Packet Architects AB

Compiler Optimizations applied to deeply pipelined packet processing

Abstract

The packet processing in routers and switches is complex and requires very high performance. The implementation even in leading edge ASIC or FPGA technologies requires deep pipelining. This is very time consuming to develop and to adapt to new requirements which is quite frequent in the world of networking.

Packet Architects have developed a new high level language, PAC, and a synthesis tool chain that produces very efficient pipelined RTL code for hardware implementation in FPGAs or ASICs. With this tool we have developed fully featured packet processing for routers. By changing parameters to the optimization algorithms we can create very efficient hardware whether the customer wants processing speeds at Gbit/s or Tbit/s speed.

The goal of this project is to take inspiration from the breadth of compiler optimizations and apply this to our tool flow and target technology. Adapting algorithms for algebraic factorization and common subexpression elimination could be a starting point.

The input is a data flow graph using the Graphviz dot file format and the output should be an optimized data flow graph in the same format. To evaluate the result of the optimization the data flow graph is run through our pipeline synthesis tool which provides area and timing results.

The target architecture is different than a processor in that it allows unlimited parallelism and pipelining. All inputs to the program are simultaneously available in one clock cycle. The computed result must be available simultaneously in one clock cycle. The latency from input to output must be constant.

Candidate Algorithms

CSE - common subexpression elimination

x = a*b+c
y = d*a*b
→
tmp = a*b

x=tmp+c
y=d*tmp

The obvious advantage of this optimization is that an operation is removed which can reduce the hardware cost. Since the operands are used in fewer operations it can potentially reduce the cost of pipelining the operands. However there is also a potential cost due to that the result of the operation now must be transferred to multiple receiving operators which can have a pipeline cost.
In general the total bit width of operands are greater than the width of the result (for most operations) therefore the expected pipeline cost would be reduced.

Subtree replication

Reverse of CSE to introduce more parallelism.
c = a*b
x=d+c
y=e*c
→
x = a*b+c
y = e*a*b

There are situations where the reuse of a result can result in a slower or larger pipeline. Replicating subexpression allows more parallelism and decouples the operators allowing freer placement in the pipeline that can reduce the pipeline cost.

Arithmetic tree height reduction

x = a+b+c+d
x = a+(b+(c+d)) maximum height tree
x = (a+b)+(c+d) balanced tree

The balanced tree is the preferred structure in many cases because it reduces the depth and depth has a high cost in the form of pipelining of the operands and results.

A parameter should control flatness from fully balanced to maximum height.

Algebraic simplifications

a+a → 2*a → a << 1

x = a*b + a*c → x = a*(b+c) *same tradeoff as for CSE vs replication*
-1-x → -(1+x)

Some of these simplifications requires a model for the hardware cost in order to make a good choice.

Operator optimization

Some operators can be implemented much more efficiently, e.g. shift instead of multiplication. This optimization will normally be done in the RTL synthesis but doing it in the compiler before pacopt will improve the quality of the pacopt optimizations.

x/2 → x>>1
y*2 → y<<1
a * const → a + a << n1 + a << n2 …
x>>1 → x<hi:1>
y<<1 → { y , 0 }

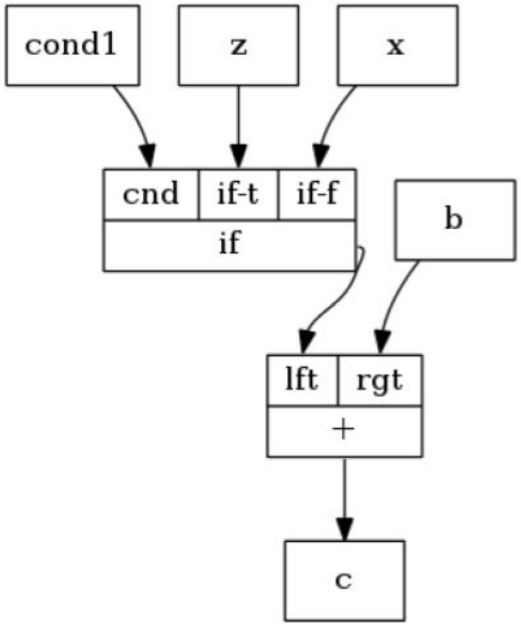
Data width/range analysis

The code might have unused bits due to being generic code that doesn't have optimal calculated bit widths. It is possible to calculate needed bit widths from constants and port sizes for all operands in the complete program.

Dead code elimination / constant propagation

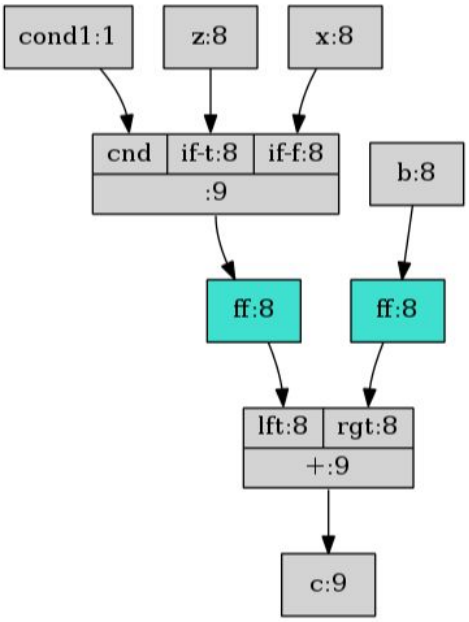
Current compiler does very limited optimizations so there can be constant expressions that could be evaluated at compile time.

Data Flow Graph (DFG)



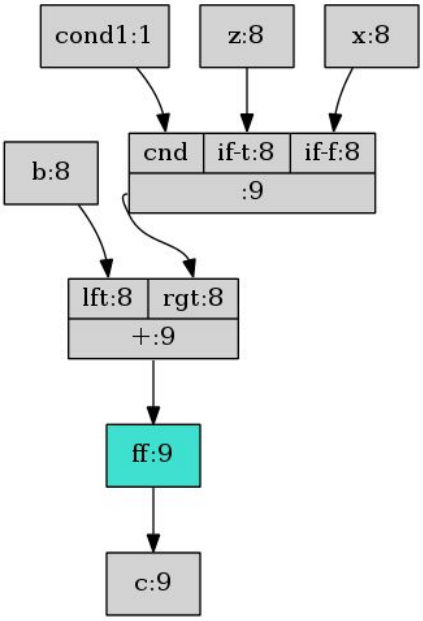
- Directed acyclic graph.
- Nodes represent operations. Similar to assembler instructions.
- Edges represent the data flow.
- No "variables".

DFG after scheduling - 1



- Two pipeline stages.
- Conditional/mux before FFs.
- Add operation after FFs.

DFG after scheduling - 2



- Both conditional and add operation in the first pipeline stage.
- Number of FFs is now lower.
- Delay in pipeline stage 1 is now longer.
- The slowest pipeline stage determines the pipeline frequency.
- If stage 1 is not the slowest stage then this schedule is always better.
- If stage 1 is now the slowest stage and it makes the pipeline frequency too slow, this is not a valid schedule.

Hardware Constraints

Constraints needed for transforming the DFG into correct hardware

- Each clock cycle a new calculation is started. All inputs arrive simultaneously in the same cycle.
- Each clock cycle the pipeline produces a result on all outputs simultaneously in the same cycle (unless the pipeline is empty).
- All DFG operators are combinational with the exception of:
  - **ramrd/ramwr** These operators have a fixed pipeline latency (typically 1-3 cycles).
  - **regrd** This can be regarded as a constant that can be placed freely in the pipeline. In the hardware this is registers that issetup by software before the pipeline processing is started.
  - **regwr** Similar to regwr but output from the pipeline. Can be regarded as a combination output that can be placed freely in the pipeline.
  - **blkrd** Input from other hardware units, freely placed and does not have any visible latency.

- **blkwr** Output to other hardware units, freely placed and does not have any visible latency
- **ff** The original DFG does not contain any pipeline flipflops. These are added by the *pacopt* program.

DFG - Operator Description

This is a description of the operations/vertexes in the data flow graph. This describes a newer version of the operators that is similar but not identical to the version you will be working with.

-- bitconcat --
Concatenates the inputs in0..n, in0 being lsb/msb? Size of each
input must be taken from each input edge. Size of output is sum of input
widths. No limit on number of inputs, except >= 2.

bitconcat_101 [label="{{<in0> in0 | <in1> in1 | <in2> in2} | <out> bitconcat_101}"];

-- bitextc --
Extracts bits from a bitstring at constant index. Indexes are specified in the name of the out port in the form
<hi:lo>. With hi being the msb. Indexes must be numeric constants in base 10. Single bits are extracted by having index hi==lo.

bitext_0_0_120 [label="\<0:0\>"];
bitext_111_0_151 [label="\<111:0\>"];
==>
bitextc_151 [label="{{idx=111:0 | <in> in} | <out> bitextc_151}"]

-- bitext --
Extracts bits from a bitstring with a variable index. Number of bits extracted
is specified in the labels width parameter. Note that the width is not a port
and no edge can be connected to that. Width must be a base 10 numeric constant.

bitext_5 [label="{{width=1 | <idx> idx | <in> in} | <out> bitext_5}"];

-- lshift --
Perform left shift, <<. Size of operands from edges. Shift in 0.
lshift_12 [label="{{<in> in | <shift> shift} | <out> lshift_12}"]

-- rshift --
Perform right shift, >>. Size of operands from edges. Shift in 0.
rshift_12 [label="{{<in> in | <shift> shift} | <out> rshift_12}"]

-- if --
If the condition (must be a single bit) is true the output equals <ift> else <iff>.
Bit width of ift, iff and ifout must be equal.

if_164 [label="{{<cnd> true | <ift> ift | <iff> iff} | <out> if}"];
if_166 [label="{{<cnd> false | <ift> ift | <iff> iff} | <out> if}"];
==>
if_164 [label="{{<cnd> cnd | <ift> ift | <iff> iff} | <out> if_164}"];

-- setbit --
Will set one bit to the value of <fdata> determined by <idx> in <in> and leave the other bits unmodified.
Width of in/out and idx are determined by edges.

in<idx> = fdata; --> in'2 = $setbit(idx,fdata,in'1) -->
setbit_5 [label="{{<idx> idx | <fdata> fdata | <in> in} | <out> setbit_5}"];

-- setbits -- Will set a consecutive bitstring to the value of <fdata>.
Position of fdata<0> bit is determined by <idx>. Leaving the other bits of <in> unmodified. Width of in/out and idx are determined by edges. Width of the bitstring <fdata> is determined by <fwidth>.

in<idx+:fwidth> = fdata; --> in'2 = $setbits(idx,fwidth,fdata,in'1) -->
setbits_5 [label="{{<idx> idx | fwidth=2 | <fdata> fdata | <in> in} | <out> setbits_5}"];

-- add, sadd --
sadd does signed addition which differ from unsigned add in that the smallest operand is sign extended.
add_5 [label="{{<left> left | <right> right} | <out> add_5}"];

-- sub --
signedness?
sub_56 [label="{{<left> left | <right> right} | <out> sub_56}"];

-- mul, smul --
smul does signed multiplication which differ from unsigned mul in that the smallest operand is sign extended.

mul_65 [label="{{<left> left | <right> right} | <out> mul_65}"];

-- and --

Bitwise and. Also used for the && PAC operator since the operands to && is always one bit wide.
and_8 [label="{{<left> left | <right> right} | <out> and_8}"];

-- or --
Bitwise or. Also used for the || PAC operator since the operands to || is always one bit wide.
or_52 [label="{{<left> left | <right> right} | <out> or_52}"];

-- xor --
xor_681 [label="{{<left> left | <right> right} | <out> xor_681}"];

-- eq --
Output is one bit which is set if both operands are equal.
eq_117 [label="{{<left> left | <right> right} | <out> eq_117}"];

-- ne --
Output is one bit which is set if both operands are unequal.
ne_117 [label="{{<left> left | <right> right} | <out> ne_117}"];

-- gt --
Output is one bit which is set if <left> is greater than <right>. Comparison is unsigned.
gt_107 [label="{{<left> left | <right> right} | <out> gt_107}"];

-- gteq --
Output is one bit which is set if <left> is greater or equal than <right>. Comparison is unsigned.
gteq_107 [label="{{<left> left | <right> right} | <out> gt_107}"];

-- lt --
Output is one bit which is set if <left> is less than <right>. Comparison is unsigned.
lt_16 [label="{{<left> left | <right> right} | <out> lt_16}"];

-- lteq --
Output is one bit which is set if <left> is less or equal than <right>. Comparison is unsigned.
lteq_16 [label="{{<left> left | <right> right} | <out> lt_16}"];

-- not --
not_1 [label="{{<in> in} | <out> not_1}"];

-- rev --
Reverses the bit order so that lsb in the input will be msb on the output.
rev_1 [label="{{<in> in} | <out> rev_1}"];

-- ramrd --

Before pipeline insertion:
ramrd_110 [label="{{<exe> exe | <ix> ix } | {r_srcPortMem}|{<out> ramrd_110}}"];
ramrd_110 [label="{{<exe> exe | <ix> ix | <ix2> ix2 } | {r_srcPortMem}|{<out> ramrd_110}}"];
ramrd_110 [label="{{<exe> exe | <ix> ix | <ix2> ix2 | <ix3> ix3 } | {r_srcPortMem}|{<out>
ramrd_110}}"];
After pipeline insertion:
ramrd_110 [label="{{<valid> valid | <exe> exe | <ix> ix } | {r_srcPortMem}|{<out> ramrd_110}}"];

-- ramwr --
Before pipeline insertion:
ramwr_110 [label="{{<in> d | <exe> exe | <ix> ix } | {r_srcPortMem}|{ramwr_110}}"];
ramwr_110 [label="{{<in> d | <exe> exe | <ix> ix | <ix2> ix2 } | {r_srcPortMem}|{ramwr_110}}"];
ramwr_110 [label="{{<in> d | <exe> exe | <ix> ix | <ix2> ix2 | <ix3> ix3 } |
{r_srcPortMem}|{ramwr_110}}"];
After pipeline insertion:
ramwr_110 [label="{{<valid> valid | <in> d | <exe> exe | <ix> ix } |
{r_srcPortMem}|{ramwr_110}}"];

-- assert --
assert_9 [label="{{<in> d | <exe> exe} | assert_9}" print="line:10"];

-- print --
print_9 [label="{{<in> d | <exe> exe } | print_9}" print="and=%d"];

-- regrd --
Before pipeline insertion:
regrd_110 [label="{{<exe> exe } | {r_srcPortMem}|{<out> regrd_110}}"];
regrd_110 [label="{{<exe> exe | <ix> ix } | {r_srcPortMem}|{<out> regrd_110}}"];
regrd_110 [label="{{<exe> exe | <ix> ix | <ix2> ix2 } | {r_srcPortMem}|{<out> regrd_110}}"];
regrd_110 [label="{{<exe> exe | <ix> ix | <ix2> ix2 | <ix3> ix3 } | {r_srcPortMem}|{<out>
regrd_110}}"];
After pipeline insertion:
regrd_110 [label="{{<valid> valid | <exe> exe } | {r_srcPortMem}|{<out> regrd_110}}"];

-- regwr --
Before pipeline insertion:
regwr_110 [label="{{<in> d | <exe> exe } | {rg_data}|{regwr_110}}"];
regwr_110 [label="{{<in> d | <exe> exe | <ix1> ix1 } | {rg_data}|{regwr_110}}"];
After pipeline insertion:
regwr_110 [label="{{<valid> valid | <in> d | <exe> exe } | {rg_data}|{regwr_110}}"];

-- blkwr --
blkwr_2 [label="{{<exe> exe | <in> data} |{learnBlock}| <out> blkwr_2}"];

-- pidff, validff, pidin, pidout, validin,validout, ff

References

McFarland, Michael C., Alice C. Parker, and Raul Camposano. "Tutorial on high-level synthesis." *Proceedings of the 25th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1988.
http://homepages.cae.wisc.edu/~ece734/references/tutorialHLS.pdfZ

http://repository.tudelft.nl/assets/uuid:a775a647-83b8-43c0-978a-44bba80bb5d4/thesis.pdf

A. Hosangadi, F. Fallah, and R. Kastner, Optimizing polynomial expressions by algebraic factorization and common subexpression elimination, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 25 (2006), no. 10, 2012 –2022.
http://cseweb.ucsd.edu/~kastner/papers/tech-poly_factorization_cse.pdf

- Make these optimizations aware of the data transport cost in a pipeline. Depth of expression. Width of operands and intermediate results.

http://guillotjeremie.free.fr/RESEARCH/ARTICLES/DATE09/02.7_1.PDF

Gao, Xitong, Samuel Bayliss, and George A. Constantinides. "SOAP: Structural optimization of arithmetic expressions for high-level synthesis." *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 2013.
http://cas.ee.ic.ac.uk/people/gac1/pubs/XitongFPT13.pdf

Sehwa: a software package for synthesis of pipelines from behavioral specifications

Optimizing Data Flow Graphs to Minimize Hardware Implementation
https://dl.acm.org/citation.cfm?id=1874649

Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs

Contact information

Kenny Ranerup, CEO
kenny.ranerup@packetarc.com