Interface and Extension of the Open Research Compiler

Sebastian Pop

September 5, 2002

1 Introduction

This report presents the work I've been working on during my *Diplôme d'Études Supérieures Spécialisées* internship of the *Université Louis Pasteur de Strasbourg*.

My work during this internship was supervised by Albert Cohen in the *A3 project* at the *Institut National de Recherche en Informatique et en Automatique*.

The *A3 project* aims at optimizing performance of a large range of programs on new processor architectures, either high-performance general-purpose processors or embedded/specialized processors. A3 is a joint project between *INRIA Rocquencourt* and the *PRiSM laboratory* (University of Versailles and CNRS).

The main contributions of this project are:

- a description of the ORC compiler,
- a hyperlink based documentation of the algorithms coming from ORC's source code,
- an algebraic specification of a partial redundancy elimination (PRE) based on the work of Ivan Djelic,
- a reimplementation of the predicate partition graph for including support for atomic predicates,
- and an implementation of the PRE pass based on atomic predicates.

After a global description of the ORC compiler, and a short description of the PRE algorithm, the algebraic specification of the PPG construction and the PRE algorithm is included. Finally, I conclude this report with a short description of the implementation method, and some general notes about this internship project.

2 Open64/ORC compilers

2.1 An industrial compiler for researchers

The Open64 compiler is an open source project that originated with the SGI Pro64 compiler. The Open64 compiler is currently maintained by researchers from the University of Delaware and from the University of Alberta. The Open Research Compiler (ORC) is an extension of the Open64 compiler developed jointly by Intel and Chinese Academy of Sciences, Institute of Computing Technology (ICT).

SGI launched the Pro64 project back in 1999 in order to provide a state of the art compiler for EPIC architectures. Like RISC or CISC before it, EPIC isn't a strict set of rules or a design specification; it's the name for a collection of techniques and an overall design philosophy. Since EPIC was founded on the learnings comming from existing RISC and CISC architectures and on the latest advances in compiler technology, EPIC architecture places the compiler in a central point. The compiler is thus expected to produce code that takes profit from hardware support.

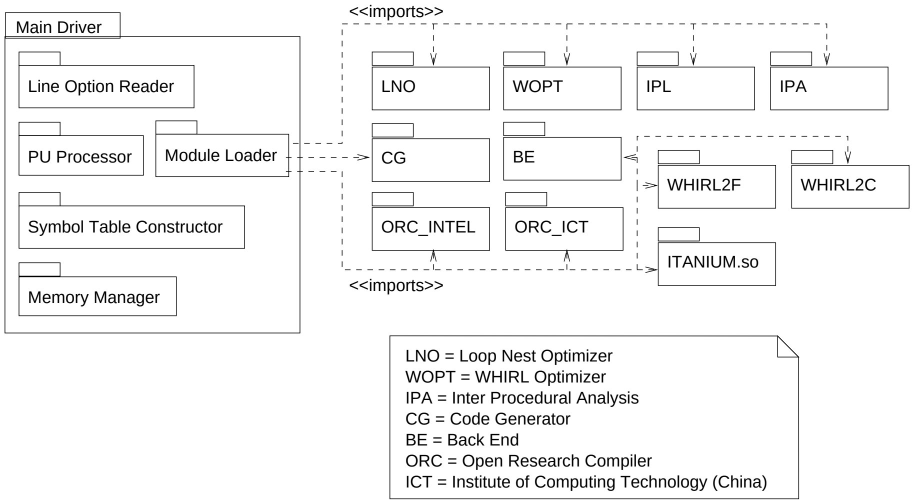
As the EPIC architecture is only in its early days, compiler research community is very active and involved in this challenge.

ORC provides an open-source compiler that targets the Itanium IA-64 implementation of EPIC. ORC's industrial origins makes the compiler a robust support for research projects; it implements a great repertoire of optimization passes ranging from the high level loop nest optimizations (LNO) to the low level software pipelining (SWP).

A global overview of the entire compiler is provided in the next subsection as a top-down decomposition; then a description of the main intermediate representation follows. Further informations about different passes can be found in source files as comments, or in the following structured web page [6].

2.2 Open64/ORC architecture

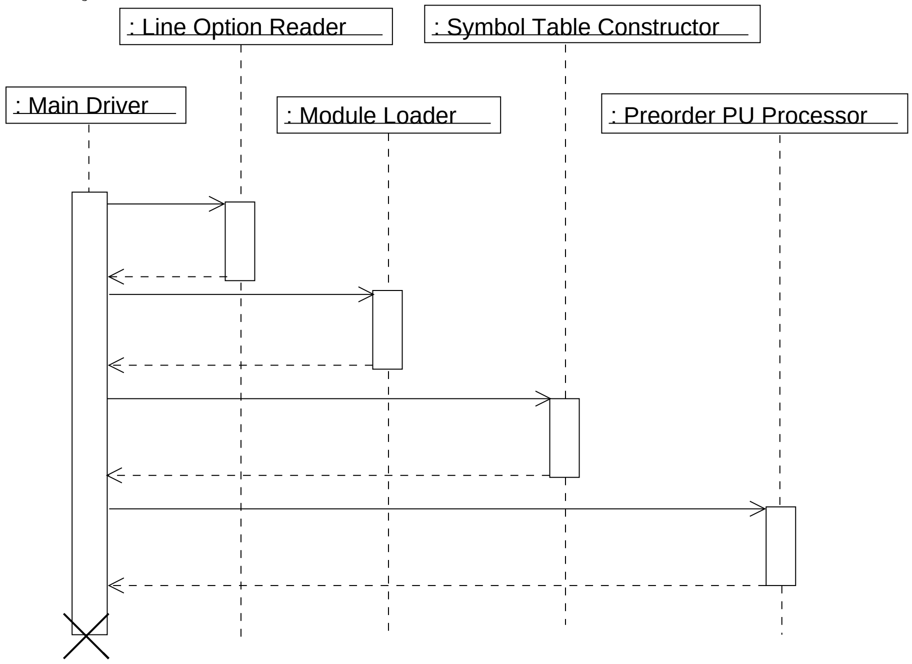
The compiler is built around a main driver (./osprey1.0/be/be/driver.cxx) that orchestrates the execution flow.



Each module defines an extern interface that allows the main driver to deal with initialization, finalization, and entry points. For example extern symbols defined in lno.so are:

```
From be/lno/lnodriver.h
extern void lno_main (INT argc, char **argv, INT, char **);
extern void Lno_Init (void);
extern void Lno_Fini (void);
extern WN *Perform_Loop_Nest_Optimization (PU_Info*, WN *, WN *, BOOL);
```

The main driver is in charge of recognizing line parameters then loading shared objects:



Symbol Table Constructor parses the file and constructs the intermediate representations (IR): the WHIRL and a symbol table (ST). IRs are then checked for consistency and are passed to the *Preorder PU Processor*.

A Processing Unit (PU) is the basic unit on which the compiler works for most of optimizations and for which the compiler produces assembler code. A PU is a function in the C language, and nested functions are represented as distinct PUs.

The compiler uses different abstractions for representing information comming from the processed source code. This information is stored under different structures in order to make accesses to this information more effective. The integrity of informations stored in all these structures is thus crucial, and an optimization has to transfer its effects on all these structures.

Analyzes aim to enrich existant representations with their results, or to construct new structures containing these results.

Lowering transformations keep the same amount of information. They aim to reduce different aspects of the same information to a unique representative. For example syntactic desugaring can be considered as a lowering process. Thus lowering processes transform IR informations into *semantic-equivalent* informations belonging to a subset of the same IR.

Traductions are not supposed to keep the same amount of information (for example assembler does not keep information about types we can find in high level programming languages). However traductions have to keep correct *execution-equivalence*.

2.2.1 Winning Hierarchical Intermediate Representation (WHIRL)

WHIRL defines five layers in order to perform the right optimizations on the right abstraction level.

- Very High level
- High level
- Middle level
- Low level
- Very low level

WHIRL node structure (class WN) is declared in *./osprey1.0/common/com/wn_core.h*. The interface for accessing WHIRL nodes is described in the source code. These comments are gathered in the web page [6] under the section *WHIRL nodes*.

WHIRL representation is based on the structure of tree, where nodes represent:

- expressions;
- leaves (constants, references to the symbol table, ...);
- statments (the side-effecting roots of expression trees commonly called "stats" in other compilers);
- units of hierarchical control flow such as do loops and if statements.

WHIRL targets the C abstract machine; i.e. the result of the WHIRL lowering remains a subset of the C language and has no specific architecture junks (specific instructions proper to a given architecture). This makes the targeting to a new architecture simpler since it concerns only the translation to the CGIR module, and doesn't affect the WHIRL optimizer.

It is difficult to figure out how much modular this architecture is, since a single target architecture was provided in Open64 compiler: the Linux IA-64 architecture. There exist other back-ends specific to MIPS processors, but these target architectures were not released by SGI under GPL.

Translation between the different levels of WHIRL is performed by progressive lowering. The lowering is performed following the LOWER_ACTIONS argument passed to WN_Lower (function defined in ./osprey1.0/be/com/wn_lower.cxx).

In the following call-graph diagrams the WN_Lower function appears whenever the lowering is needed. Some of these lowerings are performed only under some constraints. The best place for checking these constraints is the source code.

2.2.2 Other Intermediate representations

WHIRL is the main representation used in the compiler. It uses a symbol table in order to store information about types. Thus the main representation of the compiler doesn't contain all the information, but uses other apropriate structures for storing specific information.

The *Control Flow Graph* (CFG) is embedded in the WHIRL representation since the WHIRL contains units of hierarchical control flow such as do loops and if statements. However this information is often gathered in a graph structure in order to avoid to re-walk the whole AST for every access. CFG reduces the AST representation to a single aspect: its control flow. The CFG is called a *sublanguage* or an *aspect language* in [4].

Another important aspect of a declarative language is the definition and use of a variable (def-use). The corresponding aspect language is the *Static Single Assignment* (SSA). SSA is the main representation used by optimizers in WOPT. It is stored in a map over all WHIRL nodes and its purpose is to represent data-flow (def-use) information in an effective way. For further informations on the SSA form, see Muchnick [2] p.252, and for further informations on SSA implementation in Open64, see the web page [6].

Dependences between stores and reads of a same variable are gathered in Data Dependence Graphs (DDG). There are 3 levels of dependence graphs:

- LNO DDG expressing dependences between array accesses,
- a level DDG for fission and fusion,
- and Code Generator DDG for scalars/array accesses in the innermost loop (the array deps are derived from LNO and alias analysis). See dep_graph.h for details.

Finally there are more specific representations like

- *LEGO* for representing data distributions,
- access arrays and vectors used to represent array and bounds expressions
- and polytopes

As usual, more informations about all these IR can be found on the web page [6].

A last translation transforms WHIRL trees into a list of instructions for a given architecture. Convert_WHIRL_To_OPs initializes the Code Generator. The *Code Generator Intermediate Representation* (CGIR) is the main representation of the CG. Instructions belongs to basic blocks such that the CFG is encoded dirrectly in the CGIR.

2.2.3 General notes about dumping Intermediate Representations

An optimizer compiler has to provide tools for verifying results of optimizations. Intermediate Representations (IR) are generally the first structures to be dumped during the development process. But this is generally not enough since the end-user has to understand the dumping format.

A pretty printer is generally used to transform the IR back to the original programming language. The pretty printer doesn't dump enough information in order to be compilable again. This supposes to be able to keep or reconstruct a too big amount of information. The later is the technique used in Open64, and the result is compilable. However the dumper is not really useful for projects re-engineering since source comments are all lost as well as macro definitions during the preprocessing for the C and C++ front-ends. F90 parser keeps comments since some of them contains directives (for parallelisation for example), and thus it allows to dump comments as well.

Thus the main role of a pretty printer is to write the results of an optimizer in a human readable form. Another goal of a pretty printer is to verify a parser:

$$unparser(parser(unparser(parser(file)))) == unparser(parser(file))$$

Open64 provides two pretty printers: one that dumps the WHIRL representation back into C language: *whirl2c*, and another for pretty printing into Fortran: *whirl2f*.

For compiler developement purposes, Open64 contains a WHIRL browser: *WB_BROWSER* can be invoked directly from the debugger as documented in the source code (see also appendix A for general notes about how to debug Open64 compiler):

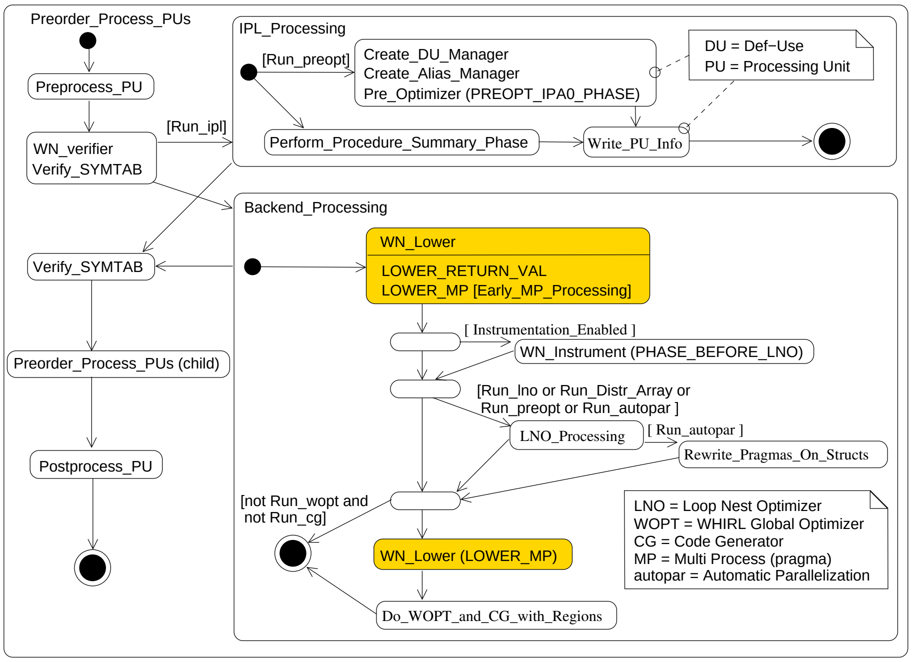
```
//-----------------------------------------------------------------------
// NAME: WB_BROWSER::Debug
// FUNCTION: The main entry point to the whirl browser.  To start up the
//   whirl browser in dbx, say 'p debug()'.
//-----------------------------------------------------------------------
```

The WHIRL can also be dumped into an ASCII file. A parser can reconstruct the original WHIRL tree. An extra file containing symbol tables is needed for completing the WHIRL tree information. Thus if the ASCII WHIRL is modified consistently with respect to symbol tables, then the modified ASCII WHIRL can be reparsed for initializing the WHIRL representation. ASCII WHIRL can be used for experiment hand-written WHIRL transformations.

2.2.4 Preorder PU processor

Next subsections explore the call graph of the compiler, beginning with the top level. There are two different paths for the compilation of Processing Units (PU):

- the PU is dumped into a .o file for further Inter Procedural Analysis (IPA), or
- the PU is passed to the backend. The PU is optimized through the Loop Nest Optimizer (LNO) and the WHIRL Optimizer (WOPT) and finally is passed to the Code Generator (CG) that will emit code.



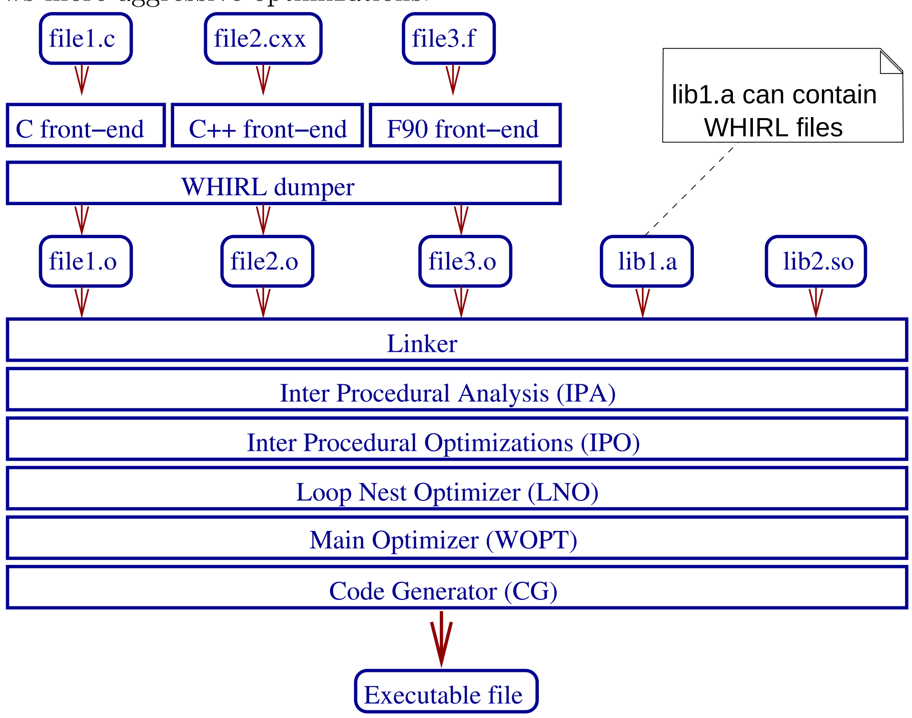
2.2.5 Inter Procedural Analysis (IPA)

Open64 has support for IPA analyzes and optimizations. IPA combines information from several processing units in order to take profit from a global information over the whole project.

Main interprocedural optimizations include:

- dead function elimination
- interprocedural constant propagation
- memory disambiguation for precise alias analysis

IPA's implementation is based on the reconstruction of a global AST representation at the end of the building process. For this, the compiler dumps the intermediate representation just after the first lowering process finished (ie. fake .o files contains *High level WHIRL* trees). At link time if some .o or .a files contain WHIRL trees, the compiler enables the inter procedural analysis and optimizations, then perform all the remaining optimization passes. The information gathered over the whole project is more precise and thus allows more aggressive optimizations.

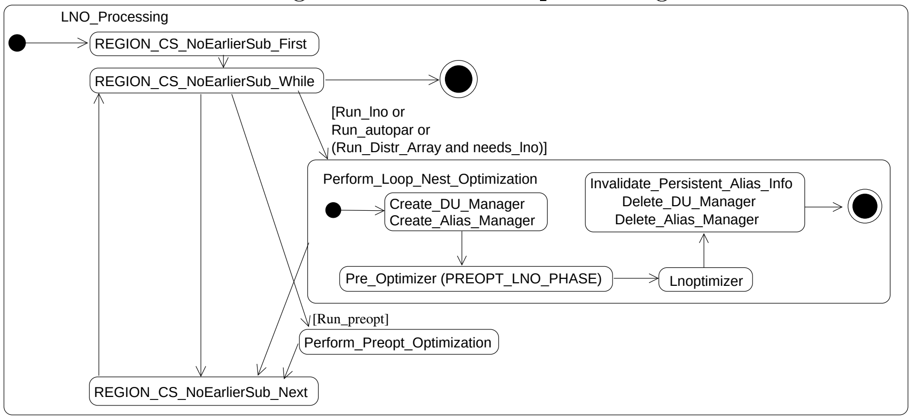


2.2.6 Loop Nest Optimizer (LNO)

LNO works on the *High level WHIRL* and thus if IPA is enabled it takes profit from the interprocedural optimizations. After the quick overview of the LNO package, the reader can consult the web page [6] for further informations about different passes. This web page contains algorithm descriptions for each pass. These notes were gathered from source code and classified in main passes groups.

The following description is a top-down decomposition of the call graph in which only main components were included. Thus the following schemas are correct under the assumption that the reader is not concerned by consistency calls, nor by feedback debugging calls, ... For a more precise overview the reader can browse the source code.

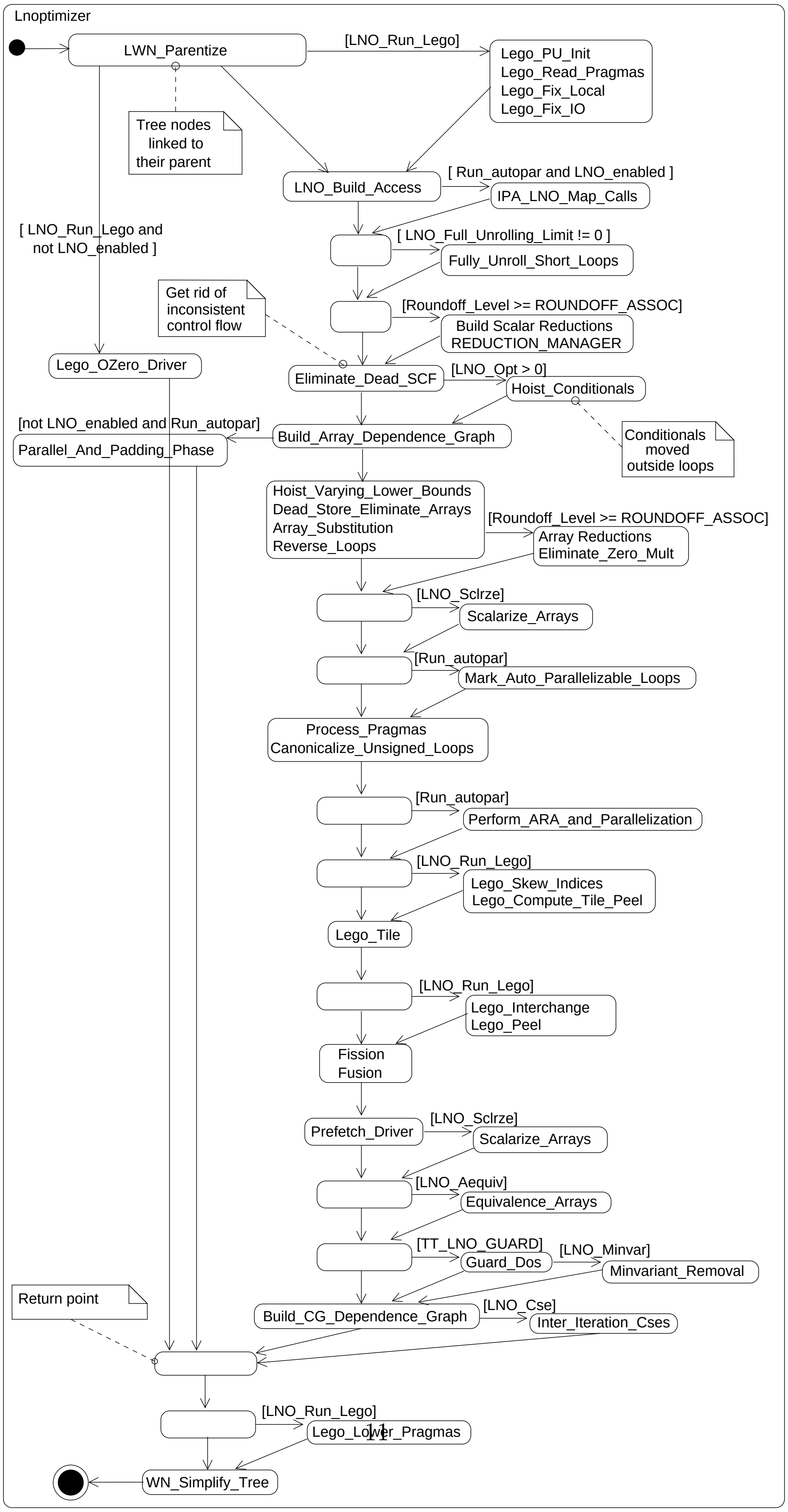
An iteration over all Regions in the current processing unit calls the LNO.



Perform_ Preopt_ Optimization is called in place of LNO if -PHASE:p is set.

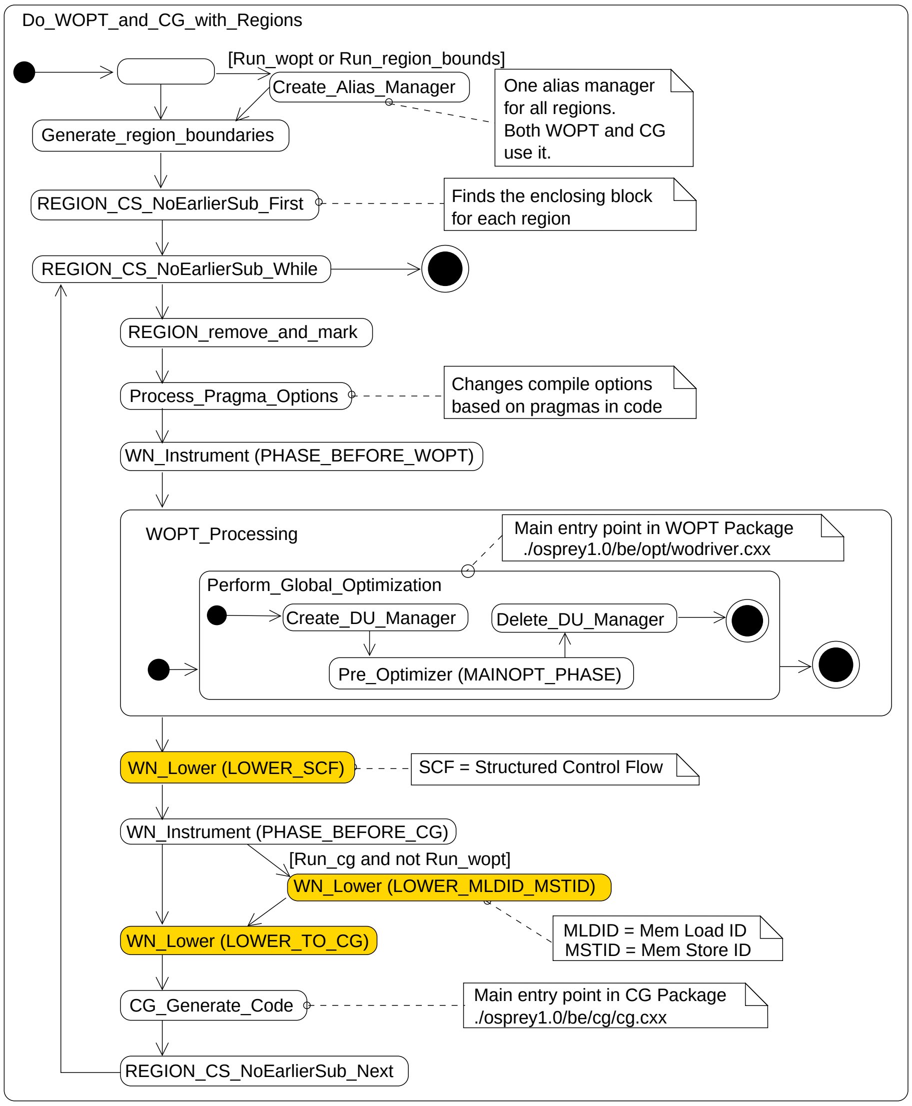
The entry point in the LNO package is in *osprey1.0/be/lno/lnodriver.cxx*. LNO keeps information from alias analysis and SSA during all the phases in this package. It frees this information at the end of the LNO, but transforms the dependence graph into a dependence graph used by the code generator as we'll see later.

Lnoptimizer can be found in *osprey1.0/be/lno/lnopt_ main.cxx*.

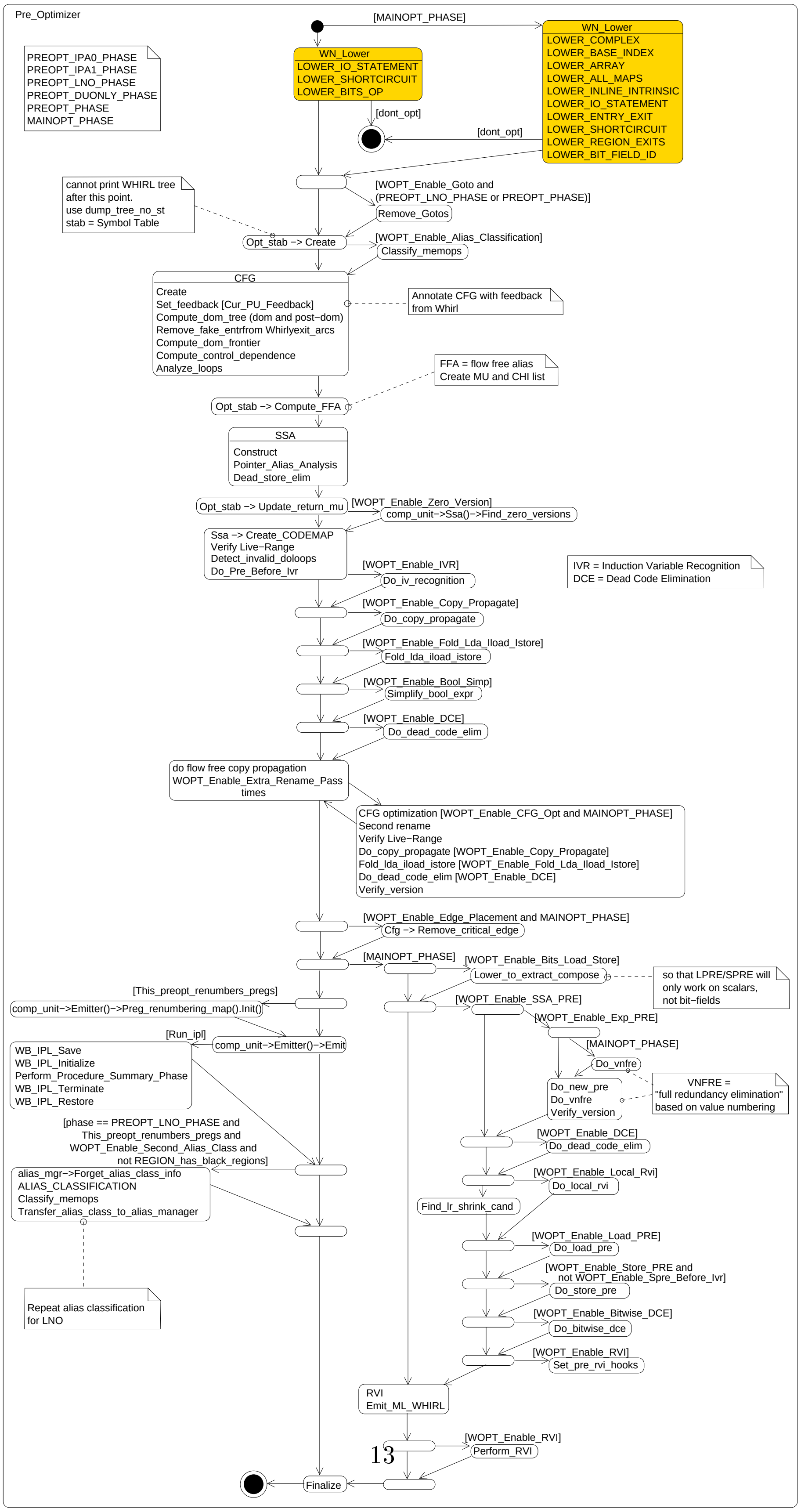

Lnoptimizer
LWN_Parentize
[LNO_Run_Lego]
Lego_PU_Init
Lego_Read_Pragmas
Lego_Fix_Local
Lego_Fix_IO
Tree nodes linked to their parent
LNO_Build_Access
[Run_autopar and LNO_enabled]
IPA_LNO_Map_Calls
[LNO_Run_Lego and not LNO_enabled]
[LNO_Full_Unrolling_Limit != 0]
Fully_Unroll_Short_Loops
Get rid of inconsistent control flow
[Roundoff_Level >= ROUNDOFF_ASSOC]
Build Scalar Reductions
REDUCTION_MANAGER
Lego_OZero_Driver
Eliminate_Dead_SCF
[LNO_Opt > 0]
Hoist_Conditionals
[not LNO_enabled and Run_autopar]
Parallel_And_Padding_Phase
Build_Array_Dependence_Graph
Conditionals moved outside loops
Hoist_Varying_Lower_Bounds
Dead_Store_Eliminate_Arrays
Array_Substitution
Reverse_Loops
[Roundoff_Level >= ROUNDOFF_ASSOC]
Array Reductions
Eliminate_Zero_Mult
[LNO_Sclrze]
Scalarize_Arrays
[Run_autopar]
Mark_Auto_Parallelizable_Loops
Process_Pragmas
Canonicalize_Unsigned_Loops
[Run_autopar]
Perform_ARA_and_Parallelization
[LNO_Run_Lego]
Lego_Skew_Indices
Lego_Compute_Tile_Peel
Lego_Tile
[LNO_Run_Lego]
Lego_Interchange
Lego_Peel
Fission
Fusion
Prefetch_Driver
[LNO_Sclrze]
Scalarize_Arrays
[LNO_Aequiv]
Equivalence_Arrays
[TT_LNO_GUARD]
Guard_Dos
[LNO_Minvar]
Minvariant_Removal
Return point
Build_CG_Dependence_Graph
[LNO_Cse]
Inter_Iteration_Cses
[LNO_Run_Lego]
Lego_Lower_Pragmas
WN_Simplify_Tree

2.2.7 Main Optimizer (WOPT)

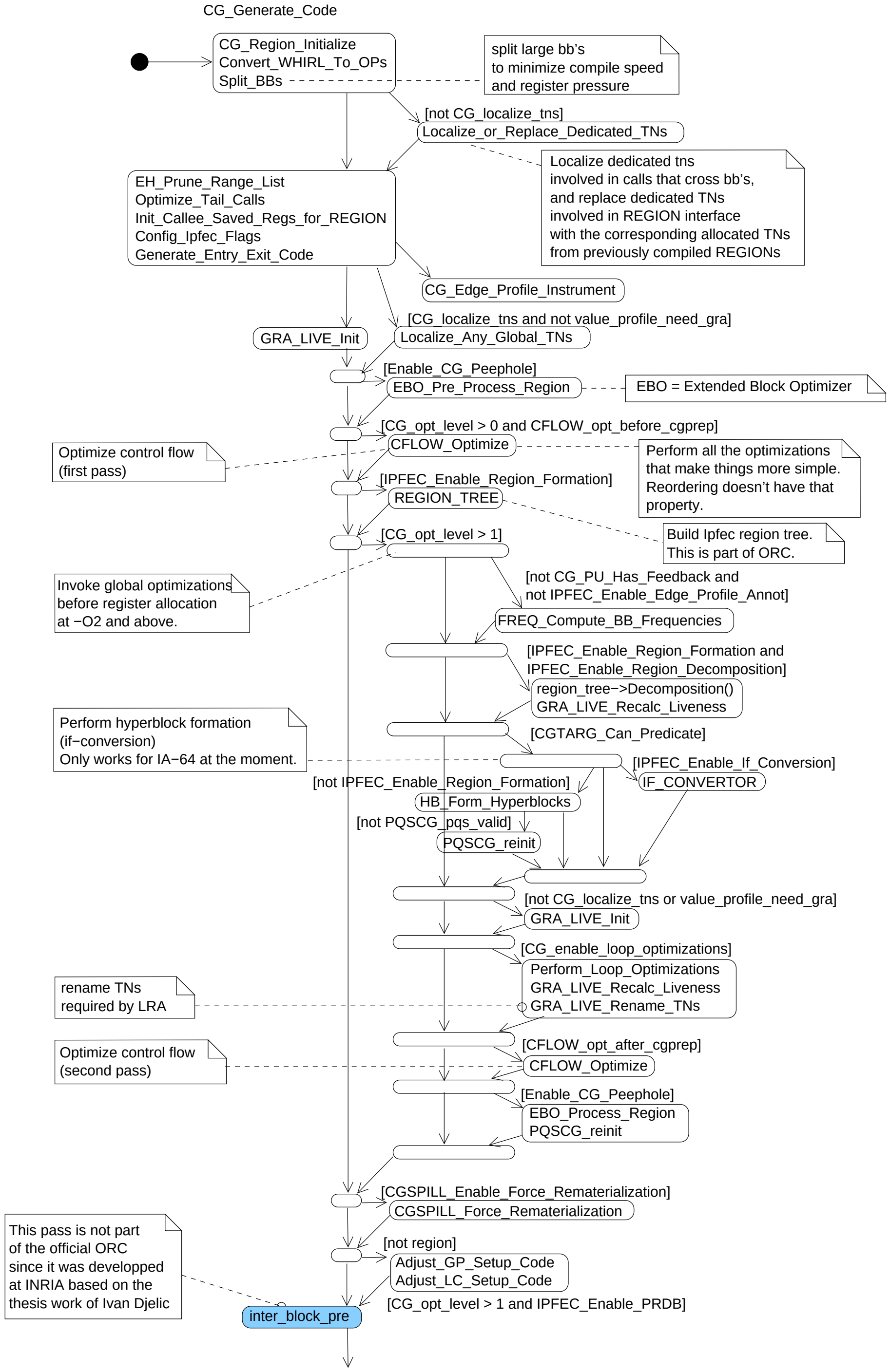
WOPT is the defalut optimizer in the compiler. It shares the same alias manager with the code generator. The main driver calls WOPT once per region and passes the result directly to the CG. This behavior is equivalent to GCC's function-at-time model (GCC parses a function and constructs its AST representation, traduces AST to RTL and finally produces assembly code, then process the next function).

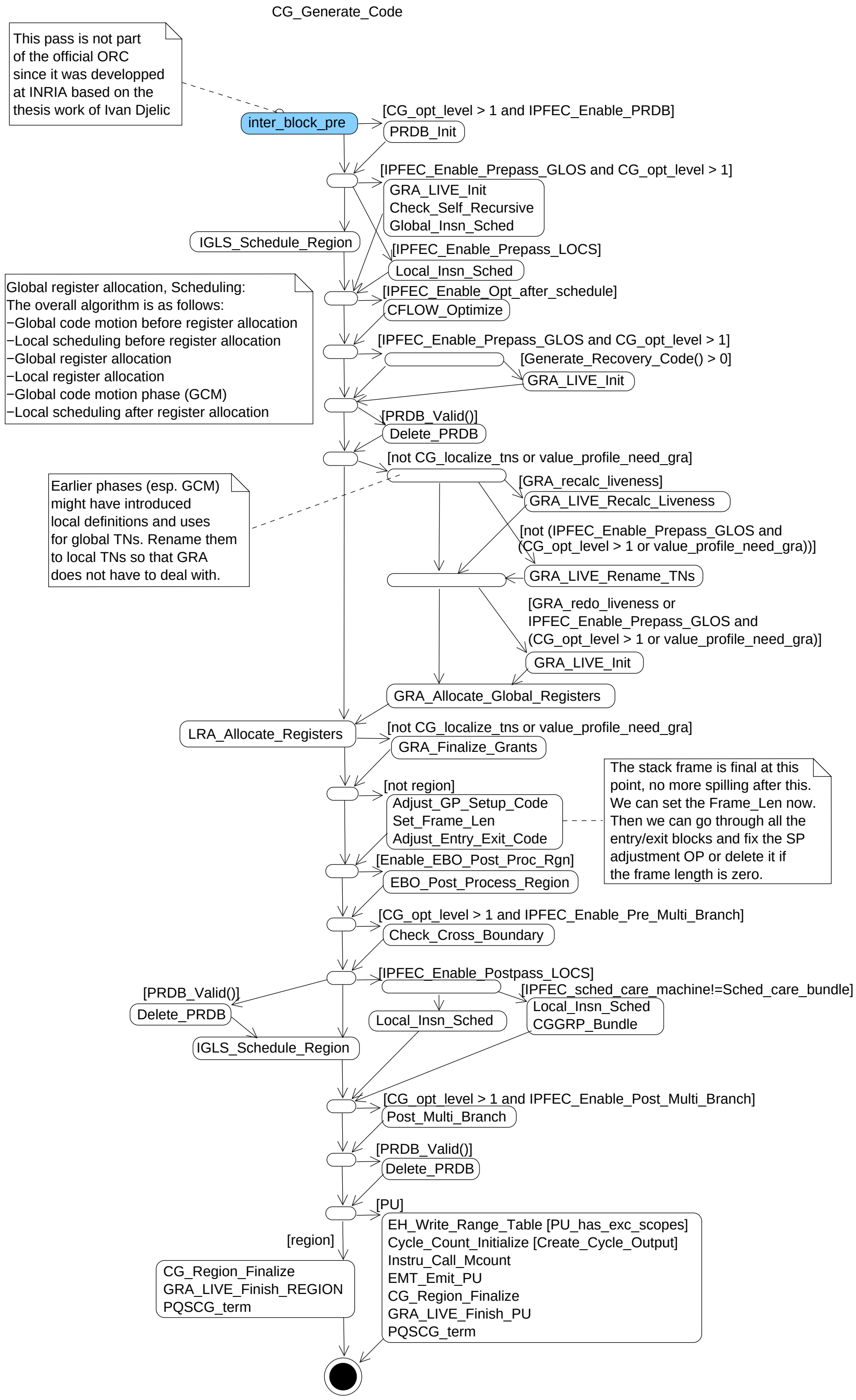


WOPT entry point can be found in: *osprey1.0/be/opt/opt_main.cxx*.


Pre_Optimizer
[MAINOPT_PHASE]
WN_Lower
LOWER_COMPLEX
LOWER_BASE_INDEX
LOWER_ARRAY
LOWER_ALL_MAPS
LOWER_INLINE_INTRINSIC
LOWER_IO_STATEMENT
LOWER_ENTRY_EXIT
LOWER_SHORTCIRCUIT
LOWER_REGION_EXITS
LOWER_BIT_FIELD_ID
PREOPT_IPA0_PHASE
PREOPT_IPA1_PHASE
PREOPT_LNO_PHASE
PREOPT_DUONLY_PHASE
PREOPT_PHASE
MAINOPT_PHASE
WN_Lower
LOWER_IO_STATEMENT
LOWER_SHORTCIRCUIT
LOWER_BITS_OP
[dont_opt]
[dont_opt]
cannot print WHIRL tree after this point. use dump_tree_no_st stab = Symbol Table
[WOPT_Enable_Goto and (PREOPT_LNO_PHASE or PREOPT_PHASE)]
Remove_Gotos
Opt_stab −> Create
[WOPT_Enable_Alias_Classification]
Classify_memops
CFG
Create
Set_feedback [Cur_PU_Feedback]
Compute_dom_tree (dom and post−dom)
Remove_fake_entrfrom Whirlyexit_arcs
Compute_dom_frontier
Compute_control_dependence
Analyze_loops
Annotate CFG with feedback from Whirl
FFA = flow free alias
Create MU and CHI list
Opt_stab −> Compute_FFA
SSA
Construct
Pointer_Alias_Analysis
Dead_store_elim
Opt_stab −> Update_return_mu
[WOPT_Enable_Zero_Version]
comp_unit−>Ssa()−>Find_zero_versions
Ssa −> Create_CODEMAP
Verify Live−Range
Detect_invalid_doloops
Do_Pre_Before_Ivr
[WOPT_Enable_IVR]
Do_iv_recognition
IVR = Induction Variable Recognition
DCE = Dead Code Elimination
[WOPT_Enable_Copy_Propagate]
Do_copy_propagate
[WOPT_Enable_Fold_Lda_Iload_Istore]
Fold_lda_iload_istore
[WOPT_Enable_Bool_Simp]
Simplify_bool_expr
[WOPT_Enable_DCE]
Do_dead_code_elim
do flow free copy propagation
WOPT_Enable_Extra_Rename_Pass times
CFG optimization [WOPT_Enable_CFG_Opt and MAINOPT_PHASE]
Second rename
Verify Live−Range
Do_copy_propagate [WOPT_Enable_Copy_Propagate]
Fold_lda_iload_istore [WOPT_Enable_Fold_Lda_Iload_Istore]
Do_dead_code_elim [WOPT_Enable_DCE]
Verify_version
[WOPT_Enable_Edge_Placement and MAINOPT_PHASE]
Cfg −> Remove_critical_edge
[MAINOPT_PHASE]
[WOPT_Enable_Bits_Load_Store]
Lower_to_extract_compose
so that LPRE/SPRE will only work on scalars, not bit−fields
[This_preopt_renumbers_pregs]
comp_unit−>Emitter()−>Preg_renumbering_map().Init()
[WOPT_Enable_SSA_PRE]
[WOPT_Enable_Exp_PRE]
[Run_ipl]
comp_unit−>Emitter()−>Emit
[MAINOPT_PHASE]
Do_vnfre
WB_IPL_Save
WB_IPL_Initialize
Perform_Procedure_Summary_Phase
WB_IPL_Terminate
WB_IPL_Restore
Do_new_pre
Do_vnfre
Verify_version
VNFRE = "full redundancy elimination" based on value numbering
[phase == PREOPT_LNO_PHASE and This_preopt_renumbers_pregs and WOPT_Enable_Second_Alias_Class and not REGION_has_black_regions]
[WOPT_Enable_DCE]
Do_dead_code_elim
alias_mgr−>Forget_alias_class_info
ALIAS_CLASSIFICATION
Classify_memops
Transfer_alias_class_to_alias_manager
[WOPT_Enable_Local_Rvi]
Do_local_rvi
Find_lr_shrink_cand
[WOPT_Enable_Load_PRE]
Do_load_pre
[WOPT_Enable_Store_PRE and not WOPT_Enable_Spre_Before_Ivr]
Do_store_pre
Repeat alias classification for LNO
[WOPT_Enable_Bitwise_DCE]
Do_bitwise_dce
[WOPT_Enable_RVI]
Set_pre_rvi_hooks
RVI
Emit_ML_WHIRL
[WOPT_Enable_RVI]
Perform_RVI
Finalize

2.2.8 Code Generator (CG)




CG_Generate_Code
This pass is not part of the official ORC since it was developped at INRIA based on the thesis work of Ivan Djelic
inter_block_pre
[CG_opt_level > 1 and IPFEC_Enable_PRDB]
PRDB_Init
[IPFEC_Enable_Prepass_GLOS and CG_opt_level > 1]
GRA_LIVE_Init
Check_Self_Recursive
Global_Insn_Sched
IGLS_Schedule_Region
[IPFEC_Enable_Prepass_LOCS]
Local_Insn_Sched
Global register allocation, Scheduling:
The overall algorithm is as follows:
−Global code motion before register allocation
−Local scheduling before register allocation
−Global register allocation
−Local register allocation
−Global code motion phase (GCM)
−Local scheduling after register allocation
[IPFEC_Enable_Opt_after_schedule]
CFLOW_Optimize
[IPFEC_Enable_Prepass_GLOS and CG_opt_level > 1]
[Generate_Recovery_Code() > 0]
GRA_LIVE_Init
[PRDB_Valid()]
Delete_PRDB
[not CG_localize_tns or value_profile_need_gra]
Earlier phases (esp. GCM) might have introduced local definitions and uses for global TNs. Rename them to local TNs so that GRA does not have to deal with.
[GRA_recalc_liveness]
GRA_LIVE_Recalc_Liveness
[not (IPFEC_Enable_Prepass_GLOS and (CG_opt_level > 1 or value_profile_need_gra))]
GRA_LIVE_Rename_TNs
[GRA_redo_liveness or IPFEC_Enable_Prepass_GLOS and (CG_opt_level > 1 or value_profile_need_gra)]
GRA_LIVE_Init
GRA_Allocate_Global_Registers
LRA_Allocate_Registers
[not CG_localize_tns or value_profile_need_gra]
GRA_Finalize_Grants
[not region]
Adjust_GP_Setup_Code
Set_Frame_Len
Adjust_Entry_Exit_Code
The stack frame is final at this point, no more spilling after this. We can set the Frame_Len now. Then we can go through all the entry/exit blocks and fix the SP adjustment OP or delete it if the frame length is zero.
[Enable_EBO_Post_Proc_Rgn]
EBO_Post_Process_Region
[CG_opt_level > 1 and IPFEC_Enable_Pre_Multi_Branch]
Check_Cross_Boundary
[IPFEC_Enable_Postpass_LOCS]
[IPFEC_sched_care_machine!=Sched_care_bundle]
[PRDB_Valid()]
Delete_PRDB
Local_Insn_Sched
Local_Insn_Sched
CGGRP_Bundle
IGLS_Schedule_Region
[CG_opt_level > 1 and IPFEC_Enable_Post_Multi_Branch]
Post_Multi_Branch
[PRDB_Valid()]
Delete_PRDB
[PU]
[region]
EH_Write_Range_Table [PU_has_exc_scopes]
Cycle_Count_Initialize [Create_Cycle_Output]
Instru_Call_Mcount
EMT_Emit_PU
CG_Region_Finalize
GRA_LIVE_Finish_PU
PQSCG_term
CG_Region_Finalize
GRA_LIVE_Finish_REGION
PQSCG_term

3 Partial Redundancy Elimination (PRE)

After a short description of the general PRE algorithm, I will discuss the characteristics of the PRE algorithm I implemented. At the end of this global overview I included an algebraic specification of the PRE algorithm on predicated code. This specification is based on Ivan Djelic thesis work [1].

3.1 PRE: overview

Ref. Muchnick – Section 13.3 – p.407

A *Partial Redundancy* is a computation that is done more than once on some path through the flowgraph.

The PRE pass aims to eliminate all these redundant computations by replacing them by a unique computation. This computation is placed at an optimal point in the flowgraph and its result is stored in a temporary variable. The temporary variable replaces all the subsequent computations.

The substitution is performed only if some properties based on the flowgraph are true. Basically for a given expression e there are two main properties for each basic block in the control flowgraph: *anticipability* and *availability*.

The first step of the PRE algorithm computes these properties. The second step consists in replacing the expression by the temporary variable, following the valuation of these properties.

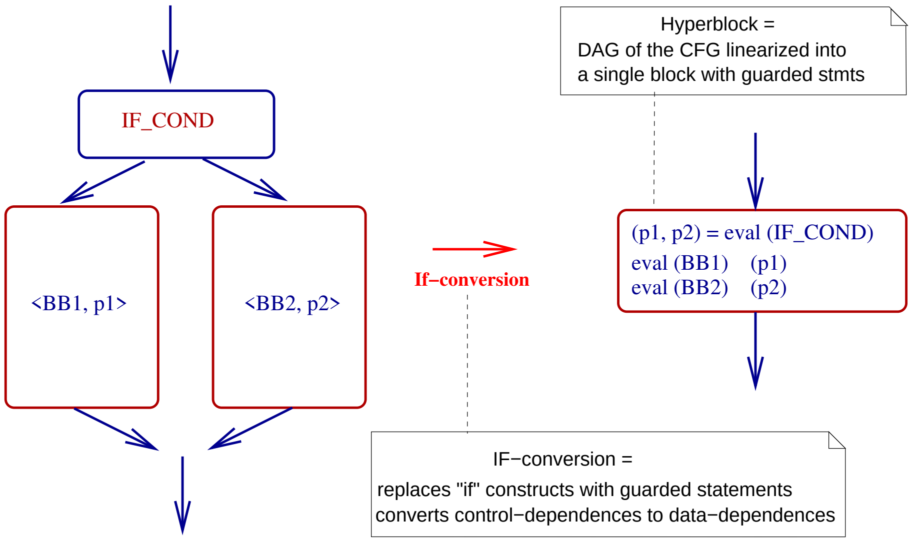
3.2 PRE on guarded code

3.2.1 Predication support on EPIC

Predicated code is one of the new features EPIC architectures have to implement. The underlying idea is to avoid small branches introduce bubbles into the pipeline. For this, each branch is allocated a boolean register that *predicates* the results of the entire branch. This allows either parallel execution of both branches or sequential execution with code reordering among branches for improving data locality. Results are committed or discarded following the value of the boolean register.

3.2.2 An example of predicated code

```
Original code:              Predicated code:

if (cond) {                 p1 p2 = cmpp (cond)
  a = x + y;                a = x + y              (p1)
  b = x + y;                x = 1                  (p2)
}                           b = x + y              (p1)
else
  x = 1;
```



3.2.3 Predicate Query System (PQS)

PQS is a representation of the relations among predicate registers. The leading idea is that the compiler have to be aware about predicates disjoint and subset relations. There are several techniques that deal with efficiently answer these queries:

- Predicate Partition Graph (PPG)
- Predicate Relation DataBase (PRDB)
- Atomic Predicates (AP)

ORC constructs a PRDB as well as a PPG for a given region.

3.2.4 Predicate Partition Graph (PPG)

PPG represents in a hierarchical way predicates as subsets in a directed acyclic graph. This graph is built under the following pseudo algorithm (an algorithmic specification can be found later).

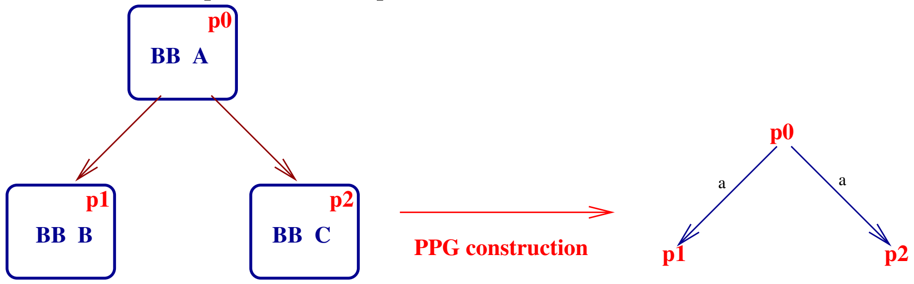
PPG nodes are built either:

- for a materialized predicate (a predicate that appears explicitly in the instruction)
- for a control predicate (a predicate that controls the execution of a whole basic block)

and thus the same PPG node could be the representative of a materialized predicate and of a control predicate (for example when a branch instruction defines two predicates and at the same time splits the control flow into two basic blocks).

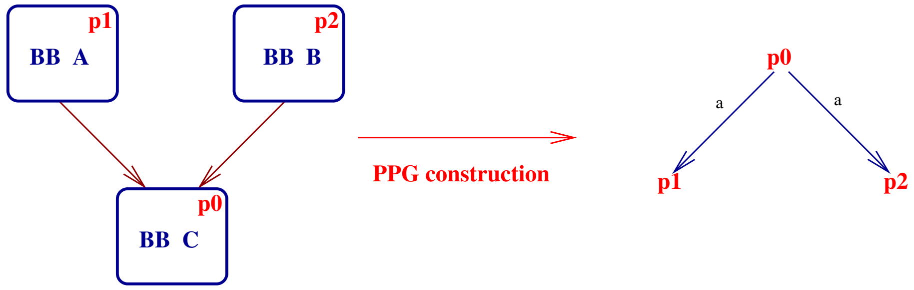
Relations (edges) between PPG nodes are added in a second pass and are labeled by pairs. These pairs represent partitions generated by comparison instructions.

A control flow split defines a partition.



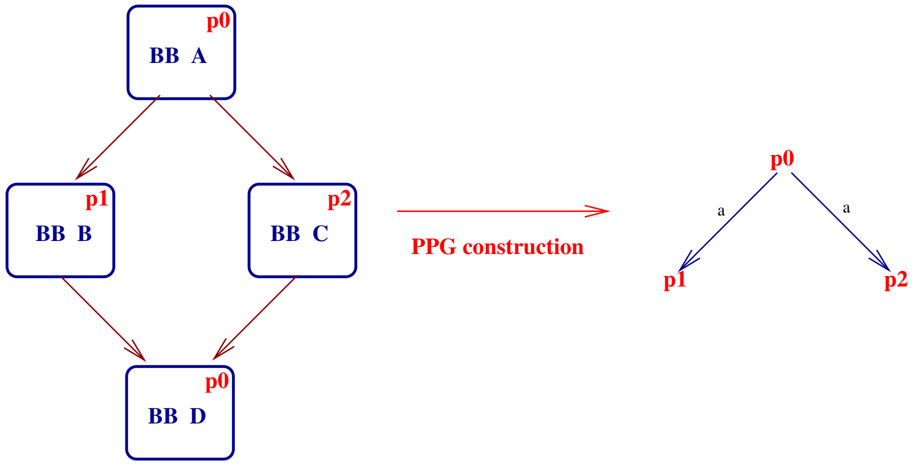
For example if the CFG contains three basic blocks A, B and C and two edges (A, B) and (A, C), then this information is mapped on the partition graph by adding two oriented edges between the PPG node corresponding to basic block A and PPG nodes corresponding to basic blocks B and C.

A control flow join defines a partition.



For example if the CFG contains three basic blocks A, B and C and two edges (A, C) and (B, C), then this information is mapped on the partition graph by adding two oriented edges between the PPG node corresponding to basic block C and PPG nodes corresponding to basic blocks A and B.

Furthermore, if the CFG contains two control equivalent basic blocks (ie. A dominates D and D post-dominates A), then their control predicates is represented by a unique PPG node.



For example if the CFG contains four basic blocks A, B, C, D and four edges (A, B), (A, C), (B, D), (C, D) then A dominates D, and D post-dominates A. This information is mapped on the partition graph by adding three PPG nodes p0, p1 and p2 and two oriented edges: (p0, p1) and (p0, p2).

3.3 Predicate Relation DataBase (PRDB)

PRDB stores information about

- subset relation
- disjoint relation

between predicates. Its implementation uses often bit sets (bit-vectors or bit-matrices) for memory efficiency and speed accesses.

A predicate relation database (PRDB) is constructed in ORC and is based on a per region basis: ie. the PPG and other structures are constructed on demand for a given region.

3.3.1 Regions

The notion of regions is described in [5] and is based on the CFG. Regions aim to structure the CFG under the form of a tree: nodes representing regions and edges representing the inclusion relation.

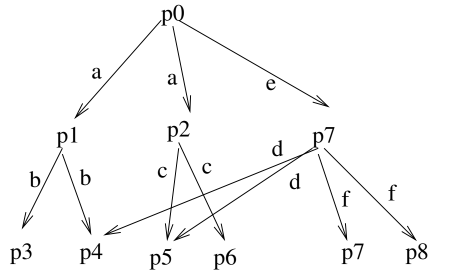
They define two kind of regions:

- MEME: Multiple Entries, Multiple Exits
- SEME: Single Entries, Single Exits

as basic entities on which analyzers and optimizers could work more efficiently.

3.3.2 Atomic Predicates (AP)

AP are defined in [1]. The basic idea is that they constitute the smaller entities in the predicates set: ie. a predicate is atomic iff the intersection with an other predicate is either empty, or itself. Thus, any predicate can be described as a disjunction of atomic predicates.



Predicates expressed
as a union of
atomic predicates :
Pred1 = a4 or a5
Pred2 = a1 or a2 or a5

Atomic Predicates = {p3, p4 inter p8, p5 inter p8, p4 inter p9, p5 inter p9, p6}

Atomic Predicate / PPG Node	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	Pred1	Pred2
a1				1								1
a2					1				1			1
a3						1			1			
a4					1					1	1	
a5						1				1	1	1
a6							1					

Possible implementation of Atomic predicates
as bit vectors.

This formalism provides a solid basis for predicate relations extraction. The PRE data-flow analysis uses a lattice based on sets of predicates described in [1]. The following algebraic specification gives more details on this data-flow analysis.

3.4 Algebraic Specification

This specification is *not* in OBJ syntax and is not intended to be executable. Its only purpose is to provide a support for algorithm description. The specification has been refined into an executable version written in C++ and was integrated in the ORC compiler.

Low level components of the system (modules SET, LIST, MAP, FUNCTION, BB) are not specified below since they were already implemented in ORC. Their interfaces were used during the refinement process.

```
Module SET
// Usual mathematical set module.

====
Module LIST
// Usual list module.

====
Module MAP<T1,T2>: imports SET, LIST
// Usual map module.

SIGNATURE:
op map:  T1   T2                       ->
op map_value: MAP<T1,T2>  T1           ->  T2
op map_elt:   MAP<T1,T2>  T1  T2       ->  MAP<T1,T2>
op map_list:  MAP<T1,T2>  list<T1>  T2  ->  MAP<T1,T2>

AXIOMS:
map_value (map (et1, et2), et1) = et2.

map_elt (m, et1, et2) = res
  res = m
  res.push_back (map (et1, et2))

map_list (m, empty, e) = m.

map_list (m, l, e) = res
  res = m
  map_list (map_elt (res, head (l), e), tail (l), e)
```

```
Module FUNCTION
 Questions:
   what is a function?
   how is it represented?
   what are its operations?

 Since this specif is not intended to be executable, it can be incomplete.
 This module have to describe what is a function.

 A very quick description (not yet refined) is:
 - representation:
   a sequence of statements,
   a map<stmt, bb>,
   ...
 - operations:
   foreach_op (op, function)
   foreach_bb (bb, function)
   ...

====
Module BB: imports STMT

 Classic view:
 A BB is a sequence of statements executed from the first to the last.
 (there's a single path of control in this sequence of operations:
 ie. a branch operation produces two new basic blocks)

 Predicated code view:
 A BB can contain operations from two different branches of a control
 operator, since each operation is guarded by an exclusive boolean register:
 ie. p1, p2 = cmp (cond) creates 2 branches, the first branch predicated
 by p1 and the second one by p2.  p1 and p2 are mutually exclusives since
 the result of the cmp operation is stored in a single boolean register.

 In this module again there is a decision in the next refinement step
 that consists in choosing one of the two possible views.

 I have to choose the second one: predicated code.
```

```
Module STMT

 Here it is possible to choose the level of the implementation in the compiler.
 A statement is an operation :
 either at a very high level (WHIRL)
 or at the machine level (CGIR)
 The implementation of this specification has to start by refinig this module:
 ie. make a choice between these two levels.

 I have to work at the very lower level (IA-64 instructions)
 and eliminate redundances at this level (CGIR). Thus the construction of
 the PPG has to deal with partitions created by instructions that create
 materialized predicates.

 The other solution is to deal with high level statements
 (could be WHIRL trees, or GCC trees, ...)
 This solution constructs a PPG based on the CFG that is different.
 Statements cannot create new partitions (unless the high-level language
 defines this kind of statements : bool1, bool2 = cmp (...))

 This module "parameterizes" the PRE module (as well as the BB's module).
```

```
// The following module is not fully specified.
// This module was written after its implementation
// for providing a global overview of the PPG construction.

Module APPG (partition graph with atomic predicates for an entire function):

SIGNATURE:
op APPG:  function  ->  APPG
op qualifying_predicate: OP  ->  PREDICATE

op ppg_nodes:  ->  APPG_NODE_SET
op Root_node:  -> APPG_NODE

op Build_APPG_nodes:  function  ->
op Link_APPG_nodes:  function  ->

op control_flow_equivalent_APPG_NODE:  BB  BB_SET  ->  APPG_NODE
op add_relations_due_to_control_predicates BB APPG_NODE  ->
op add_relations_due_to_materialized_predicates: BB APPG_NODE  ->

AXIOMS:

APPG (fun) =
  root_node = APPG_NODE
  ppg_nodes.push_back (root_node)
  Build_APPG_nodes (fun)
  Link_APPG_nodes (fun)
```

```
Build_APPG_nodes (fun) = res
  bb_map = MAP
  equal_classes = BB_SET
  // entry BB corresponds to Root_node in the APPG:
  // entry point is always predicated by true and is always executed.
  map_elt (bb_map, entry_basic_block, Root_node)

  // Step 1: construct APPG_NODEs for basic blocks.
  Calculate_Dominators ()
  foreach_bb (bb, fun)
    graph_node = control_flow_equivalent_APPG_NODE (bb, equal_classes)
    graph_node.Add_Related_BB (bb)
    map_elt (bb_map, bb, graph_node)
  end_for
  Free_Dominators ()

  // Step 2: construct APPG_NODEs for ops.
  foreach_bb (bb, fun)
    map_result_predicates (bb)
    map_operand_predicates (bb)
  end_for

control_flow_equivalent_APPG_NODE (bb, equal_classes) = res
  res = APPG_NODE
  FOR_ALL_BB_SET_members (equal_classes, ec)
    if (
        ((ec dominates bb) and (bb post_dominates ec))
        or
        ((bb dominates ec) and (ec post_dominates bb))
       )
      // ec is the equivalence class for bb.
      res = map_value (bb_map, ec)
    end_if
  end_for

  if (res.notInitialized)
    // there is no equivalence class for this bb.
    ppg_nodes.push_back (res)
    BB_SET_Union (equal_classes, bb)
  end_if
```

```
Link_APPG_nodes (fun) =
  foreach_bb_in_topological_order (bb, fun)
    bb_predicate = map_value (bb_map, bb)
    add_relations_due_to_control_predicates (bb, bb_predicate)
    add_relations_due_to_materialized_predicates (bb, bb_predicate)
  end_for

add_relations_due_to_control_predicates (bb, bb_predicate) =
  if (number_predecessors (bb) >= 2)
    children = get_bb_predecessors_predicates (bb)
    parent = map_value (bb_map, bb)
    Add_Partition (parent, children)
  end_if

  if (number_successors (bb) == 2)
    child1 = map_value (bb_map, bb_successor (bb, 1))
    child2 = map_value (bb_map, bb_successor (bb, 2))
    parent = map_value (bb_map, bb)
    Add_Partition (parent, child1, child2)
  end_if

add_relations_due_to_materialized_predicates (bb, bb_predicate) =
  FOR_ALL_BB_OPs (bb, op)
    if (op_defines_predicates (op))
      parent = qualifying_predicate (op)
      child1 = qualifying_predicate (result1 (op))
      child2 = qualifying_predicate (result2 (op))
      Add_Partition (parent, child1, child2)
    end_if
  end_for
```

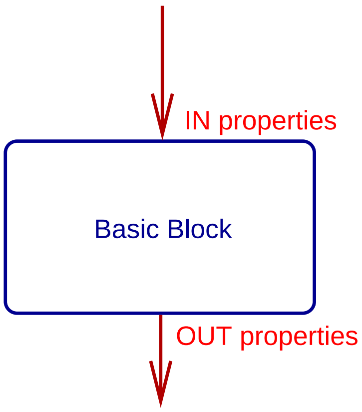
```
Add_Partition (parent, child1, child2) =
  Dp = parent.Decomposition
  foreach_atomic_predicate (a, Dp)
    copy_a = copy (a)
    remove parent from copy_a
    a1 = new_atomic_predicate (copy_a)
    a2 = new_atomic_predicate (copy_a)
    a1.push_back (child1)
    a2.push_back (child2)
    foreach_predicate (q, ppg_nodes)
      Dq = q.Decomposition
      if (Dq contains a)
        remove a from Dq
        if (q != child1)
          Dq.push_back (child2)
        end_if
        if (q != child2)
          Dq.push_back (child1)
        end_if
        q.Decomposition = Dq
      end_if
    end_for
  end_for
```

```
Module PREDICATE: imports SET, PPG, ATOMIC_PREDICATE

SIGNATURE:
op MEET:                set<predicate> set<predicate>  ->  set<predicate>
op predicate_UNION:     predicate predicate            ->  predicate
op contains:            predicate atomic_predicate     ->  BOOL
op intersects:          ppg_node predicate             ->  BOOL

AXIOMS:
MEET (A, B) = res
  res = empty
  foreach_predicate (p, A)
    foreach_predicate (q, B)
      res.push_back (predicate_UNION (p, q))
    end_for
  end_for

predicate_UNION (p, q) = res
  res = p
  foreach_atomic_predicate (a, q)
    if (not contains (res, a))
      res.push_back (a)
    end_if
  end_for

contains (p, a) = res
  res = false
  foreach_atomic_predicate (a1, p)
    if (a == a1)
      res = true
    end_if
  end_for

intersects (ps, p) = res
  res = false
  foreach_atomic_predicate (a, p)
    foreach_ppg_node (pn, a)
      if (pn == ps)
        res = true
      end_if
    end_for
  end_for
```

```
Module PRE: imports SET, LIST, MAP, PREDICATE, PPG, FUNCTION, BB,

SIGNATURE:

op inter_block_pre:    Function  ->  Function
op PRE_expression:  OP Function  ->  Function

op Earliest_OUT: BB OP  ->  BOOL
op Earliest_IN:  BB OP  ->  BOOL

op Ant_OUT:     BB OP  ->  BOOL
op Ant_IN:      BB OP  ->  BOOL

op Avail_OUT:   BB OP  ->  BOOL
op Avail_IN:    BB OP  ->  BOOL

// MFP =  Maximal Fixed Point
op ANT_OUT_MFP:       BB OP          ->  set<predicate>
op ANT_IN_MFP:        BB OP          ->  set<predicate>
op AVAIL_OUT_MFP:     BB OP          ->  set<predicate>
op AVAIL_IN_MFP:      BB OP          ->  set<predicate>

op ANT_MAP:                        ->  MAP<BB, set<predicate>>
op AVAIL_MAP:                      ->  MAP<BB, set<predicate>>
op ANT_MAP_OUT_MFP:   set<BB> OP  ->  MAP<BB, set<predicate>>
op AVAIL_MAP_IN_MFP:  set<BB> OP  ->  MAP<BB, set<predicate>>

// Recursive description (MOP = Meet Over all Paths)
op ANT_IN:      BB OP     ->  set<predicate>
op ANT_OUT:     BB OP     ->  set<predicate>
op ANT_OUT_rec: list<BB>  ->  set<predicate>

// Flow functions
op F_ANT:       BB OP set<predicate>  ->  set<predicate>
op F_AVAIL:     BB OP set<predicate>  ->  set<predicate>

op Add:         BB OP set<predicate>  ->  set<predicate>
op Modify:      BB OP set<predicate>  ->  set<predicate>
op Approximate: BB OP set<predicate>  ->  set<predicate>
op Approximate_predicate:  predicate OP ppg_node  ->  predicate
op Approximate_atomic_predicate: atomic_predicate OP ppg_node -> atomic_predicate
```

```
AXIOMS:

inter_block_pre (function) = res
  res = function
  foreach_op (op, res)
    if (is_op_good_for_PRE (op))
      res = PRE_expression (op, res)
  end_for

PRE_expression (e, function) = res
  res = function
  inserted = empty

  // Step1: Insert a BB containing a computation he = e (predicated by p0)
  //        at earliest point: be = new_bb ("he = e (p0)")
  be = new_bb (he, e, p0)
  foreach_bb (b, function)
    if (Earliest_OUT (b, e))
      insert_bb_after (be, b)
      inserted.push_back (be)
    end_if
  end_for

  // Step2: Replace every computation of the initial code 'e'
  //        with a reference to 'he'
  foreach_op (o, function)
    if (o == e)
      replace (o, he)
    end_if
  end_for

  // Step3: Eliminate bb introduced by Step1 for which
  //        Avail_IN (bb, e) = true.
  //        (these computations are redundant)
  foreach_elt (b, inserted)
    if (Avail_IN (b, e))
      cfg_erase (b)
    end_if
  end_for
```

```
Earliest_OUT (b, e) = and (Ant_OUT (b, e), not Ant_IN (b, e)).
Earliest_IN (b, e) = res
  and_rec = true
  foreach_predecessor (m, b)
    and_rec = and (and_rec, not Ant_OUT (m, e))
  end_for
  res = and (Ant_IN (b, e), and_rec)

Ant_OUT (b, e) = not is_empty (ANT_OUT_MFP (b, e)).
Ant_IN (b, e) = not is_empty (ANT_IN_MFP (b, e)).

Avail_OUT (b, e) = not is_empty (AVAIL_OUT_MFP (b, e)).
Avail_IN (b, e) = not is_empty (AVAIL_IN_MFP (b, e)).
```



BB #	Anticipability		Availability	
	IN	OUT	IN	OUT
1	T	F	F	F
2	...	...	...	...
3				
4				
...				

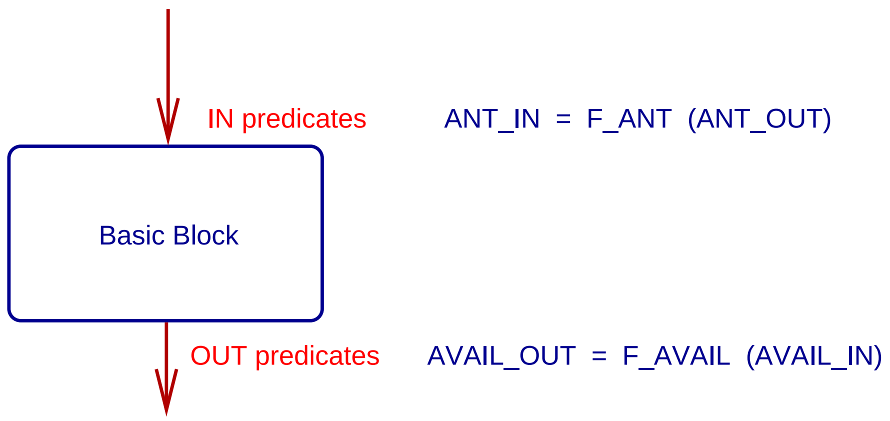
Anticipability and Availability properties are mapped on each BB.

```
// MFP =  Maximal Fixed Point
// Data Flow Analysis algorithm
// slightly adapted from Muchnick p.232

ANT_OUT_MFP (bb, e) = res
  if (empty (ANT_MAP))
    //bbs = set of all BB of the current function
    ANT_MAP = ANT_MAP_OUT_MFP (bbs, e)
  res = map_value (ANT_MAP, bb)

ANT_IN_MFP (bb, e) = F_ANT (bb, e, map_value (ANT_MAP, bb)).

ANT_MAP_OUT_MFP (nodes, e) = res
  map_list (res, nodes, empty)
  bb = exit_basic_block            // exit BB of this function
  worklist = empty
  worklist.push_back (predecessors (bb))

  while (not empty (worklist))
    bb = head (worklist)
    worklist = tail (worklist)

    totaleffect = empty
    foreach_successor (succ_bb, bb)
      effect = F_ANT (succ_bb, e, map_value (succ_bb))
      totaleffect = MEET (totaleffect, effect)
    end_for

    if (totaleffect != map_value (res, bb))
      map_elt (res, bb, totaleffect)
      worklist.push_back (bb)
    end_if

  end_while
```

```
AVAIL_IN_MFP (bb, e) = res
  if (empty (AVAIL_MAP))
    //bbs = set of all BB of the current function
    AVAIL_MAP = AVAIL_MAP_IN_MFP (bbs, e)
  res = map_value (AVAIL_MAP, bb)

AVAIL_OUT_MFP (bb, e) = F_AVAIL (bb, e, map_value (AVAIL_MAP, bb)).

AVAIL_MAP_IN_MFP (nodes, e) = res
  map_list (res, nodes, empty)
  bb = entry_basic_block          // entry BB in this function
  worklist = empty
  worklist.push_back (successors (bb))

  while (not empty (worklist))
    bb = head (worklist)
    worklist = tail (worklist)

    totaleffect = empty
    foreach_predecessor (pred_bb, bb)
      effect = F_AVAIL (pred_bb, e, map_value (pred_bb))
      totaleffect = MEET (totaleffect, effect)
    end_for

    if (totaleffect != map_value (res, bb))
      map_elt (res, bb, totaleffect)
      worklist.push_back (bb)
    end_if

  end_while
```

```
// Recursive description (MOP meet over all paths)
// Ivan Djelic showed that:
//     MFP solution == MOP solution
// Since the MFP converges to the right solution,
// it's enough to study the MFP property
// (simpler to compute, and avoids infinite recursion on CFG cycles).

ANT_OUT (exit_basic_block, e) = empty.
ANT_OUT (b, e) = ANT_OUT_rec (successors (b), e).
ANT_OUT_rec (succs, e) = res
  if (length (succs) == 1)
    res = ANT_IN (head (succs), e)
  else
    res = MEET (ANT_IN (head (succs), e),
                ANT_OUT_rec (tail (succs), e))
  end_if
ANT_IN (b, e) = F_ANT (b, e, ANT_OUT (b, e)).

AVAIL_IN (entry_basic_block, e) = empty.
AVAIL_IN (b, e) = AVAIL_IN_rec (predecessors (b), e).
AVAIL_IN_rec (preds, e) = res
  if (length (preds) == 1)
    res = AVAIL_IN (head (preds), e)
  else
    res = MEET (AVAIL_OUT (head (preds), e),
                AVAIL_IN_rec (tail (preds), e))
  end_if
AVAIL_OUT (b, e) = F_AVAIL (b, e, AVAIL_IN (b, e)).
```

```
// Transfert functions
F_ANT (b, e, set_pr) = res
  res = set_pr
  res = Modify (b, e, res)
  res = Add (b, e, res)
  res = Approximate (b, e, res)

F_AVAIL (b, e, set_pr) = res
  res = set_pr
  res = Add (b, e, res)
  res = Modify (b, e, res)
  res = Approximate (b, e, res)
```



Effects of a BB on predicates
Transfer functions evaluate these effects on a BB crossing

```
Add (b, e, A) = res
  res = A
  foreach_statement (s, b)
    ps = qualifying_predicate (s)
    if (s == e)
      res.push_back (ps)
    end_if
  end_for

Modify (b, e, A) = res
  res = A
  foreach_statement (s, b)
    ps = qualifying_predicate (s)
    if (s.modifies_arguments (e))
      res = empty
      foreach_predicate (p, A)
        if (not intersects (ps, p))
          res.push_back (a)
        end_if
      end_for
    end_if
  end_for

Approximate (b, e, A) = res
  res = A
  foreach_statement (s, b)
    ps = qualifying_predicate (s)
    if (s.defines_predicates ())
      res = empty
      foreach_predicate (p, A)
        res.push_back (Approximate_predicate (p, e, ps))
      end_for
    end_if
  end_for
```

```
Approximate_predicate (p, e, ps) = res
  res = empty
  foreach_atomic_predicate (a, p)
    res.push_back (Approximate_atomic_predicate (a, e, ps))
  end_for

Approximate_atomic_predicate (a, e, ps) = res
  res = empty
  //e is a comparison instruction under the form:  " pi,pj = cmpp (...)"
  pi, pj = predicates (e)
  foreach_ppg_node (pn, a)
    if (pn == pi)
      res.push_back (ps)
    else if (pn == pj)
      res.push_back (ps)
    else
      res.push_back (pn)
    end_if
  end_for
```

3.5 Some notes on refined versions

In FUNCTION module, code is supposed to be in *if-converted form.*

Note also that the previous specification inherits BB = OP from the high level description of the PRE algorithm in [1].

3.6 Implementation methodology: from algebraic specification to a tuned executable version.

The specification process begins with the overall description of the system and continues on several steps of refinement. These refinements aims to give an efficient executable version for the specified system.

The role of the specifier is to produce a description of the system at the highest abstraction level. This specification is then refined (by including language specificities, memory management, ...) into an executable version of the specification. A next step of refinement handles time and space issues of the executable implementation. The final goal is the production of an efficient implementation of the system.

A system can be built in a three step process.

- Specifying at a very high level is a top-down process. This specification intends to separate domain specific difficulties from system integration difficulties. Domain difficulties are solved by the specification process, which produces an algorithmic description of the system.

- Specification refinement and integration in an existing system is a bottom-up process. System integration solves all low level issues where the specification remains incomplete, or even inconsistent with the existent system. System integration starts with the lower levels of the specification and continues until the top level of the specification is reached, that is when the system is completely implemented.

- refine the implementation for performance improvements (tuning the system)

The main advantages in using this three step development model are:

- The specification language is a communication tool. It can be used in a relatively free syntax in order to avoid specification difficulties. A strict syntax allows the specifier to check the consistency of his work (for example one could use OCaml as a functional language specification tool, or CafeOBJ for algebraic specifications).

- The specification dissociates domain specific difficulties from system integration

- This development model allows planification on tight schedules.

- This model is close to the prototype-driven development: it is based on an incremental development and refinement of the working version

A drawback of this development model is that intermediate specifications of the same problem have to be kept up to date (as a compiler would have done on its intermediate representation after an optimization or lowering pass). However this is a quite difficult constraint when working on a large project. Thus, intermediate specifications are often left under an old version and not updated: only the end version is further refined making old versions obsolete.

System tuning for better performance is an optional pass that could be left to an optimizer compiler. However sometimes it is interesting to have better optimizations specific to the current system, and that are not provided by the compiler. As an example, the matrix multiplication on general matrices can be suboptimal for sparse matrices, and thus libraries containing different refinements for the same set of operations are provided. These types of optimizations, that occurs relatively often, are little by little included as automatic optimizations in compilers allowing the programmer to concentrate on more abstract levels than memory structural representations.

3.7 Results

There was not enough time to run performance tests on large-scale benchmarks, this is left for future work by Ivan Djelic and Albert Cohen...

4 Conclusion

In April 2002 I met the A3 team after a presentation of the GCC project by Gabriel Dos Reis. Two months later I accepted the internship proposed by Albert Cohen for working on the Open64/ORC compiler.

During this internship I discovered the structure of SGI's compiler. Its architecture was surprisingly close to GCC's; my previous experience on GCC helped me to explore Open64 compiler. Source exploration tools as xemacs, gdb, etags and oo-browser had a major effect on my working effectiveness.

Open64 provides an amazing collection of high level optimizations (principally in the LNO).

Thus I have to thank Albert Cohen for having proposed such an interesting internship subject and for his help for solving difficult problems. I have also to thank A3 team for the excellent working conditions.

A Debugging Open64 compiler

Here is how I debug Open64 optimizations:

1. Compile Open64 with the -g option in order to dump all the debugging symbols.

2. execute gdb on the following executable:
   /.../usr/lib/gcc-lib/ia64-orc-linux/orcc-1.0/be

3. set some environement variables:

```
set environment LD_LIBRARY_PATH=.../usr/lib/gcc-lib/ia64-orc-linux/orcc-1.0/
set environment DAVINCIHOME=.../daVinci/
set environment PATH=.../daVinci/
cd /working_dir/
```

4. set a breakpoint in the main driver just *after* the module loader have done his work. At this point all the symbol tables were loaded into the memory and thus you can set breakpoints in the dynamic shared objects. Then tel gdb to continue silently...

```
break driver.cxx:1788
commands
 silent
 break inter_block_pre.cxx:400
 break cg.cxx:742
 break ...
 ...
 continue
end
```

5. Finally run the program (with the right options). You could run the compiler once with the wrapper and tel it to be verbose... It will dump the right line with which you can execute the back-end (be).

```
run -PHASE:l:w:c -G8 -TENV:PIC -m1 -INTERNAL:return_val=on
-INTERNAL:mldid_mstid=on -INTERNAL:return_info=on -show -O3
-PHASE:c=on:w=on:l=on:p=on -TARG:abi=i64 -LANG:=ansi_c two.c
```

   -PHASE option selects the phases to be runned : here l, w, and c (ie. LNO, WOPT and CG).

I have to write a short note about a memory problem under RedHat Linux 7.1 and gdb (in fact I was not able to detect the origin of this problem, maybe the machine on which I work has not enough memory). When you call a function in gdb with *call debug ()*, the debugger begins to allocate memory and stops with a sig once it filled all the available memory. When calling some small functions as fprintf, it works the first time, but the second call is always fatal. A possible workaround to this problem is to modify the source code by inserting the call at the right place, then recompile the concerned module. This makes the debugging a real nightmare, but anyway it works . . .

B Documentation

You can download an archive of the web page [6] containing :

- Open64 compiler documentation comming from source code;
- this report;
- slides written in *Prosper*;
- figures drawn in *xfig*.

You are *allowed* to copy them and modify them as you want.

References

[1] I. Djelic, Élimination de redondances partielles pour architectures EPIC, TSI, 2002

[2] S. Muchnick, Compiler Design and Implementation, Morgan Kaufmann Publishers

[3] J.R.Allen, K.Kenedy, C.Porterfield, and J.Waren. Conversion of control dependence to data dependence. In proceedings of the 10th ACM Symposium on POPL.

[4] E. Visser, A survey of Rewriting Strategies in Program Transformation Systems, http://www.stratego-language.org

[5] L. Yang, Z. Zhaoqing, Q. Ruliang, A Region-Based Compilation Infrastructure

[6] http://www-rocq.inria.fr/~pop/