Chapter 4

The Optimizer

During the non-deterministic search process of the vectorizer, it is not clear if a (global) solution of the search process exists, as the vectorizer maintains only weak (local) consistency, ensuring that all SIMD instructions extracted so far are compatible with each other.

For that reason, it is better to postpone all further optimization of the SIMD code until the vectorizer can guarantee that a solution exists. Delaying the following optimization steps, i. e., *peephole optimization* and *scheduling*, until one particular solution has been found entails two positive effects: (*i*) The implementation of the vectorizer is kept simple as vectorization and optimization are not mixed up. (*ii*) An overall speed-up of the vectorization process is achieved, as no effort is wasted in incrementally optimizing partial (local) solutions that are not a part of a global solution.

Once the vectorizer has found a solution and committed to it, the resulting SIMD SSA code is put into the optimizer that consists of two parts.

The first part is a peephole optimizer that matches a set of transformation rules against the SIMD code, substituting code sequences by optimized, semantically equivalent code, until a fixed point is reached. The implemented rewriting rules perform both (*i*) commonplace compiler optimization (carrying out dead code elimination, copy propagation, constant folding, and redundant instruction elimination, a special case of common subexpression elimination) and (*ii*) FFT specific SIMD optimization.

The second part of the optimizer topologically sorts the dag in an attempt to maximize register usage by exploiting specific knowledge about the topological structure of the DFT dag. For transforms whose size is a power of two, the resulting *schedule* provably minimizes the number of register spills, no matter how many registers the target machine has. To improve the quality for non-power of two and SIMD cases, these schedules are adapted accordingly in an extra pass.

The result of that second part of the optimizer, i. e., vectorized, optimized, and scheduled SIMD code, is put into the backend for compilation to x86 assembly.

The remaining parts of this chapter are organized as follows. Section 4.1 introduces the peephole optimizer, explains how it works, and describes what kind of optimizations it performs. Section 4.2 shows how the scheduler works and how its output can be further refined in certain cases.

4.1 Peephole Optimization

Peephole optimization is a local code rewriting technique. A window (*peephole*) slides over several target instructions and special techniques are applied to optimize them according to a set of transformation rules. More specifically, a combination of instructions that matches a specified pattern is transformed into another one according to the associated transformation rules. The newly obtained sequence is either shorter or comprises faster but semantically equivalent instructions. The instructions inside a peephole are logically connected by data dependencies [2].

It is a characteristic of peephole optimization that new rewriting possibilities arise after carrying out a first set of transformations, i. e., several iterations are needed to yield a satisfactory optimization result. Because of the locality of the approach, peephole optimization usually is not able to eliminate all redundancies in a given code.

This section describes the goals of the peephole optimizer in Subsection 4.1.1, the rewriting engine in Subsection 4.1.2, and a set of various rewriting rules in Subsections 4.1.3 through 4.1.5.

4.1.1 Goals of the Peephole Optimizer

The peephole optimizer tries to maximize the computational performance of the input code by pursuing the following goals.

Reduce the Number of Instructions. The output of the vectorizer usually includes many redundant operations, e. g., SIMD swaps. By applying dead code elimination, constant folding, copy propagation, and by rewriting SIMD specific code patterns, the size of the SIMD code can be reduced significantly.

Shorten the Length of the Critical Path. In certain cases, the length of the critical path of the data dependency graph can be shortened by replacing a dependency on some instruction with a dependency on a predecessor of that particular instruction. Applying rules of this group slightly increases the register pressure.

Reduce the Number of Source Operands. Some transformation rules reduce the number of source operands needed to perform an operation, locally reducing the register pressure.

Exploit Target Specific SIMD Features. These rules only work on machines offering intra-operand SIMD instructions, as they focus on substituting patterns using a parallel SIMD style with sequences of intra-operand SIMD instructions.

Rewrite Unsupported Instructions. As some SIMD instructions extracted by the vectorizer may not be available on a particular target machine, all unsupported instructions need to be rewritten into supported ones.

4.1.2 The Rewriting Engine

In FFTW-GEL, peephole optimization is implemented by a committed-choice term rewriting system, using monads for simulating backtracking, including support for indexing to speed up the lookup time for instructions lying in a peephole. Multiple passes are used to split non-terminating rule sets into several terminating ones.

As the actual OCAML implementation is not well-suited for the purpose of illustration, a PROLOG prototype having identical behavior with regard to rule selection and application is presented in the following.

Data-Type Definitions

All optimization rules operate on SIMD instructions. SIMD instructions are defined by the predicate `is_vsimdinstr/1`.

```
:- block is_vsimdinstr(-).
is_vsimdinstr(unaryop2(UOp,S,D)) :-                % Unary Op
   is_vsimdunaryop(UOp),
   is_vsimdregs([S,D]).
is_vsimdinstr(binop2(BOp,S1,S2,D)) :-              % Binary Op
   is_vsimdbinop(BOp),
   is_vsimdregs([S1,S2,D]).
is_vsimdinstr(loadQ2(Array,Index,D)) :-            % Quadword Load
   is_inputarray(Array),
   is_index(Index),
   is_vsimdreg(D).
is_vsimdinstr(loadD2(Access,Array,Index,D)) :-     % Doubleword Load
   is_access(Access),
   is_inputarray(Array),
   is_index(Index),
   is_vsimdreg(D).
is_vsimdinstr(storeQ2(S,Array,Index)) :-           % Quadword Store
   is_vsimdreg(S),
   is_outputarray(Array),
   is_index(Index).
is_vsimdinstr(storeD2(P,S,Access,Array,Index)) :-  % Doubleword Store
   is_lohi(P),
   is_vsimdreg(S),
   is_access(Access),
   is_outputarray(Array),
   is_index(Index).
```

The definition of the predicate `is_vsimdinstr/1` relies on the following straightforward definitions of auxiliary predicates.

```
is_vsimdpos(lo).                                   % Positions in a word
is_vsimdpos(hi).
is_vsimdpos(lohi).
```

```
:- block is_vsimdreg(-).
is_vsimdreg(_).                                   % (structures/atoms)

is_vsimdunaryop(copy).                            % SIMD unary ops
is_vsimdunaryop(swap).
is_vsimdunaryop(chs(P)) :-
   is_vsimdpos(P).
is_vsimdunaryop(mulconst(K1,K2)) :-
   is_floats([K1,K2]).

is_vsimdbinop(i(Op1,Op2)) :-                      % Inter-operand SIMD
   is_scalarbinop(Op1),
   is_scalarbinop(Op2).
is_vsimdbinop(acc(P1,P2)) :-                      % Accumulates
   is_posneg(P1),
   is_posneg(P2).
is_vsimdbinop(unpack(P)) :-
   is_lohi(P).
is_vsimdbinop(shuffle(Bitmask)) :-
   is_integer(Bitmask).

parallelvsimdbinop(i(Op,Op)) :-
   is_scalarbinop(Op).

is_lohi(lo).
is_lohi(hi).

is_posneg(pos).
is_posneg(neg).

loHi_hiLo(lo, hi).
loHi_hiLo(hi, lo).
```

All other auxiliary predicates utilized by the peephole optimizer and a precise definition of the semantics of all data types are provided in Appendix A.

Entry Point of the Peephole Optimizer

The grammar `passes_instrs_peepholeOptimized/5` is the entry point of the rewriting system. Its first argument, `Ps`, is a list of flags that control rule application in the individual passes. The second argument, `Is0`, is a list of SIMD instructions, as defined by the predicate `is_vsimdinstrs/1`. Upon return, the third argument, `Is`, is the list of optimized instructions. The grammar describes a string listing the individual rules that were applied during the rewriting process.

```
passes_instrs_peepholeOptimized(Ps, Is0, Is) -->
   passes_instrs_peepholeOptimized_(Ps, Is0, Is, 0, _UId).
```

Performing Multiple Passes

The grammar `passes_instrs_peepholeOptimized/5` calls another grammar `passes_instrs_peepholeOptimized_/7` to iterate over the list of passes and call `instrs_peepholeOptimized_uid0_uid_flags/7`.

```
passes_instrs_peepholeOptimized_([], Is, Is, UId, UId) -->
   [].
passes_instrs_peepholeOptimized_([Fs|Fss], Is0, Is, UId0, UId) -->
   [newPassWithFlags(Fs)],
   instrs_peepholeOptimized_uid0_uid_flags(Is0, Is1, UId0, UId1, Fs),
   passes_instrs_peepholeOptimized_(Fss, Is1, Is, UId1, UId).
```

Performing One Pass

The grammar `instrs_peepholeOptimized_uid0_uid_flags/7` performs one pass of the rewriting. This includes the following steps.

The predicate `peepholerule/3` is called, passing the current state, i. e., a list of all instructions, a list of flags in this pass, and some auxiliary information needed for creating new symbols in a side-effect free way. `peepholerule/3` picks a peephole optimization rule, in an attempt to apply it. The order, in which rules are picked, is determined by their order in the program text (the first rule is chosen first, the last one last).

If a rule applies, all alternative rules are not considered any more and the same pass is started (first clause of the grammar).

If no rule applies, the fixed point with regard to the rule-set is reached and the list of optimized instructions is returned (second clause of the grammar).

```
instrs_peepholeOptimized_uid0_uid_flags(Is0, Ps, UId0, UId, Fs) -->
   { peepholerule(RuleName, st(UId0,Fs,Is0,Is1), st(UId1,Fs,Is1,Is)) },
   !,                                               % Commit
   [applying(RuleName)],
   instrs_peepholeOptimized_uid0_uid_flags(Is, Ps, UId1, UId, Fs).
instrs_peepholeOptimized_uid0_uid_flags(Ps, Ps, UId, UId, _Fs) -->
   [].
```

The Syntax of Rewriting Rules

All of the following rewriting rules are implemented as clauses of the predicate `peepholerule/3` using a DCG-like syntax to hide the internal state similar to state monads in a functional programming language.

The concrete syntax of rules is defined by the predicate `is_rule/1`. Note that two additional arguments `Xs0` and `Xs` are used for passing around the state.

```
is_rule((peepholerule(Name,Xs0,Xs) :- Body)) :-
   nonvar(Name),
   is_rulebody_(Body, Xs0, Xs).

is_rulebody_((A,B), Xs0, Xs) :-
   is_rulebody_(A, Xs0, Xs1),
   is_rulebody_(B, Xs1, Xs).
is_rulebody_(+(I,Xs0,Xs), Xs0, Xs) :-
   is_vsimdinstr(I).
is_rulebody_(-(I,Xs0,Xs), Xs0, Xs) :-
   is_vsimdinstr(I).
is_rulebody_(=(I,Xs0,Xs), Xs0, Xs) :-
   is_vsimdinstr(I).
is_rulebody_(++(Is,Xs0,Xs), Xs0, Xs) :-
   is_vsimdinstrs(Is).
is_rulebody_(--(Is,Xs0,Xs), Xs0, Xs) :-
   is_vsimdinstrs(Is).
is_rulebody_(==(Is,Xs0,Xs), Xs0, Xs) :-
   is_vsimdinstrs(Is).
is_rulebody_(unusedReg(R,Xs0,Xs), Xs0, Xs) :-
   var(R).
is_rulebody_(unusedRegs(Rs,Xs0,Xs), Xs0, Xs) :-
   is_list(Rs).
is_rulebody_(newSymbol(Symbol,Xs0,Xs), Xs0, Xs) :-
   var(Symbol).
is_rulebody_(newSymbols(Symbols,Xs0,Xs), Xs0, Xs) :-
   is_list(Symbols).
is_rulebody_(flagIsSet(_Flag,Xs0,Xs), Xs0, Xs).
is_rulebody_(target(Ts,Xs0,Xs), Xs0, Xs) :-
   is_list(Ts).
is_rulebody_(pass(Ps,Xs0,Xs), Xs0, Xs) :-
   is_list(Ps).
is_rulebody_(goal(Goal,Xs0,Xs), Xs0, Xs) :-         % any Prolog-goal
   callable(Goal).
```

As this definition is actually executable, it can be used to check all rule definitions for (extended) syntax. The following assertion states that all defined peephole rules have a valid syntax according to the predicate `is_rule/1`.

```
:- \+ (Head = peepholerule(_,_,_),
      clause(Head, Body), \+ is_rule((Head :- Body))).
```

The Semantics of Rewriting Rules

The various parts of the body of a peephole rule have the following meaning.

`-I` removes the instruction `I` from the list of SIMD instructions.
`+I` adds the instruction `I` to an auxiliary list that is added to the list of SIMD instructions, as soon as the rule fires.
`=I` matches a pattern `I` in the list of SIMD instructions, effectively removing the respective instruction from the list of SIMD instructions, adding it again, as soon as the rule fires.

```
% Define new operators (syntactic sugar)
:- op(500, fx, =).

% Add a single instruction to the instruction store
+(P, st(UId,Fs,Is,Ms), st(UId,Fs,Is,[P|Ms])).

% Remove a single instruction to the instruction store
-(P, st(UId,Fs,Is0,Ms), st(UId,Fs,Is,Ms)) :-
   member_of_rest(P, Is0, Is).

% Assert that some instruction is present in the instruction store
=(P) -->
   -P,
   +P.
```

`--Is` applies `-I` for all `I` in the list `Is`.
`++Is` applies `+I` for all `I` in the list `Is`.
`==Is` applies `=I` for all `I` in the list `Is`.

```
:- op(500, fx, ==).
:- op(500, fx, ++).
:- op(500, fx, --).

++[] -->
   [].
++[P|Ps] -->
   +P,
   ++Ps.

--[] -->
   [].
--[P|Ps] -->
   -P,
   --Ps.

==[] -->
   [].
==[P|Ps] -->
   =P,
   ==Ps.
```

`flagIsSet(F)` asserts that `F` is a flag that is set in the internal state.
`target(Ts)` asserts that the target flag is set to one of the values in the list `Ts`.
`pass(Ps)` asserts that the pass flag is set to one of the values in the list `Ps`.
`goal(G)` states that `G` is a provable PROLOG goal.

```
% Assert that a given flag is set.
% This predicate provides parametric control over the application of rules.
flagIsSet(F, st(UId,Fs,Is,Ms), st(UId,Fs,Is,Ms)) :-
   member_of(F, Fs).

target(Ts) -->
   flagIsSet(target(T)),
   { member_of(T, Ts) }.

pass(Ps) -->
   flagIsSet(pass(P)),
   { member_of(P, Ps) }.

goal(G) -->
   { call(G) }.
```

`unusedReg(R)` asserts that there is no instruction present in the current list of SIMD instruction that uses `R` as its input operand.
`unusedRegs(Rs)` applies `unusedReg(R)` for all `R` in the list `Rs`.

```
% Code for determining use of registers/instructions
% actually used for finding out about dependencies
unusedReg(R, st(UId,Fs,Is,Ms), st(UId,Fs,Is,Ms)) :-
   vsimdinstrs_doNotUse(Is, R).

unusedRegs([]) -->
   [].
unusedRegs([R|Rs]) -->
   unusedReg(R),
   unusedRegs(Rs).
```

`newSymbol(X)` allocates a new symbol and unifies it with `X`.
`newSymbols(Xs)` applies `newSymbol(X)` for all `X` in the list `Xs`.

```
% Create a new symbol
newSymbol(s(S0), st(S0,Fs,Is,Ms), st(S,Fs,Is,Ms)) :-
   S is S0+1.

newSymbols([]) -->
   [].
newSymbols([Sym|Syms]) -->
   newSymbol(Sym),
   newSymbols(Syms).
```

4.1.3 Target Architecture Independent Optimization Rules

Optimization rules that improve the code, regardless of the actual target architecture, are presented in the following.

Dead Code Elimination

Dead code is eliminated in the rewriting process by the following rule.

```
peepholerule(eliminate_dead_code) -->
   -I,                                                 % Remove instruction I
   goal(vsimdinstr_writes(I, R)),                      % if its output R
   unusedReg(R).                                       % is unused by others.
```

Copy Propagation

The following five rules perform copy propagation for unary operations, binary operations, and stores.

```
peepholerule(copy_propagation_unaryop) -->             % UnaryOps
   =unaryop2(copy,A,B),
   -unaryop2(Op,B,C),
   +unaryop2(Op,A,C).
peepholerule(copy_propagation_binop_1) -->             % BinOps (1st source)
   =unaryop2(copy,A,B),
   -binop2(Op,B,C,D),
   +binop2(Op,A,C,D).
peepholerule(copy_propagation_binop_2) -->             % BinOps (2nd source)
   =unaryop2(copy,A,B),
   -binop2(Op,C,B,D),
   +binop2(Op,C,A,D).
peepholerule(copy_propagation_doublestore) -->         % Double Stores
   =unaryop2(copy,X,Y),
   -storeD2(Pos,Y,Access,Array,Index),
   +storeD2(Pos,X,Access,Array,Index).
peepholerule(copy_propagation_quadstore) -->           % Quad Stores
   =unaryop2(copy,X,Y),
   -storeQ2(Y,Array,Index),
   +storeQ2(X,Array,Index).
```

Remove Redundant Operations

As FFTW's code generator `genfft` already performs common subexpression elimination, there is no need to implement this functionality again.

However, as the vectorizer may emit redundant operations, e. g., multiple swaps of the same input to different outputs, the respective code patterns need to be optimized. The following two rules implement the detection and elimination of these special cases.

```
peepholerule(remove_redundant_unaryop) -->          % Remove Red. UnaryOps
   =unaryop2(Op,A,B),
   goal(dif(Op, copy)),
   -unaryop2(Op,A,C),
   +unaryop2(copy,B,C).
peepholerule(remove_redundant_binop) -->            % Remove Red. BinOps
   =binop2(Op,A,B,C),
   -binop2(Op,A,B,D),
   +unaryop2(copy,C,D).
```

Constant Folding

Similarly to the scalar case of constant folding, the following rule implements the SIMD case of constant folding.

```
peepholerule(constant_folding) -->                  % Combine UnaryOps
   =unaryop2(Op1,A,B),
   -unaryop2(Op2,B,C),
   goal(dif(Op1,copy)),
   goal(dif(Op2,copy)),
   goal(unaryop_unaryop_combined(Op1, Op2, Op12)),
   +unaryop2(Op12,A,C).
```

Swap Specific Rules

This set of rules is used to topologically shift swap instructions up and down in the instruction sequence looked at. More detailed, it allows for the interchange of swap instructions with (*i*) any unary, and (*ii*) any parallel binary instruction. The application of these rules often results in new instruction sequences, enabling other rules to fire and improve the code.

```
peepholerule(swap_1) -->                            % Eliminate Swap (1)
   =unaryop2(swap,A,B),
   --[unaryop2(swap,C,D), binop2(Op,B,D,E)],
   goal(parallelvsimdbinop(Op)),
   unusedReg(D),
   ++[binop2(Op,A,C,D), unaryop2(swap,D,E)].
peepholerule(swap_2) -->                            % Eliminate Swap (2)
   =unaryop2(swap,A,B),
   --[binop2(Op,B,C,D), unaryop2(swap,D,E)],
   goal(parallelvsimdbinop(Op)),
   unusedReg(D),
   ++[unaryop2(swap,C,D), binop2(Op,A,D,E)].
peepholerule(swap_3) -->                            % Eliminate Swap (3)
   =unaryop2(swap,A,B),
   --[binop2(Op,C,B,D), unaryop2(swap,D,E)],
   goal(parallelvsimdbinop(Op)),
   unusedReg(D),
   ++[unaryop2(swap,C,D), binop2(Op,A,D,E)].
```

Table 4.1 gives an example of swap specific rewriting.

Sequence	⇒	*Sequence'*
	⇒	
unaryop2(swap,A,B) unaryop2(swap,C,D) binop2(add,B,D,E)	⇒	unaryop2(swap,A,B) binop2(add,A,C,D) unaryop2(swap,D,E)

Table 4.1: Rule swap_1. A combination of two swaps and one addition is rewritten. In the resulting sequence, the addition depends on the inputs of its former predecessors. An additional swap of the output of the addition is inserted accordingly.

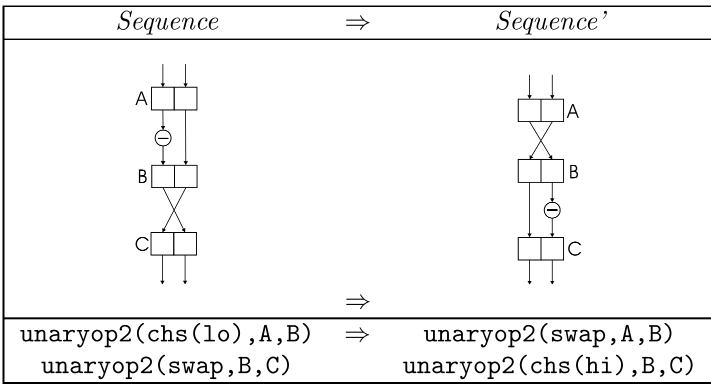
Swap Specific Rules (Multi Pass)

Because of the locality of the peephole optimization, certain combinations that could easily be optimized, can not be rewritten.

There are (at least) two different methods addressing this problem. First, the size of the peephole could be enlarged. Secondly, peephole optimization could be split into multiple consecutive passes that include different rules to move instructions into the peephole of some other rules, enabling further optimization.

As the first strategy would immensely increase the number and the complexity of the optimization rules, hindering verification and debugging, the peephole optimizer of FFTW-GEL implements the second strategy.

Table 4.2 illustrates the working of a multi pass swap rule.



Sequence	⇒	*Sequence'*
	⇒	
unaryop2(chs(lo),A,B) unaryop2(swap,B,C)	⇒	unaryop2(swap,A,B) unaryop2(chs(hi),B,C)

Table 4.2: Rule chsLo_swap_1. The topological position of a sign change operation and a swap is exchanged.

The following rules implement the exchange of the relative position of two unary operations in the SIMD dag.

```
peepholerule(mirror_chsLoSwap_pass1) --> pass([1]),  % Mirror ChsLo/Swap
   --[unaryop2(chs(lo),A,B),
      unaryop2(swap,B,C)],
   unusedReg(B),
   ++[unaryop2(swap,A,B),
      unaryop2(chs(hi),B,C)].
peepholerule(mirror_swapChsHi_pass2) --> pass([2]),  % Mirror Swap/ChsHi
    --[unaryop2(swap,A,B),
       unaryop2(chs(hi),B,C)],
    unusedReg(B),
    ++[unaryop2(chs(lo),A,B),
       unaryop2(swap,B,C)].
peepholerule(mirror_swapChsLo_pass1) --> pass([1]),  % Mirror Swap/ChsLo
   --[unaryop2(swap,A,B),
      unaryop2(chs(lo),B,C)],
   unusedReg(B),
   ++[unaryop2(chs(hi),A,B),
      unaryop2(swap,B,C)].
peepholerule(mirror_chsHiSwap_pass2) --> pass([2]),  % Mirror ChsHi/Swap
   --[unaryop2(chs(hi),A,B),
      unaryop2(swap,B,C)],
   unusedReg(B),
   ++[unaryop2(swap,A,B),
      unaryop2(chs(lo),B,C)].
peepholerule(mulconst_swap_pass1) --> pass([1]),     % Move Swap Over Mul
   --[unaryop2(mulconst(Nl,Nh),A,B),
      unaryop2(swap,B,C)],
   unusedReg(B),
   ++[unaryop2(swap,A,B),
      unaryop2(mulconst(Nh,Nl),B,C)].
peepholerule(swap_mulconst_pass2) --> pass([2]),     % Move Mul Over Swap
   --[unaryop2(swap,A,B),
      unaryop2(mulconst(Nl,Nh),B,C)],
   unusedReg(B),
   ++[unaryop2(mulconst(Nh,Nl),A,B),
      unaryop2(swap,B,C)].
```

Sign Change Specific Rules

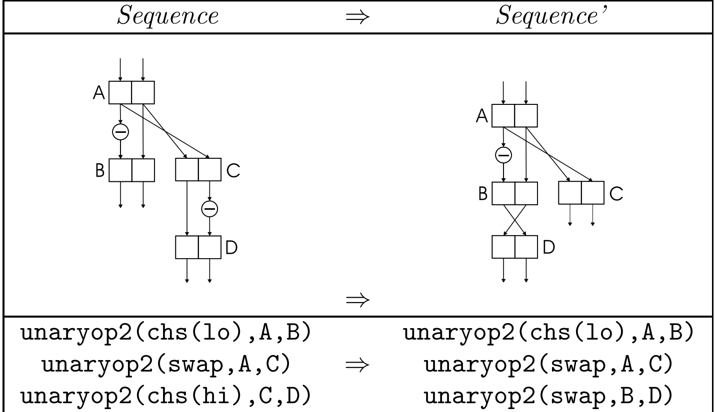
The vectorizer may extract mixed add/sub SIMD instructions from scalar code. As these instructions are not directly supported by any of the instruction sets of the Intel Pentium 4, AMD K7, or AMD K6, parallel mixed add/sub SIMD instructions are translated into one addition or subtraction plus an appropriate sign change operation by one of the rules in Subsection 4.1.5.

Sign change specific transformations try to remove sign change instructions or to replace sign changes with cheaper instructions, e. g., swap. The following program code provides an implementation of these rules.

```
peepholerule(chs_1) -->                                % Elim. Chs (1)
   ==[unaryop2(chs(lo),A,B), binop2(i(add,add),V,W,_D)],
   --[unaryop2(chs(hi),A,C), binop2(i(add,add),X,Y,E)],
   unusedRegs([B,C]),
   goal(member_of(B, [V,W])),
   goal(member_of(t(C,Z), [t(X,Y), t(Y,X)])),
   +binop2(i(sub,sub),Z,B,E).
peepholerule(chs_2) -->                                % Elim. Chs (2)
   --[unaryop2(chs(P1),A,B), unaryop2(chs(P2),C,D),
      binop2(i(add,add),B,D,E)],
   unusedRegs([B,D]),
   goal(member_of(t(P1,P2,P3), [t(lo,hi,lo), t(hi,lo,hi)])),
   ++[unaryop2(chs(P3),D,E), binop2(i(sub,sub),A,C,D)].
peepholerule(chs_3) -->                                % Elim. Chs (3)
   ==[unaryop2(chs(P1),A,B), unaryop2(swap,A,C)],
   -unaryop2(chs(P2),C,D),
   unusedReg(C),
   goal(loHi_hiLo(P1,P2)),
   +unaryop2(swap,B,D).
```

Table 4.3 illustrates how sign change instructions can be transformed into more favorable swap instructions.

Sequence	⇒	*Sequence'*



unaryop2(chs(lo),A,B) unaryop2(swap,A,C) unaryop2(chs(hi),C,D)	⇒	unaryop2(chs(lo),A,B) unaryop2(swap,A,C) unaryop2(swap,B,D)

Table 4.3: Rule chs_1. One swap instruction and two sign change operations are replaced with a computationally cheaper combination of one sign change operation and two swaps.

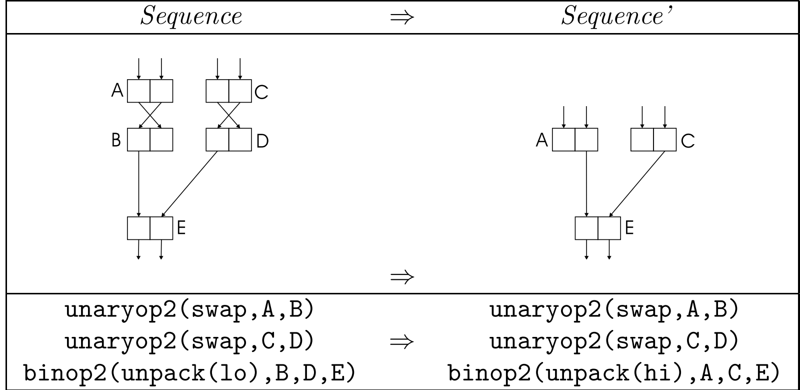
Unpack Specific Rules

When some rule converts intra-operand SIMD style code into parallel SIMD style code, unpack operations are used.

The following rules simplify certain combinations of unpacks and neighboring swap instructions.

```
peepholerule(unpack_1) -->                               % Mirror Unpack
   -binop2(unpack(LH),B,D,E),
   ==[unaryop2(swap,A,B), unaryop2(swap,C,D)],
   goal(loHi_hiLo(LH, HL)),
   +binop2(unpack(HL),A,C,E).
peepholerule(unpack_2) -->                               % UnpackLo+Swap
   --[binop2(unpack(lo),A,B,C), unaryop2(swap,C,D)],
   unusedReg(C),
   +binop2(unpack(lo),B,A,D).
peepholerule(unpack_3) -->                               % UnpackHi+Swap
   --[binop2(unpack(hi),A,B,C), unaryop2(swap,C,D)],
   unusedReg(C),
   +binop2(unpack(hi),B,A,D).
```

Table 4.4 illustrates the working of one of the rules presented.



Sequence	⇒	*Sequence'*
unaryop2(swap,A,B)		unaryop2(swap,A,B)
unaryop2(swap,C,D)	⇒	unaryop2(swap,C,D)
binop2(unpack(lo),B,D,E)		binop2(unpack(hi),A,C,E)

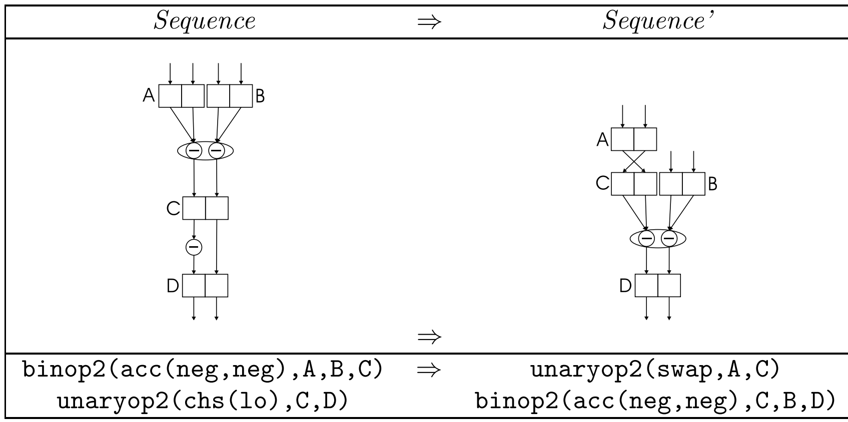
Table 4.4: Rule unpack_1. An unpack operation depending on two swaps is transformed to an unpack that depends on the sources of these two swaps.

Optimizing unpacks is done mostly for the benefit of architectures that have minimal or no support for intra-operand SIMD style (AMD K6 and Intel Pentium 4).

Accumulate Specific Rules (Single/Multi Pass)

The following rules exploit identities of accumulate instructions with regard to swapping of the input/output. Note that the rules `acc_1` and `acc_3` execute in different passes to prevent non-termination.

```
peepholerule(acc_1) --> pass([1]),                      % Swap+[NP]PAcc
   =unaryop2(swap,A,B),
   -binop2(acc(Op,pos),C,B,D),
   +binop2(acc(Op,pos),C,A,D).
peepholerule(acc_2) -->                                 % Swap+P[NP]Acc
   =unaryop2(swap,A,B),
   -binop2(acc(pos,Op),B,C,D),
   +binop2(acc(pos,Op),A,C,D).
peepholerule(acc_3) --> pass([2]),                      % Swap+NPAcc
   =unaryop2(swap,A,B),
   -binop2(acc(neg,pos),B,A,C),
   unusedReg(B),
   +binop2(acc(neg,pos),B,B,C).
peepholerule(acc_4) -->                                 % 2*Swap+NNAcc+Mul
   ==[unaryop2(swap,A,B), unaryop2(swap,C,D)],
   --[binop2(acc(neg,neg),B,D,E), unaryop2(mulconst(Nl,Nh),E,F)],
   unusedReg(E),
   goal((Ml is -Nl, Mh is -Nh)),
   ++[binop2(acc(neg,neg),A,C,E), unaryop2(mulconst(Ml,Mh),E,F)].
peepholerule(acc_5) -->                                 % XXAcc+Swap
   --[binop2(acc(Op,Op),A,B,C), unaryop2(swap,C,D)],
   unusedReg(C),
   +binop2(acc(Op,Op),B,A,D).
peepholerule(acc_6) -->                                 % NNAcc+Chs
   --[binop2(acc(neg,neg),A,B,C), unaryop2(chs(P),C,D)],
   unusedReg(C),
   goal(member_of(t(P,X,Y,Z), [t(lo,A,C,B), t(hi,B,A,C)])),
   ++[unaryop2(swap,X,C), binop2(acc(neg,neg),Y,Z,D)].
```



Sequence	⇒	*Sequence'*
(diagram: A, B, C, D)	⇒	(diagram: A, C, B, D)
binop2(acc(neg,neg),A,B,C) unaryop2(chs(lo),C,D)	⇒	unaryop2(swap,A,C) binop2(acc(neg,neg),C,B,D)

Table 4.5: Rule acc_1. A sign change operation is moved up over a negative-negative accumulate instruction and converted into a swap.

4.1.4 Target Architecture Specific Optimization Rules

While the rules presented so far were independent of the target architecture, FFTW-GEL also offers additional optimizations targeted at the peculiarities of a particular target architecture.

The following optimization rules focus on optimizing code patterns that frequently occur in FFT code ("butterflies").

Rules for Extracting Accumulates

Certain combinations of instructions allow to transform patterns containing parallel SIMD additions/subtraction instructions into intra-operand style SIMD instructions, i. e., to extract new mixed positive-negative or negative-positive accumulate instructions.

This rewriting is only done, however, when generating code for an architecture that actually supports the respective instructions. At the moment, the AMD K7 is the only target architecture supported by FFTW-GEL meeting all these criteria.

The following rewriting rules implement the extraction of mixed accumulate instructions.

```
peepholerule(k7_opt_1) --> target([k7]),              % K7: Chs+Sub
    =unaryop2(swap,A,B),
    --[unaryop2(chs(lo),B,C), binop2(i(sub,sub),A,C,D)],
    unusedReg(C),
    +binop2(acc(pos,neg),B,B,D).
peepholerule(k7_opt_2) --> target([k7]),              % K7: Swap+Chs+Add
    --[unaryop2(swap,A,B), unaryop2(chs(lo),B,C),
       binop2(i(add,add),X,Y,D)],
    unusedRegs([B,C]),
    goal(member_of(t(X,Y), [t(A,C), t(C,A)])),
    +binop2(acc(neg,pos),A,A,D).
peepholerule(k7_opt_3) --> target([k7]),              % K7: Swap+Chs+Add
    --[unaryop2(swap,A,B), unaryop2(chs(hi),A,C),
       binop2(i(add,add),X,Y,D)],
    unusedRegs([A,C]),
    goal(member_of(t(X,Y), [t(B,C), t(C,B)])),
    +binop2(acc(pos,neg),A,A,D).
```

4.1.5 Rules for Rewriting Unsupported Instructions

Although the vectorizer tries to avoid the extraction of SIMD instructions that are not supported by the target architecture (e. g., intra-operand instructions on a Intel Pentium 4), such instructions are sometimes needed to prevent the vectorization from failure.

Whenever the vectorizer extracts instructions that are not available on the target architecture, the optimizer first tries to improve code sequences containing these

instructions by applying target architecture independent optimization rules as presented in Subsection 4.1.3.

As target architecture independent optimization rules have no specific knowledge about the instructions actually supported by the target machine, they do not convert intra-operand instructions into parallel instructions (or vice versa) or rewrite swaps into equivalent sequences of code. This step of target architecture dependent rewriting is left to a final pass of the optimizer that is described in the following.

Rewriting Mixed Parallel Instruction

Neither the AMD K6, nor the AMD K7, nor the Intel Pentium 4 supports mixed parallel add-sub and sub-add SIMD instructions. The following two rules rewrite these instructions into a sequence of one sign change operation and one addition.

```
peepholerule(rewrite_addsub) -->
   -binop2(i(add,sub), S1, S2, D),
   newSymbol(T),
   ++[unaryop2(chs(hi), S2, T), binop2(i(add,add), S1, T, D)].
peepholerule(rewrite_subadd) -->
   -binop2(i(sub,add), S1, S2, D),
   newSymbol(T),
   ++[unaryop2(chs(lo), S2, T), binop2(i(add,add), S1, T, D)].
```

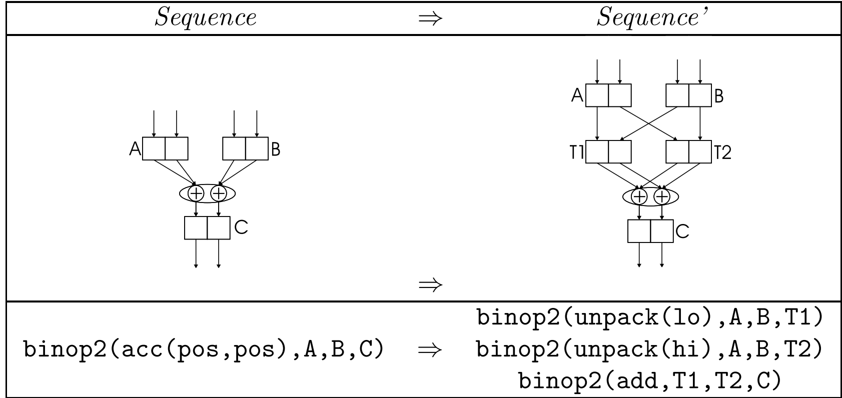
Rewriting Accumulate Instructions

The vectorizer knows about four different variants of accumulate instructions (positive-positive, positive-negative, negative-positive, and negative-negative).

While the AMD K7 offers all but the second one (positive-negative), the AMD K6 supports only the first one (positive-positive), and the Intel Pentium 4 does not implement any of these variants.

Table 4.6 illustrates the rewriting of a positive-positive accumulate instruction, implemented by the following Prolog code.

```
peepholerule(no_ppacc) --> target([p4]),            % P4: Rewrite PPAcc
   -binop2(acc(pos,pos),A,B,C),
   newSymbols([T1,T2]),
   ++[binop2(unpack(lo),A,B,T1), binop2(unpack(hi),A,B,T2)],
   +binop2(i(add,add),T1,T2,C).
```

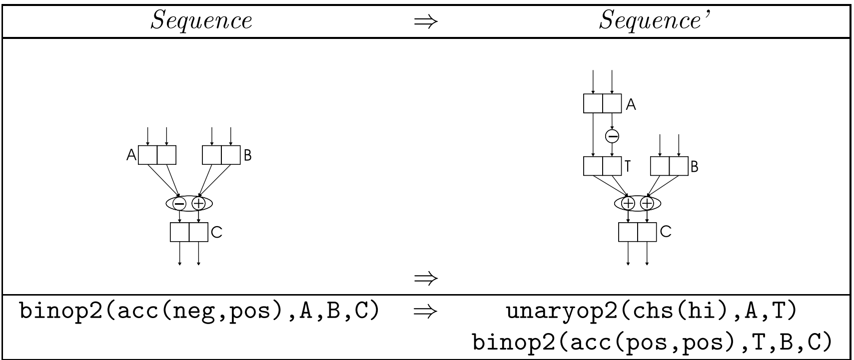


Sequence	⇒	*Sequence'*
binop2(acc(pos,pos),A,B,C)	⇒	binop2(unpack(lo),A,B,T1) binop2(unpack(hi),A,B,T2) binop2(add,T1,T2,C)

Table 4.6: Rule no_ppacc. When generating code for the Intel Pentium 4, positive-positive accumulate instructions are rewritten into a sequence of two unpacks and one addition.

Unlike the AMD K7, the AMD K6 and the Intel Pentium 4 do not offer a negative-negative and a negative-positive accumulate instruction. When generating code for any of these target architectures, alternative equivalent sequences are used instead, as shown in Table 4.7.

Note that for the Pentium 4, an application of rule `no_npacc` causes a future application of rule `no_ppacc`.



Sequence	⇒	*Sequence'*
binop2(acc(neg,pos),A,B,C)	⇒	unaryop2(chs(hi),A,T) binop2(acc(pos,pos),T,B,C)

Table 4.7: Rule no_npacc. For architectures not supporting negative-positive accumulate instructions (AMD K6 and Intel Pentium 4), equivalent code of one sign change operation and one positive-positive accumulate instruction is generated.

The following rules implement the rewriting of the remaining three variants of accumulate instructions.

```
peepholerule(no_nnacc) --> target([k6,p4]),          % K6/P4: Rewrite NNAcc
   -binop2(acc(neg,neg),A,B,C),
   newSymbols([T1,T2]),
   ++[binop2(unpack(lo),A,B,T1), binop2(unpack(hi),A,B,T2)],
   +binop2(i(sub,sub),T1,T2,C).
```

```
peepholerule(no_npacc) --> target([k6,p4]),          % K6/P4: Rewrite NPAcc
   -binop2(acc(neg,pos),A,B,C),
   newSymbol(T),
   ++[unaryop2(chs(hi),A,T), binop2(acc(pos,pos),T,B,C)].
peepholerule(no_pnacc_k6_p4) --> target([k6,p4]),    % K6/P4: Rewrite PNAcc
   -binop2(acc(pos,neg),A,B,C),
   newSymbol(T),
   ++[unaryop2(chs(hi),B,T), binop2(acc(pos,pos),A,T,C)].
peepholerule(no_pnacc_k7) --> target([k7]),          % K7: Rewrite PNAcc
   -binop2(acc(pos,neg),A,B,C),
   newSymbol(T),
   ++[binop2(acc(neg,pos),B,A,T), unaryop2(swap,T,C)].
```

Note that positive-negative accumulate instructions are rewritten into different code sequences, depending on the target architecture. For the AMD K7, a sequence of one negative-positive accumulate and one swap is generated, while, for the AMD K6 and the Intel Pentium 4, a sign change instruction and a positive-positive accumulate is output.

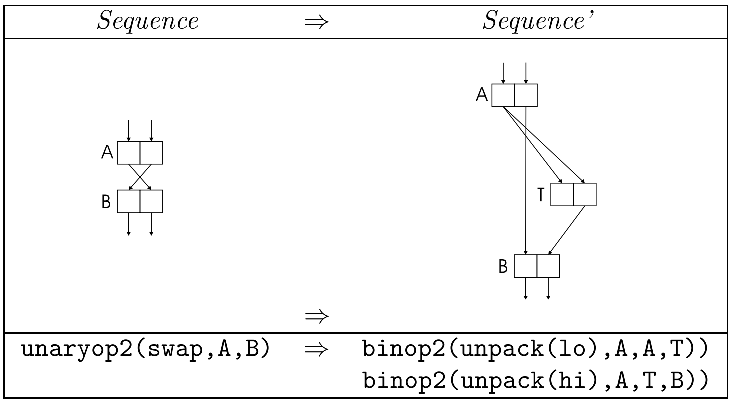
Rewriting Swap Instructions

Both the Intel Pentium 4 and the AMD K6 do not have a non-destructive SIMD swap instruction, like the AMD K7 does.

However, the Intel Pentium 4 has a binary SIMD instruction `shuffle` that can be used to emulate a (destructive) swap. The following rule shows the trivial translation of swaps to shuffles.

```
peepholerule(p4_rewrite_swap) --> target([p4]),      % P4: Rewrite Swap
   -unaryop2(swap,A,B),
   +binop2(shuffle(9),A,A,B).
```

Unlike the Intel Pentium 4, the AMD K6 does not support a shuffle instruction that can be used for implementing a cheap SIMD swap. Thus, one swap needs to be substituted with two unpack instructions, as shown in Table 4.8.



Sequence	⇒	*Sequence'*
	⇒	
`unaryop2(swap,A,B)`	⇒	`binop2(unpack(lo),A,A,T))` `binop2(unpack(hi),A,T,B))`

Table 4.8: Rule k6_no_swap_1. A swap instruction is replaced by two dependent unpacks.

When generating code for the AMD K6, the following rules are applied.

```
peepholerule(k6_no_swap_1) --> target([k6]),          % K6: Rewrite Swap (1)
   -unaryop2(swap,A,B),
   unusedReg(A),
   newSymbol(T),
   ++[binop2(unpack(lo),A,A,T),
      binop2(unpack(hi),A,T,B)].
peepholerule(k6_no_swap_2) --> target([k6]),          % K6: Rewrite Swap (2)
   -unaryop2(swap,A,B),
   newSymbol(T),
   ++[binop2(unpack(hi),A,A,T),
      binop2(unpack(lo),T,A,B)].
```

Note that there are two rules for rewriting a swap instruction when generating code the AMD K6. The first one, `k6_swap_1`, asserts that the source of the swap is not read by any other instruction. The code sequence the swap is rewritten to allows for more parallelism when it is compiled by the the backend, resulting in higher performance. The second rule, `k6_swap_2`, covers the general case.

4.2 The Scheduler

The scheduling process of FFTW-GEL is virtually identical with its counterpart in FFTW 2.1.3. It has been extended from operating on scalar instructions to be equivalently usable on virtual SIMD instructions. In the following, the goals and the basic functionality of the scheduler will be described.

The scheduling phase produces a topologically sorted order of the dag, i. e., a list of static single assignments, called a *schedule*. This schedule can be executed by a sequential processor. The goal of scheduling is to minimize the variable lifetime in the resulting code to enhance locality, i. e., reduce register pressure by reducing register spills.

The scheduling process is divided into a scheduling phase and into an annotated scheduling phase.

The *scheduling phase* transforms the dag into a recursive decomposition of serial and parallel subdags. A serial decomposition specifies that a subdag `D1` has dependencies to another subdag `D2` and therefore `D1` has to be executed before `D2`. In a parallel decomposition the relative execution order of a subdag is not specified.

The *annotated scheduling phase* annotates a serial order onto parallel blocks of the serial-parallel dag. Parallel blocks using mostly the same set of register variables are scheduled consecutively, i. e., they get an annotation which defines their scheduled order. This order is optimized w. r. t. minimizing variable lifetime. Therefore, the annotated scheduler finds the smallest subdag that encompasses the entire lifespan of the variable. This is necessary for finding nested scopes in a set of subdags.

A more detailed description of the scheduler and an example illustrating its functionality can be found in [23].

4.2.1 Improvements of the Scheduler

As the access patterns occurring in SIMD vectorized code may differ significantly from the scalar case, FFTW-GEL uses a set of heuristics to reorder instructions, thus breaking FFTW's schedule.

Moreover, there are usually more unary instructions in SIMD code than in scalar code, because these instructions are needed to do auxiliary operations, e. g., data shuffling and sign change operations.

To improve the output of FFTW's scheduler of in these cases, FFTW-GEL employs a local reordering strategy, trying to reduce the register pressure.

In the following *move up* refers to moving an instruction upwards in the program text until it shares an operand with its immediate predecessor. Analogously, *move down* refers to moving an instruction downwards in the program text until it shares an operand with its immediate successor.

The following steps are performed: (*i*) Loads are moved down. (*ii*) Stores are moved up. (*iii*) Unary instructions, whose source operand is not read by any instruction following in the program text, are moved up. (*iv*) All other unary instructions are moved down. (*v*) Binary instructions are moved up if both of their source operand are not used in the following program text. This step has turned out to be especially beneficial for SIMD codes mainly consisting of intra-operand instructions.