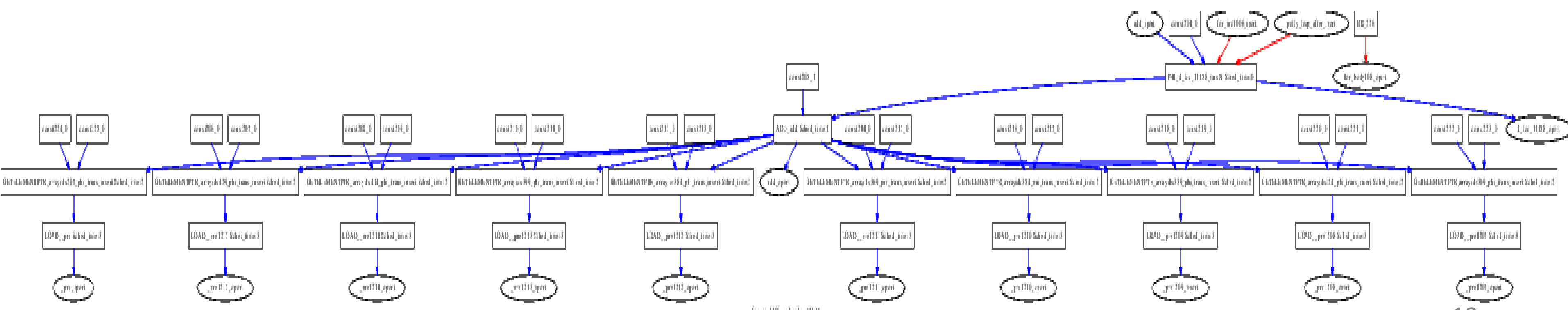


Instruction Parallelism from Scheduling

Sequential instructions from Smith-Waterman test

```
for.cond106.preheader:
  %d_loc.11158 = phi i64 [ 0, %polly.loop_after ], [ %add, %for.inc1006 ]
  %add = add i64 %d_loc.11158, 1
  %arrayidx429.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg9, i64 0, i64 %add, i64 0
  %arrayidx414.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg8, i64 0, i64 %add, i64 0
  %arrayidx399.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg7, i64 0, i64 %add, i64 0
  %arrayidx384.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg6, i64 0, i64 %add, i64 0
  %arrayidx369.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg5, i64 0, i64 %add, i64 0
  %arrayidx354.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg4, i64 0, i64 %add, i64 0
  %arrayidx339.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg3, i64 0, i64 %add, i64 0
  %arrayidx324.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg2, i64 0, i64 %add, i64 0
  %arrayidx309.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg1, i64 0, i64 %add, i64 0
  %arrayidx297.phi.trans.insert = getelementptr [78 x [78 x i64]]* %Hg0, i64 0, i64 %add, i64 0
  %.pre = load i64* %arrayidx297.phi.trans.insert, align 16, !tbaa !0
  %.pre1207 = load i64* %arrayidx309.phi.trans.insert, align 16, !tbaa !0
  %.pre1208 = load i64* %arrayidx324.phi.trans.insert, align 16, !tbaa !0
  %.pre1209 = load i64* %arrayidx339.phi.trans.insert, align 16, !tbaa !0
  %.pre1210 = load i64* %arrayidx354.phi.trans.insert, align 16, !tbaa !0
  %.pre1211 = load i64* %arrayidx369.phi.trans.insert, align 16, !tbaa !0
  %.pre1212 = load i64* %arrayidx384.phi.trans.insert, align 16, !tbaa !0
  %.pre1213 = load i64* %arrayidx399.phi.trans.insert, align 16, !tbaa !0
  %.pre1214 = load i64* %arrayidx414.phi.trans.insert, align 16, !tbaa !0
  %.pre1215 = load i64* %arrayidx429.phi.trans.insert, align 16, !tbaa !0
  br label %for.body108
```

Parallel instructions from Smith-Waterman test



New matching technology

- STP/SMT matching (“Functional match”).
- New approach
  - Match code fragments over sequences.
  - Matches functions (uses formal verification).
- Value Proposition:
  - Match dsp kernels onto DSP fabric
  - Examples: FFT, FIR, loops

Matching: llvm <-> Library

- <cell>
- <name>Butterfly</name>
- <inputs>
- <input>
- <name>a</name>
- <type>i32</type>
- </input>
- ...
- </inputs>
-
- <outputs>
- ...
- </outputs>
- <signature>
- op = a * b + c * d
- </signature>
- <llvm>
- %mul = mul nsw i32 %b, %a
- %mul1 = mul nsw i32 %d, %c
- %op = add nsw i32 %mul1, %mul
- </llvm>
- <cost>1</cost>
- </cell>

Matching Example

- Real design part before and after covering

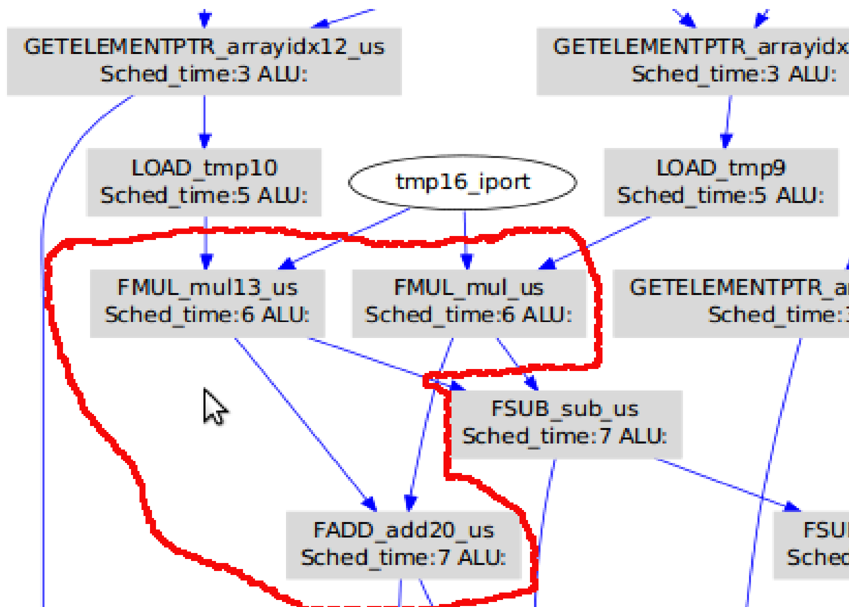


Figure 1.
The selected part is "Butterfly" and should be matched

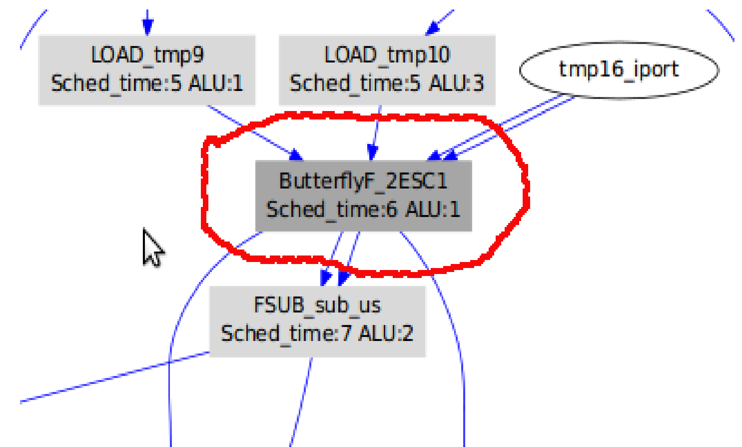


Figure 2.
The "Butterfly" is matched and covered by The Library element called "Batterfly_2ESC"

Big Kernels we want to match:

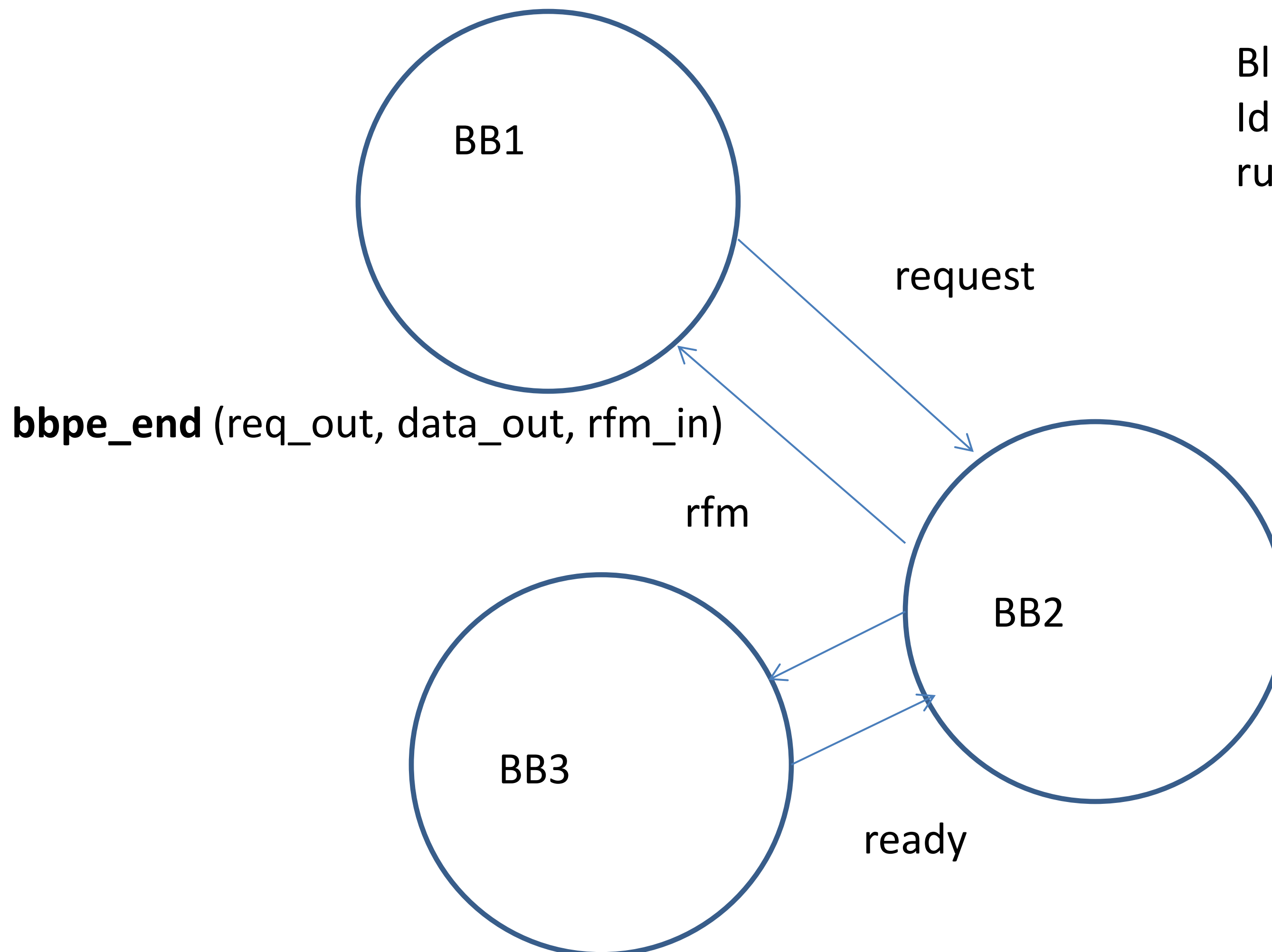
ShiftAmount(5) = 5’d16-levelSuffixSize(4);
level_suffix(12) = BitStreamBufferOutput(16) >>
ShiftAmount[4:0](5)
{levelCode2B1(8),levelCode2B0(8)} = levelCode3(8) +
level_suffix(12) ß 13 bit addition result.
But only bits 9:0 used in result. So we can restrict level_suffix(12)
{levelCodeB1,levelCodeB0}=(level_prefix(4) << suffixLength(4)) +
{levelCode2B1(8),levelCode2B0(8)} ← 16 bit adder
levelCodeMsb(8)={levelCodeB1(8),levelCodeB0(8)}>>1
levelCodeLsb(8) = levelCodeB0 & 1’b1;

which can be rewritten as:
levelCode_tmp_0(8) = level_prefix(4) << suffixLength(4);
levelCode_tmp_1(8) = levelCode3(8) + level_suffix(12);
{levelCodeB1(8),levelCodeB0(8)} = levelCode_tmp_0(8) +
levelCode_tmp_1(8);
levelCodeMsb(8)={levelCodeB1(8),levelCodeB0(8)}>>1
levelCodeLsb(8) = levelCodeB0 & 1’b1;

Basic Block Execution:Execution model for Data Flow

bbpe_start (req_in, data_in, rfm_out)

bbpe_end (req_out, data_out, rfm_in)



BBPE states:

Blocked
Idle
running

BB2 blocked when:
BB3 cannot take data

Cannot access shared resource (eg memory).

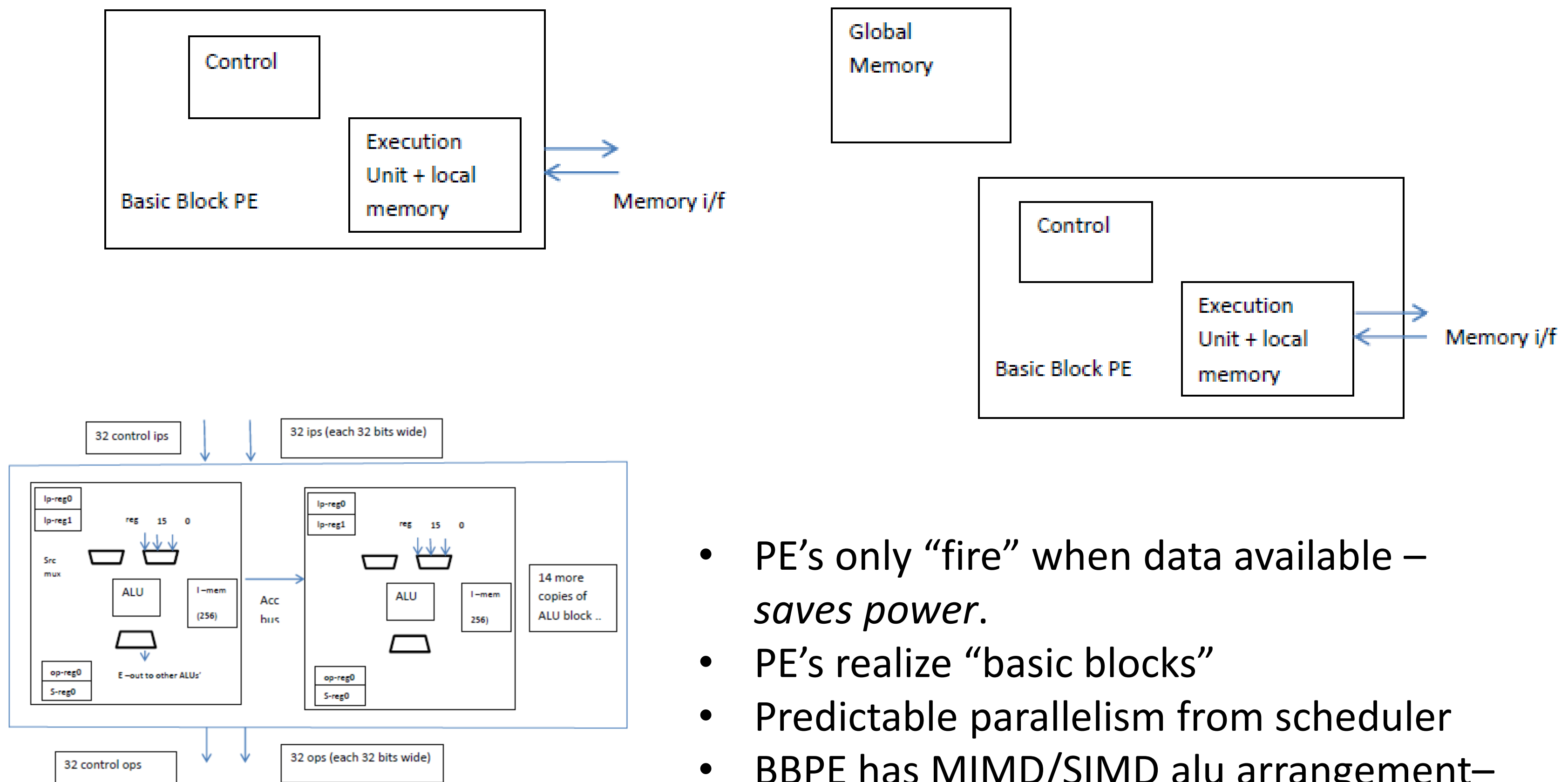
BB2 “executes” when request line asserted and it is not blocked

Verilog

See file: bbpectl.v

- Allows for bbpe to request memory (shared resource) and stall until resource available.
- Keeps track of current instruction in basic block
- Allow for multiple requests/multiple data.

Execution Model: Data Flow network of PE's



- PE's only "fire" when data available – *saves power*.
- PE's realize "basic blocks"
- Predictable parallelism from scheduler
- BBPE has MIMD/SIMD alu arrangement– *parallelism saves power - low frequency*.

Summary Results

Design	CPU Power	Accelerator Power	Savings
DCT	36.8	7.6	4.8x
CIC	2.14	1.2	1.7x
Smith Waterman	5773	1530	3.7x

Data: CIC

Basic Block	Basic Block frequency	Total power accelerator	Total power CPU	Average number of instructions in parallel per cycle
BB1	500	0.21	0.525	3
BB2	400	0.015	0.015	1.2
BB3	500	0.3	0.7	2.8
BB4	400	0.015	0.015	1.2
BB5	500	0.702	0.8775	1.5
BB6	200	0.003	0.005	2
BB7	200	0.003	0.0025	1

Data: DCT

Basic Block	Basic Block frequency	Total power accelerator	Total power CPU	Average number of instructions in parallel per cycle
BB1	500	0.21	0.525	3
BB2	400	0.015	0.015	1.2
BB3	500	0.3	0.7	2.8
BB4	400	0.015	0.015	1.2
BB5	500	0.702	0.8775	1.5
BB6	200	0.003	0.005	2
BB7	200	0.003	0.0025	1

Data: Smith Waterman

Basic Block	Basic Block frequency	Total power accelerator (mw)	Total power CPU (mw)	Average number of instructions in parallel per cycle
BB1	500	4.674	8.075	2.07
BB2	500	0.9	4.3125	5.75
BB3	300	0.012	0.1025	10.25
BB4	200	0.003	0.0525	21 (Alloca)
BB5	1000	375	0.375	1
BB6	200	0.003	0.0025	1
BB7	500	1524.6	5760.5625	4.53

Summary

- Massive power reduction
- Dusty Deck “C” to MPI/Verilog/Verific operator netlist.
- Code available now
- Test cases run:
  - Viterbi, Turbo, FIR, FFT, DCT,..

Core Technologies

- Llvm -> Language network
- Scheduler
- Matcher
- Packer