Control sensitive global instruction selection

François Gindraud
ENS Lyon

Florian Brandner
Compsys, LIP

Alain Darte
Compsys, LIP

Contents

1 Introduction

The compilation process of a program goes through several steps. We first read the source code, then we build a syntax tree by parsing the code. This tree is called an *intermediate representation*, as it is not source code neither output code, but holds all informations about the program. Then we apply several transformations to adapt source code structures to output code, to optimise the output code, and so on. Some representations are better suited to do some specific optimisations, so it is not uncommon to translate the program to one intermediate representation to another during the compilation (each new representation is closer to the shape of the output code). And finally we use the *instruction selection* process to parse the representation and select output code instructions which will make the output program.

The initial goal of this internship was to study and understand several intermediate representations, and then see which one was best suited to extend the instruction selection method described in [5]. Ideally, the last intermediate representation should provide maximum informations for each operator, like how are its operands defined, where is the result used, in which conditions will the operator be executed, and so on. The more information we have, the more precisely we can select instructions.

These informations are often represented as dependencies. If there is a dependency between two instructions, like the second needs the result of the first, it will impose an order of execution. There are two types of dependencies. The first, and the most studied type is *data dependencies*, it is illustrated by the previous example. The second type is *control dependencies*, which is when the execution of an instruction is dependent on a result of another instruction. For example, in a `if-then-else` structure instructions are control dependent on the value of the predicate of the `if` statement.

These two types are at the same time different and similar. There are similar because in fact control dependencies are data dependencies. Processors only manipulate data, and use data (program counter) to handle control dependencies. But they are slightly different when we want to apply optimisation algorithms, which operates on higher level of code than automata (which is what processors are).

So the goal of the internship was to find representations which are able to hold control dependencies. It would after be used to extend an instruction selection algorithm to take care of control dependencies. This would have enabled to select instructions like `cmov` for x86 architectures with a generalised method.

I have made my internship in the CompSys (Compilation and Embedded Computing Systems), under the supervision of Alain Darte and Florian Brandner. CompSys is specialised in optimisations for embedded systems, which often feature very specific processors. So we need to create more efficient algorithm which will be able to use the specificities of these processors to ensure high performances. One important part of this is the need for an instruction selection algorithm able to use very specific instructions. Florian Brandner had co-created an efficient an flexible algorithm [5], so we wanted to improve it with control dependencies.

So I started my internship by reading articles provided by Florian to study the representations. The bibliography given by Florian was very complete, and I only needed to add one article. At first I had some problems because I did not know which representation I should study first. I started by reading shortly a large set of representations to intuitively see differences and similarities. It took me more time than planned to read and understand articles, so after some time we decided to focus on few representations, and stay in acyclic conditions. I studied them deeply, understanding how they where built and how to translate out of it.

Finally, it was obvious that it took me too much time to understand the representations, and I would not have enough time to adapt Florian's algorithm. We decided to reduce the expectations of the internship to the comparison of representations.

Then, after I studied a structure named Ψ-SSA, I noticed one structure used to optimise it was really interesting. It was named *Predicate Partition Graph*. However, the paper in which it was defined was not very precise, and we decided to improve the algorithm when we found another way to build the PPG. My work was then to formalize it properly, along with the notions around the PPG, which took some time.

This report is focused on the results we have obtained, mostly around the PPG. The first part is an overview of all the structures I have studied. I will focus on giving the general idea of the representation rather than give too much details. The second part give my formalization of the base representations we were relying on to build the PPG. The third part contains much of my work, which was to formalize properly the notions around the PPG and the PPG itself. Then I remind the original algorithm, and

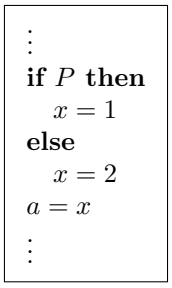

Figure 1: Plain code

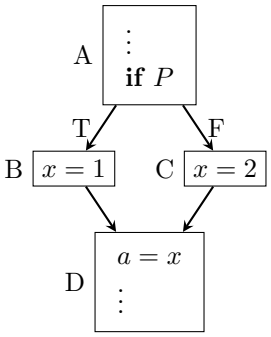

Figure 2: CFG

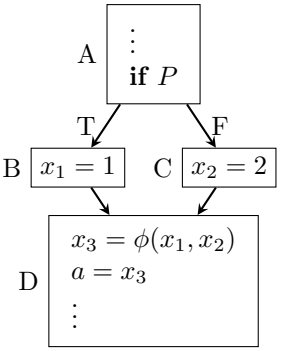

Figure 3: CFG in SSA form

give some justifications about correctness. After that I give our new algorithm, along with justifications of correctness, and its adaptation to handle loops. The last part gives some short explanations on the uses of the PPG and comparison with others representations.

2 Intermediate Representations

We will describe in this section the different intermediate representations we have studied in the internship. We will only describe them quickly, by using examples and giving the main ideas of the representation. To illustrate representations, we will start with the simple code in figure 1, and use the construction algorithms to build representations of this code.

The example is very simple, and there is no loops. This is mainly because we only want to present briefly the representations, and also because our algorithms operate in acyclic conditions (with pre treatment of loops).

2.1 Control Flow Graph

The *Control Flow Graph* is one of the simplest intermediate representations. It is obtained by isolating *basic blocks*, which are portion of code between two labels or jumps. So the portion of code inside a basic block will always be executed completely, or not at all (the code is non interruptible).

Then the basic blocks are named, and we make a graph whose nodes are basic blocks, and edges represents the target of jump instructions. Each edge starts at the node containing the jump instruction, and goes to the node containing the target label. If the jump is conditional, as in a `if-then-else` structure, we put one edge per case of the condition. And we label these edges by the value of the condition (true, false, ...). A node with multiple out edges is called a branch, and with multiple in edges it is a merge.

We give an example of CFG in figure 2. The plain code has been split in 4 basic blocks. The A block ends with a conditional jump, so we label the out going edges by true (T) and false (F). The code of the two cases of the `if-then-else` lies in B and C, which end by a jump to D to resume the execution of the program.

The CFG is a very simple intermediate representation. The main interesting feature is that it is a base for almost all algorithms that build others representations. It can be used to represent almost all program using the control flow paradigm (opposed to the functional paradigm).

We will describe it more precisely in section 3.2.

2.2 Static Single Assignment form

The *Static Single Assignment* (SSA) form of a program is a program where each variable is statically defined only once. It means that there is only one definition of a given variable in the code (but this definition can be executed multiple times if it is in a loop).

If we want to transform our program in SSA form, the main idea is to rename variables each time we encounter a definition (nodes B and C of figure 3). Each time we find a use of the variable, we replace

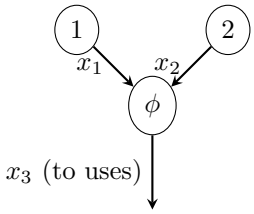

Figure 4: SSA-Graph

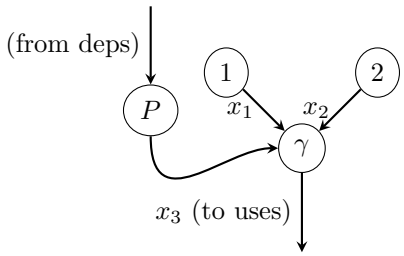

Figure 5: Gated-SSA

it with the last name we used for it. If there is multiple names reaching the node (in a merge node for example), we use a selector function named *ϕ-function*, like in D. You can find a more precise algorithm in [3]. A main problem with these selector functions is the need for a elimination step to remove them before generating actual code, which is tricky. The first algorithms for out-of-SSA had bugs, and it led to the disabling of many SSA-related optimisations in production compilers.

SSA allows us to get definition-use links, i.e. data flow links, which are difficult to extract directly from the code. This simplifies many optimisations. However as this internship is focused on the control flow representation we will mostly look at adaptations of SSA to integrate control dependencies.

2.3 SSA-Graph

A SSA-Graph is a representation which only holds data dependencies. It is a graph whose nodes are operations, and edges are data dependencies between these operations. In fact, as data dependencies means data moved through variables from one operation to another, edges represents variables of the program in SSA-form.

It can be built from a SSA-form CFG by adding an edge between two operations for each definition-use link. We give an example in figure 4, which represents variable x. A description and examples can be found in this article [5].

As we can see, there is no control flow represented in the SSA-Graph, so we cannot choose between the two values at the ϕ node. This is why the SSA-Graph is always given with the corresponding CFG, which holds the control dependencies. This feature of SSA-Graphs can be useful to speed up some optimisations that does not rely on control flow, but is costly when we need it.

2.4 Gated-SSA

Gated Static Single Assignment (Gated-SSA) was first introduced as a part of the Program Dependence Web (PDW) in [8]. However, I mostly used this article [7] from Paul Havlak to define and study Gated-SSA, because it was focused on Gated-SSA rather than the PDW.

Gated-SSA is an extension of SSA-Graphs, which annotates ϕ nodes to add control flow information in the SSA-Graph. An example is given in figure 5, where the ϕ node is replaced by a γ node. The γ node is also control dependent on P, a bloc representing the computing of the predicate at runtime. It itself has data dependencies on values used in the computing. There are also μ and η nodes for loops.

Gated-SSA succeeded at adding control flow information in SSA-Graph. However there are some problems, like adding new specific nodes in the SSA-Graph which must be removed in a post-processing step. Optimisation algorithm can easily break the structure formed by these nodes. Another problem is the locality of the control flow information: we only know which predicates chose the value of the ϕ. This is not the best information for code motion for example.

2.5 Program Dependence Web

The *Program Dependence Web* (PDW) was first defined in [8]. The article [1], though it does not give any construction algorithm, explains the roles of the specific nodes in a more synthetic way, and give some examples.

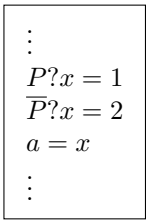

Figure 6: Predicated code

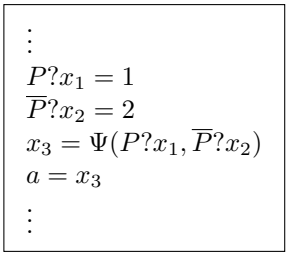

Figure 7: Ψ-SSA

Basically, the PDW is a Gated-SSA graph augmented with new specific nodes. These nodes allow interpretation of the PDW in different paradigms, such as imperative, functional of data-flow. However, as we are in an imperative point of view, the full PDW is not needed, and we will keep only Gated-SSA nodes.

2.6 Predicated code

Some processor architectures support *predicated code*. A predicated instruction is like a standard instruction, but its execution is dependent on a value of a register. If the register is true, then the instruction will execute normally, and if it is false it does nothing. In the example of figure 6, P symbolize the predicate register, and $P?instr$ means that $instr$ is predicated by the value of P.

An instruction set of a processor can support various levels of predication. Some processors will only support predicated jumps (jumps which depend on a value of a register), which is needed if we want a useful processor. Some processors will have a fully predicated instruction set, which means that even math operators are predicated. For example, the x86 instruction set supports conditional jumps and conditional moves, but is not fully predicated.

Predication is a mean to represent control flow which is not directly convertible into a CFG, so specific methods and intermediate representations must be made. An important remark is that we cannot represent loops with predication only (we need jumps). So predicated code is often studied within a *hyper-block*, which is a block of predicated code with no jump or incoming labels.

As predication is a way to optimise some programs, we sometimes want to predicate some parts of standard (CFG) programs. This is called *if-conversion*, which consists of predicating the `then` part if `if-then-else` with the predicate, and the `else` part with the opposite. Figure 6 is the if-converted version of figure 1. This method is also limited to acyclic parts.

I have studied two predicated representations, which will be described below.

2.6.1 Ψ-SSA

Ψ-SSA is an intermediate representation first introduced by *ST Microelectronics*, in [9, 4]. It introduce Ψ-functions, which are like ϕ-functions but for predicated assignments, and has a renaming step like in SSA.

The aim of this intermediate representation was to extend SSA to predicated code. So it is very similar to SSA. Predicated assignments are renamed, and Ψ-functions are added when a use is reached by multiple definitions. Unlike SSA, where there are no need for ϕ-functions in linear code, multiple paths are merged into a linear code with the predication. So after two assignments we need a Ψ-function. The example of figure 7 show us the code in Ψ-SSA. Semantically, the result of a Ψ-function the value of the first variable with an associated predicate true, from right to left.

This structure also need a post-processing step to remove the Ψ and the ϕ. Many optimisations presented in the papers used a convenient structure, called *Predicate Partition Graph*. It gives us relations between predicates, so we can for example eliminate unnecessary predicates in Ψ. We can transform $\Psi(p?a, q?b, \overline{q}?c)$ into $\Psi(q?b, \overline{q}?c)$ because we know that q and $\overline{q}$ cover all cases, and $p?a$ will never be chosen.

2.6.2 Predicated-SSA

Predicated-SSA is described in [2]. Unlike Ψ-SSA, it has no specific node which must be removed before leaving the representation. Its principle is to use as predicate for an instruction in a hyper-block a predicate representing the whole path from the start of the block. It is focused on scheduling purpose, and generate a large amount of code when the hyper-block grows.

2.7 Predicate Partition Graph

The *Predicate Partition Graph* (PPG) represent links between predicates of a program. I first saw this structure as a convenience tool for optimising Ψ-SSA, and it is only used as a tool at ST. But as I progressed into the internship, the PPG was getting more and more interesting. It is defined in [6], but not explained very formally, and with an imprecise building algorithm.

The name may suggests that it is limited to predicated code, but the PPG can represent at the same level control flow from predicated code and control flow from CFG structure. It is an improvement over the CFG, as its construction also does some optimisations, like merging equivalent nodes. Unlike Gated-SSA, it is not limited to local informations. However, it only holds control flow information, so we need to hold data-flow information elsewhere (like a SSA-Graph).

In this internship, I have made together with Florian Brandner a more precise algorithm for constructing the PPG. I have formalized the property on which is based the PPG. It will be described in section 4. An example can be found in figure 11.

3 Background

In this section we will give some formal background on the conventional intermediate representations we have used to build the PPG in this internship. It will include the notations we will use, a description of the Control Flow Graph, and a short description of the Control Dependence Graph.

3.1 Set & graph notations

Partition Set are defined as usual. We will just define a partition property, as it will be very useful later in the Predicate Partition Graph.

Definition 1. We define the partition relation for $A, B_1, \ldots, B_n$ sets :

$$A = B_1 \mid \ldots \mid B_n \Leftrightarrow \begin{cases} \bigcup_{i=1}^{n} B_i = A \\ \forall (i,j), B_i \cap B_j = \varnothing \end{cases} \tag{1}$$

Graph We define graphs as usual, i.e. an oriented graph is a pair $G = (V, E)$ with V the set of nodes, and $E \subseteq V^2$ the set of edges. We will also use the notion of hyper-graph and hyper-edges. A *hyper-graph* is also a pair $G = (V, E)$, but with $E \subseteq \mathbb{P}(V)^2$. A *hyper-edge* is an edge from a set of nodes to another set of nodes. In the following, we will only use *half hyper-edges*, edges from a node to a set of nodes.

We will use the notation $a \to b$ to represent the existence of an edge. Saying $a \to b$ is equivalent to say that $(a, b) \in E$. Edges will be annotated with the graph they come from when it is ambiguous, like when we have multiple graph representations. For example, $\to_{CFG}$ or $\to_{PPG}$.

Path *Paths* are defined on graphs as usual: a path is a sequence of nodes which have edges between each node and its successor in the graph. The notation $a \xrightarrow{\star} b$ means that there exists a path between a and b in the graph, and represents the set of nodes in the path.

We also remind that a *simple path* is a path were each node can appear at most one time. Is it equivalent to say that it is a path with no loops.

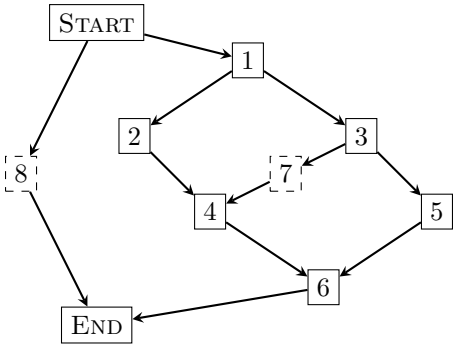

Figure 8: Example of unstructured acyclic Control Flow Graph

Notation We will define convenient notations for graph elements around a node.

Definition 2. Let $v \in V$ be a node. Then we define the sets of incoming and out going edges for v:

$$InEdge(v) = \{(w, v) \in E / w \in V\} \tag{2}$$
$$OutEdge(v) = \{(v, w) \in E / w \in V\} \tag{3}$$

And we define the set of predecessors and successors nodes for v:

$$InNode(v) = \{w \in V / (w, v) \in E\} \tag{4}$$
$$OutNode(v) = \{w \in V / (v, w) \in E\} \tag{5}$$

3.2 Control Flow Graph

The *Control Flow Graph* (CFG) was briefly described in section 2.1. Here we will define it more formally.

Definition 3. A *Control Flow Graph* is an oriented graph $G = (V, E)$. It has two special nodes named START and END, which are respectively the entry point and the end point of the program. So START has no in-edges, and END no out-edges.

START has an edge to another basic block, representing the first code which will be executed. The execution will go through edges and blocks until it reaches END. It then stops. For convenience when building the Control Dependence Graph, we suppose there is an edge from START to END.

Although the brief description in section 2.1 mentioned that out-edges of a branch have a label, we will forget about it in the following sections. The reason is that we will care about static paths in the CFG rather than real executions which need to know which edge will be taken.

Forgetting labels could be troublesome if there was multiple edges between the same nodes with different labels (they would be merged). But we will very often remove these troublesome cases by splitting critical edges, which will be described in the next section (3.2.1). An example of CFG with split critical edges is in figure 8.

3.2.1 Critical edges

Many algorithm and properties use a notion of path on the CFG. In the most general case paths should be represented as the sequence of edges, because for example we can have multiple edges (with different labels) between two nodes. However, we often want to represent paths as a sequence of nodes, which is ambiguous in this case.

A possible solution is to consider edges as *virtual nodes*, which is equivalent to add nodes on each edge. But this solution is very costly as the number of nodes grows quadratically.

In fact, only ambiguous edges need to be split by a virtual node. We name *critical edges* edges which start at a branch and end at a merge, i.e. edges (v, w) with $|OutEdge(v)| > 1$ and $|InEdge(w)| > 1$.

By splitting these edges, i.e. removing the edge (v, w) and adding a new nodes x, with new edges (v, x) and (x, w), we can ensure the CFG has now the given property :

$$\forall (v, w, l) \in E, |OutEdge(v)| = 1 \vee |InEdge(w)| = 1$$

And with this property, we can easily prove that any edge has an equivalent node associated with it, which means that a path taking this edge will also pass through the node, and reciprocally. An example of critical edges splitting is in figure 8, dashed nodes are virtual nodes added to split critical edges.

3.2.2 Dominance properties

The dominance properties are well described here [3]. So I will mostly explain them quickly and give some properties. We start with the dominance property:

Definition 4. Let a, b be two CFG nodes. We say that a *dominates* b when a is in every path from START to b. It means that a will always be executed before b is. We will note it $a \geq_{Dom} b$. When a dominates b but $a \neq b$, we say that a *strictly dominates* b ($a >_{Dom} b$).

We then define the *immediate dominator* of a, noted $Idom(a)$, as the closest strict dominator on every path from START to END. We can then define the *dominance tree* as the tree formed with nodes of the CFG, and in which START is the root, and each node has its immediate dominator as parent. The immediate dominator of START is itself by convention.

We also can define a symmetric property, called *post-dominance*:

Definition 5. A node a *post-dominates* a node b if a is in every path from b to END. We note it $a \geq_{Pdom} b$, and will define in the same way as for dominance the *strict post-dominance* ($a >_{Pdom} b$), the *immediate post-dominator* ($Ipdom(a)$), and the *post-dominance tree*.

Property 1. Dominance and post-dominance are reflexive and transitive. We also have that is $a \geq_{Dom} b$, then a will dominate all nodes in a path from a to b. We will also define the dominance of a set of nodes, $\{a_1, \ldots, a_n\} \geq_{Dom} b$, such as for each path from START to b there is an a_i in the path. And there is the same for post-dominance.

There are also the dominance frontier properties, introduced in [3] to build the Control Dependence Graph and place ϕ-functions of SSA:

Definition 6. A node a is in the *dominance frontier* (noted $DF(b)$) of a node b if there is an edge $c \to a$ such that $b \geq_{Dom} c$ but $b \not\geq_{Dom} a$. Intuitively, the nodes in the dominance frontier are nodes were paths passing through the node merge with other paths.

By symmetry, we also have a notion of post-dominance frontier:

Definition 7. A node a is in the *post-dominance frontier* (noted $PDF(b)$) of a node b if there is an edge $a \to c$ such that $b \geq_{Pdom} c$ but $b \not\geq_{Pdom} a$. Intuitively, the nodes in the post-dominance frontier are branch which choose whether b should be executed or not.

I have added the notion of *post-dominance frontier successors*, as it was used in the PPG construction. A node c is in $PDF_{succ}(b)$ if there is an edge $a \to c$ such that $b \geq_{Pdom} c$ but $b \not\geq_{Pdom} a$. These nodes are the first nodes post-dominated by b in dominance order.

We will also add a formal definition for the equivalence of two nodes, which lacks in [6].

Definition 8. Two CFG nodes a and b are equivalent, noted $a \sim_{CFG} b$, if:

$$(a \geq_{Dom} b \wedge b \geq_{Pdom} a) \vee (b \geq_{Dom} a \wedge a \geq_{Pdom} b)$$

It means that they are equivalent from the control flow point of view. Each time one of the two nodes is executed, the other will too. So for instructions in these two nodes, the only placement constraint comes from data dependencies.

3.2.3 Structural properties

We will now see some common properties of a CFG.

Definition 9. A CFG is *reachable* is for each node $v \in V$, there is a path $t = \text{START} \xrightarrow{\star} \text{END}$ containing v.

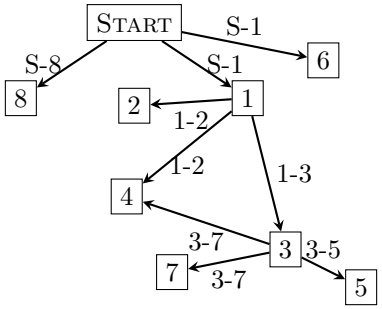

Figure 9: Control Dependence Graph built from CFG in figure 8

We will very often suppose that a CFG is reachable. If it is not, there are nodes which cannot be reached from START, and will never be executed, or there are nodes which cannot reach END which means that they have no out-edges. These two cases represents bugged or unnecessary code and will be ignored.

Definition 10. A CFG is *structured* if it has the fork-join property, which is that each time there is a branch, the out-going paths remain disjoint until they merge at the immediate post-dominator of the branch. A structured CFG is intuitively what we obtain with only `if-then-else` and no goto.

Definition 11. We name *loop header* a node in a strongly connected region which is the only one not dominated by the others. Intuitively this node will be the only entry point of the region (which symbolize a loop). Then we name *back-edges* edges from nodes of the region to the loop header. A CFG is said to be *reducible* when each strongly connected has a loop header, and this property is always valid when we iteratively remove back-edges.

Figure 12 gives an example of reducible CFG, with loop-header colored in blue.

3.3 Control dependence graph

The *Control Dependence Graph* is an alternative structure presented in [3] to represent control dependences. The article gives some formalism and a construction algorithm.

We say that a node b is control dependent on a if there is a path from a to b such that b post-dominates every node of the path except a. The article proves that is it equivalent to $a \in PDF(b)$, and then propose an algorithm based on dominance frontier for construction.

Then the CDG is built by taking the nodes of the CDG, and by adding an edge from node a to b only if b is control dependent on a. In the paper, these edges are labeled by the label of the out-edge of a that is on the path $a \xrightarrow{\star} b$ in the CFG. It is possible because if a node is in the post-dominance frontier, it is a branch and there are labels on out-edges. However in our construction of the PPG we forget labels on the CFG, so the CDG will take as labels the name of the edge in the CFG.

As critical edges have been split, each branch has at least one control dependent node per out-edge (the first node on the out-edge). So we have the property that CDG nodes have 0 or more than 1 out-edges, because if the node is a branch it has at least 2 out-edges, and if it is not a branch it will not have any out-edges. An example of CDG can be found in figure 9.

4 Predicate Partition Graph

This part will describe the *Predicate Partition Graph* (PPG). This is not an intermediate representation, as it only represents control dependences. This part will define the notion of traces, which are used to derive partition relations between nodes, and then define the PPG. Then it will describe PPG construction using two methods, one described in the original paper [6] and one new using the CDG as a base. First we will assume acyclic control flow graphs, and after we will extend it to reducible CFG.

4.1 Construction for Acyclic Flow Graphs

In this part we will suppose there is a reachable CFG $G = (V, E)$, where all critical edges have been split before.

Traces & Predicates

Definition 12. We name a *trace* a sequence of CFG nodes $t = (v_1 = \text{START}, \ldots, v_n = \text{END})$ where a simple path from START to END exists in the CFG that covers all nodes of the trace, i.e., $v_1 = \text{START} \rightarrow v_2 \rightarrow \ldots \rightarrow v_n = \text{END}$ is a valid path in the CFG.

A trace can be seen as the set of CFG nodes visited during an execution of the program. However, a trace is not always a valid execution of the program, because it ignores the predicates of branches.

As discussed in the critical edges part, the splitting of these edges guarantees that a trace represents only one path in the CFG.

Definition 13. We also define $AllTraces$ as the set of all possible traces in the CFG.

With the definition of traces we can define a notion of predicate, which is a symbol used to represent traces passing through the corresponding node.

Definition 14. A *symbolic predicate* p_v is a symbol linked to a node v of the CFG, representing all traces containing the respective node. The *domain* of a symbolic predicate gives all those traces, more formally:

$$Domain(p_v) = \{t \mid t = (v_1, \ldots, v, \ldots, v_n) \in AllTraces\}$$

Domains are used to represents which traces are linked to which nodes. Then, we can determinate relations between nodes by comparing the domains of the predicates associated to these nodes.

CFG-related properties We will now translate useful properties on the CFG to predicate domain relations. We will particularly focus on dominance and dominance frontier properties which are very important for the control flow.

Property 2. Let u, v be two CFG nodes. Then we have:

$$u \geq_{Dom} v \Rightarrow Domain(p_v) \subseteq Domain(p_u) \tag{6}$$

$$u \geq_{Pdom} v \Rightarrow Domain(p_v) \subseteq Domain(p_u) \tag{7}$$

Proof. We suppose $u \geq_{Dom} v$. For each trace $t \in Domain(p_v)$, we have $u \in t$ because t contains a sub-path from START to v in the CFG and $u \geq_{Dom} v$. So $t \in Domain(p_u)$.

Now we suppose $u \geq_{Pdom} v$. For each trace $t \in Domain(p_v)$, we have $u \in t$ because t contains a sub-path from v to END in the CFG and $u \geq_{Pdom} v$. So $t \in Domain(p_u)$. □

We can see that these properties are symmetric ; dominance becomes post-dominance if we reverse all edges in the CFG and swap START and END nodes. The same goes for post-dominance.

We also see that the inclusion between domains does not hold the difference between dominance and post-dominance. This is interesting, because the symmetry between these properties makes them very similar, and here there are merged.

In fact, the inclusion between domains exactly means that we have dominance or post-dominance between the nodes:

Property 3. Let $u, v \in V$:

$$Domain(p_v) \subseteq Domain(p_u) \Leftrightarrow u \geq_{Pdom} v \vee u \geq_{Dom} v \tag{8}$$

Proof. ($\Rightarrow$)We suppose that $u \not\geq_{Pdom} v \wedge u \not\geq_{Dom} v$. Then there is a path from START to END which contains v but does not contains u. So we can build a trace t from this path in $Domain(p_v)$ but not in $Domain(p_u)$.

($\Leftarrow$)Trivial by using the 2 previous properties. □

It follow that equivalent nodes have equals domains:

Property 4. Let $u, v \in V$:

$$u \sim_{CFG} v \Leftrightarrow Domain(p_u) = Domain(p_v) \tag{9}$$

The proof is just the application of the previous property. It allows merging of predicates of two equivalent nodes, which is a good property for the PPG.

Then we will consider relations derived from branch and merge nodes and their neighbours in the CFG. This property will be used to justify the first algorithm for PPG construction, which derive partition relations from merges and branches.

Property 5. Let $u \in V$ be a node. If u is a branch node, then we have a partition relation between it and its successors:

$$Domain(u) = \bigsqcup_{v \in OutNode_{CFG}(u)} Domain(v) \tag{10}$$

If u is a merge node, we have the same with its predecessors:

$$Domain(u) = \bigsqcup_{v \in InNode_{CFG}(u)} Domain(v) \tag{11}$$

Proof. We will only prove the first part of the property, as the second will be also proved by symmetry. As we have split the critical edges, all $v \in OutNode(u)$ have only one in-edge. So each $v \in OutNode(u)$ is dominated by u, so the domain of u contains these domains and:

$$\bigcup_{v \in OutNode(u)} Domain(v) \subseteq Domain(u)$$

Now let $t = \ldots \to u \to w \to \ldots$ be a trace. As t is also a path in the CFG we have $w \in OutNode(u)$, so:

$$Domain(u) \subseteq \bigcup_{v \in OutNode(u)} Domain(v)$$

It follows that these sets are equals.

Now for $v, w \in OutNode(u)$, let t be a trace in $Domain(p_v) \cap Domain(p_w)$. v and w have only one predecessor which is u. So as we are in a simple path, u can only appear one time in the trace, and it follows that $v = w$. So we have the contrapositive:

$$v, w \in OutNode_{CFG}(u), v \neq w \Rightarrow Domain(p_v) \cap Domain(p_w) = \varnothing \tag{12}$$

And then we have the partition property. The second one is the same by symmetry. □

Now we will consider the dominance and post-dominance frontier sets. They are not well suited to be translated into domain relations. Instead, we will consider the post-dominance frontier successors set. It is better suited to domains relations, and is very similar to the post-dominance frontier.

Property 6. Let u be a CFG node (but not START nor END), we have a partition relation between p_u and the predicates of nodes in $PDF_{succ}(u)$:

$$Domain(p_u) = \bigsqcup_{v \in PDF_{succ}(u)} Domain(p_v)$$

Proof. Each node in $PDF_{succ}(u)$ is post-dominated by u, so we have:

$$\bigcup_{v \in PDF_{succ}(u)} Domain(v) \subseteq Domain(u)$$

We also have the opposite inclusion. Let t be a trace in $Domain(u)$. Let v and w be nodes such that $t = \text{START} \xrightarrow{\star} v \to w \xrightarrow{\star} u \xrightarrow{\star} \text{END}$, and $u \geq_{Pdom} w$ but $u \not\geq_{Pdom} v$. These nodes exists because at most $v = \text{START}$, as START is only post-dominated by END. Then by definition $w \in PDF_{succ}(u)$.

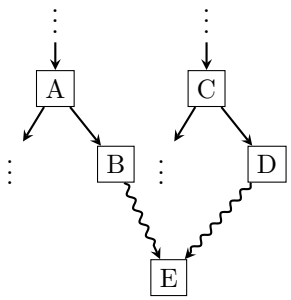

Figure 10: Example of Post-dominance frontier of a node E

We also have the disjunction of domains, because if we take a trace $t = \text{START} \xrightarrow{\star} v \xrightarrow{\star} w \rightarrow x \xrightarrow{\star} u \xrightarrow{\star} \text{END}$, with v and x in $PDF_{succ}(u)$, u will post-dominate v and so it will also post-dominate w, which is impossible. So:

$$v, w \in PDF_{succ}(u), v \neq w \Rightarrow Domain(p_v) \cap Domain(p_w) = \varnothing$$

And we have proved the partition property. □

The figure 10 shows the interesting nodes for post-dominance frontier. If we consider the node E, we have $PDF(E) = \{A, C\}$, and $PDF_{succ}(E) = \{B, D\}$. The curved arrow means that there is path, with no other incoming edge. Intuitively, all the distinct paths passing through nodes B or D will join at E, so we have a partition. This is not the case for A and C which have out edges not leading to E.

Property 7. It follows that if two nodes have the same set of post-dominance successors, they share the same partition, and so have equal domains. This property will be used to justify how we merge equivalent nodes in the construction of PPG from CDG later.

As for the others properties, the symmetric is also valid for dominance frontier. I will not describe it because it will not be used later.

4.1.1 Definition of the Predicate Partition Graph

In the previous part we have seen that we can link predicates with inclusion and partition relations. As the later gives us more information, and also hold inclusion information, we will focus on it. This definition is merely an adaptation I made from those in [6] to hyper-graphs, which are better suited to handle partition relations.

A *predicate partition graph* (PPG) is a structure which represents partition relations between symbolic predicates, more precisely the partitions formed by the respective domains. These partition relations can be represented as a hyper-graph as follows:

Definition 15. A *predicate partition graph* is a directed half-hyper-graph $H = (P, R)$, where the nodes in P represent the symbolic predicates of a CFG $G = (V, E)$, i.e., $P = \{p_v | v \in V\}$. The arcs in R of the PPG correspond to partition relations between the predicates, and have the following form: $(p_u, \{p_{v_1}, \ldots, p_{v_n}\})$, where $Domain(p_u) = Domain(p_{v_1}) \mid \ldots \mid Domain(p_{v_n})$.

An example of PPG is given in figure 11.

The graph structure of this representation allows getting information by doing graph traversals. More examples and explanations on how to use the PPG for optimisations will be given in the section 5.

In this paper we will only care about symbolic predicates (predicates coming from the CFG). However, due to the shape of predicate assignment instructions in predicated code, it can also hold relations between predicates of predicated code. It will be described in section 4.3.

Definition 16. A PPG is said to be *complete* if it has one root and all its nodes are reachable from this root node. As START and END have the `true` predicate (they are in all traces), p_{START} should be the root.

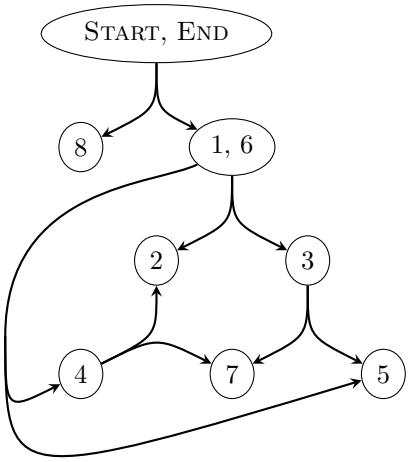

Figure 11: Predicate Partition graph of CFG in figure 8

An interesting property for completeness it to ensure that there is no loop in the PPG:

Property 8. A PPG built from a CFG is acyclic.

Proof. Let $p_a \rightarrow_{PPG} \ldots \rightarrow_{PPG} \{p_a, p_b\}$ be a loop in the PPG. We have that $Domain(p_a) = Domain(p_a) \mid Domain(p_b)$. So $Domain(p_b) = \varnothing$, which is impossible because the CFG is reachable, and there is a trace containing b. So there can be no loops in the PPG. □

Desired Properties The definition of the PPG only assures that the information stored in it will be accurate, but it says nothing about the amount of information.

We want to be able to compare any two predicated, so we want the PPG to be a connected graph. We also want a complete PPG, as it will ease creation of algorithm to query information from it. A complete PPG implies that the PPG is connected.

We also want to have precise information. For example we want to merge predicates in the PPG if they have the same domains, because it allows to regroup relations from the two node in one node. And we want to avoid to have dummy predicates, which can be used to say that there is a partition with something suitable. In fact it only gives an inclusion information.

4.1.2 Construction using the Control Flow Graph

Algorithm The algorithm 1 is adapted from the one in [6]. It takes a CFG as input, and a set a predicates where all equivalent predicates have been merged. It returns a complete PPG.

We first create a PPG with no partitions, and add predicate nodes. We suppose that there is a function `get_predicate()` which give the predicate associated to a CFG node. We then scan all nodes in the CFG. We add a partition in the PPG with `make_partition()` for each merge node to its predecessors, and to each branch node to its successors. `get_predicates()` take a set of nodes and return the set of predicates. The property 5 assures the correction of the PPG after this step.

The next step is here to make the PPG complete. For each node v which have no predecessors in the PPG in its current state, we create a partition from the immediate dominator to v and a dummy node (created by `new_predicate()`). It is correct because $Idom(v) \geq_{Dom} v$, so $Domain(v) \subseteq Domain(p_{Idom(v)})$. We ignore START in this step as it will be the root of the graph.

After this step, we know that the PPG is complete. Each PPG node has an in-edge. So if we take a PPG node p_v and follow in-edges backward, as there is no loops we will always end at p_{START}.

An example can be found in figure 11, although it is more precise because it was built with the second algorithm.

Accuracy Intuitively, the completeness step is not very accurate, as we create dummy nodes without signification in the CFG. In the original algorithm they tried to improve accuracy by saying that we

Algorithm 1: Build the Predicate Partition Graph using an acyclic CFG.

```
   Input: a CFG (V, E)
   /* make_partition, new_predicate, get_predicate(s) are explained in the text */
1  forall v ∈ V do
2  |  p_v = get_predicate(v) ;                    /* get the symbolic predicate for CFG node v */
3  |  if |OutEdge(v)| > 1 then
4  |  |  make_partition(p_v, get_predicates(OutNode(v))) ;     /* add a branch partition */
5  |  if |InEdge(v)| > 1 then
6  |  |  make_partition(p_v, get_predicates(InNode(v))) ;      /* add a merge partition */
   /* make PPG complete */
7  forall v ∈ V − {START} do
8  |  p_v = get_predicate(v) ;
9  |  if InEdge_PPG(p_v) = ∅ then
10 |  |  let p_⋆ = new_predicate() ;          /* create a new symbolic predicate in the PPG */
11 |  |  make_partition(get_predicate(Idom(v)), {p_v, p_⋆}) ;
```

should *merge equivalent CFG nodes*. They never clearly defined the condition for equivalency of CFG nodes, or which algorithm was used.

Here, we define that predicates p_u and p_v can be merged if $u \sim_{CFG} v$. The property 4 assures that $u \sim_{CFG} v$ is equivalent to the equality of domains, so this is a good definition of node equivalency.

In practice, the merging is done by `get_predicate`, which should return the same predicate for two nodes u and v such that $u \sim_{CFG} v$. We can test the equivalence of two CFG nodes in constant time by using the dominator and post-dominator tree, and interval annotation of trees. This can be precomputed in linear time by scanning nodes in the dominator tree order.

The merging step will improve accuracy because it will reduce the number of nodes with no in-edge after the first step. For example, a linear chain of nodes in the CFG will be merged into one node connected by its extremities partitions. Without merge, central nodes of the chain would be linked by dummy partitions. However, the merging is not sufficient and we still have dummy partitions on some examples (although only with unstructured code).

In the second construction, we will present an algorithm which completes the PPG with precise partitions. We could have presented it here, but as it is based on the CDG rather than CFG we choose to present it in the second algorithm.

4.1.3 Construction using the Control Dependence Graph

This algorithm build the PPG by using the CDG described in section 3.3. It is a bit more complex than the previous algorithm, as it merges equivalent nodes and build the PPG at the same time. So the code was broken into little functions, which will be described in order. We will suppose there is a function `get_predicate(CFG_node` x`)` which gives the predicate of a CFG node x.

We start with a CDG like the one in figure 9. Each edge is labeled by the name of the edge of CFG which generated the control dependence. By construction of the CDG, these labels belong to out-edges of branch nodes of the CFG. If a node x was a branch in the CFG, and now has an out-edge labeled (x, u) in the CDG, we are sure there is an edge $x \rightarrow_{CDG} u$ in the CDG.

`get_predicate_of_cdg_edge(CDG_edge` x`)` takes an edge x of the CDG. It retrieves the CFG edge (u, v) which is a label of x, and then returns p_v. It finds the predicate associated to the label of the edge, as discussed before.

Function `get_predicate_of_cdg_edge(`CDG_edge x`)`

```
   Data: x a CDG edge
   Output: The predicate corresponding to the edge x
1  let (u, v) = CFG_edge(x) ;     /* CFG_edge() gives the CFG edge that labels the CDG edge */
2  return get_predicate(v) ;      /* Return the predicate of the target node of the CFG edge */
```

`get_predicates_of_cdg_edges(set of CDG_edge` X`)` is a set version of the previous function. It returns the set of predicates obtained by applying the previous function to each edge of the set X.

In particular, `get_predicates_of_cdg_edges(`$InEdge(u)$`)` $= \{$`get_predicate(`x`)`$/x \in PDF_{succ}(u)\}$, the set of predicates of post-dominance frontier successors.

We also have that `get_predicates_of_cdg_edges(`$OutEdge(u)$`)` will give the set of predicates representing the successors of u if u is a branch.

These two special cases give set of predicates which forms a partition of u, which is why the main algorithm works. It also does some merging, when the same label appear on multiple out-edges like for the node 1 in figure 9.

Function `get_predicates_of_cdg_edges(`set of CDG_edge X`)`

Data: X a set of CDG edges
Output: The corresponding set of predicates
1 **return** $\bigcup_{x \in X}$ { get_predicate_of_cdg_edge(x) } ;

`get_predicate_of_cdg_node(CDG_node` u`)` is what actually merge nodes. It is based on the property 7, which says that nodes with same post-dominance frontier successors are equivalent. The aim of this function is to return the same predicate for equivalent nodes.

To do this, it uses an internal map which is kept between calls of the function (which is what the `static` keyword means in C). For the given node u, we compute the set pdf_{succ} which is a set of predicates of nodes in $PDF_{succ}(u)$. We want to use the map to store $pdf_{succ} \mapsto p_u$ relations.

If there are no predicate chosen for pdf_{succ} yet, we first add a new relation in the map. If the node has only one in-edge, pdf_{succ} has only one element which is a valid representative, so we choose it. If the node has multiple in-edges, only the predicate of the node itself is valid, because we cannot merge this predicate with others. Finally we return the predicate in the map for the key pdf_{succ}, which is what does the merging work.

Function `get_predicate_of_cdg_node(`CDG_node u`)`

Data: X a set of CDG edges
Output: The corresponding set of predicates
1 **static** $nodesToPredicateMap$ = init() ; /* *a map used to store the predicates already chosen* */
2 **let** pdf_{succ} = get_predicates_of_cdg_edges($InEdge(u)$) ; /* *get the set* $PDF_{succ}(u)$ */
/* *Choose a predicate to represent the set* pdf_{succ} *if it has not been done yet* */
3 **if** pdf_{succ} is not defined in $nodesToPredicateMap$ **then**
4 **if** $|InEdge_{CDG}(u)| = 1$ **then**
5 **let** $\{p\} = pdf_{succ}$; /* *extract the only predicate in* pdf_{succ} */
6 $nodesToPredicateMap[pdf_{succ}] = p$; /* *choose it* */
7 **else**
8 $nodesToPredicateMap[pdf_{succ}]$ = get_predicate(u) ; /* *choose predicate of* u */
/* *here the representative predicate for* pdf_{succ} *is defined, so return it* */
9 **return** $nodesToPredicateMap[pdf_{succ}]$

The main algorithm is now easy to understand. We take a CDG, and like the first algorithm we scan all nodes for potential partitions. We have seen before that `get_predicates_of_cdg_edges()` gives partitions when applied to $OutEdge(u)$ if u is a branch, or on $InEdge(u)$ if u is a CDG merge. So we have the correctness of the two `make_partition()`. The use of the previous function to get predicates of nodes assures that we merge equivalent nodes.

However, when u is a CDG merge, it has no parents and then the PPG is not complete. So we complete it by adding a precise partition with the function `complete()`. The main improvement is that unlike the previous algorithm, we give a precise partition.

The `complete()` function will create a partition from $Idom(v)$ to v and complementary node (instead of a dummy node). The function searches the list of complementary predicates with a graph traversal method, which is written with set notations for compactness.

Algorithm 5: Build the Predicate Partition Graph using a CDG

Input: A CDG (V, F)
forall $v \in V$ **do**
  let p_v = get_predicate_of_cdg_node(v) ;
  if $|OutEdge(v)| > 1$ **then**
    make_partition(p_v, get_predicates_of_cdg_edges($OutEdge(v)$)) ;
  if $|InEdge(v)| > 1$ **then**
    make_partition(p_v, get_predicates_of_cdg_edges($InEdge(v)$)) ;
    complete(v, p_v, $Idom(v)$) ;

First we create the list *nodesReached* of nodes visited during the traversal. We limit the traversal to the immediate dominator of v as it is the root of the partition we want to create. Then we create the list $predicatesVisited$ of predicates of edges we have seen during the traversal. And we create the list *predicatesReachable* of predicates of out-edges of nodes which were visited. Then we take all predicates which are reachable but were not visited, they are complementary to v for *idom*, so we create a partition from *idom* to the union of v and these predicates.

Function `complete`(CDG_node v, predicate p_v, CDG_node *idom*)

Input: A CDG node v and its predicate p_v
Input: The immediate dominator of v, *idom*
let $nodesReached = \{x \in V / idom \xrightarrow{\star}_{CDG} x \xrightarrow{+}_{CDG} v\}$;
let $predicatesVisited = \{$ get_predicate_of_cdg_edge(x, y) $/ idom \xrightarrow{\star}_{CDG} x \rightarrow_{CDG} y \xrightarrow{\star}_{CDG} v\}$;
let $predicatesReachable = \{$ get_predicate_of_cdg_edge(x, y) $/ x \in nodesReached \wedge x \rightarrow_{CDG} y\}$;
let $partition = \{p_v\} \cup (predicatesReachable \setminus predicatesVisited)$;
make_partition(get_predicate_of_cdg_node(*idom*), *partition*) ;

Lets see how it works with the node 4 in cdg in figure 9. The immediate dominator is 1. We have $nodesReached = \{1, 3\}$, $predicatesVisited = \{p_2, p_3, p_7\}$, and $predicatesReachable = \{p_2, p_3, p_5, p_7\}$. It will finally give us the partition $p_1 = p_4 \mid p_5$, which is correct and precise.

After this completeness step, we have a complete and precise PPG, which can be seen in figure 11.

4.2 Extension to reducible Flow Graphs

We now want to extend the algorithms to reducible CFG. The loops add some problems to the construction. The first problem is that the definition of trace is no longer valid if we want to keep the same properties. For example a trace could go through nodes 4 and 5 in figure 12, by taking each case of the branch in 3 in a different loop iteration.

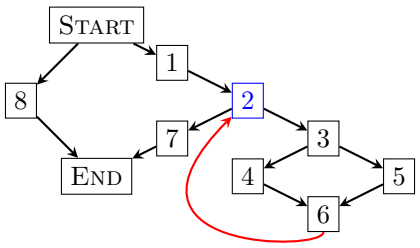

Figure 12: Example of reducible Control Flow Graph. The loop header is colored in blue, the back-edge in red.

We use a method called the *loop nesting forest*. It consists of isolating strongly connected region in the CFG, and split the CFG into two subgraphs. The first is the original CFG, where the connected

region has been assimilated to its loop header. The second is the connected region alone, but with the back-edges split. We also add to this second subgraph two new nodes START and END, to be able to use algorithms defined earlier. And to be more precise, each time an edge was going out of the connected region to a node v, we add a node $\star(v)$ in the second subgraph to represent that this is an exit of the loop. We repeat the process recursively until they are no more connected regions.

An example of splitting into loop nesting forests is given in figure 13.

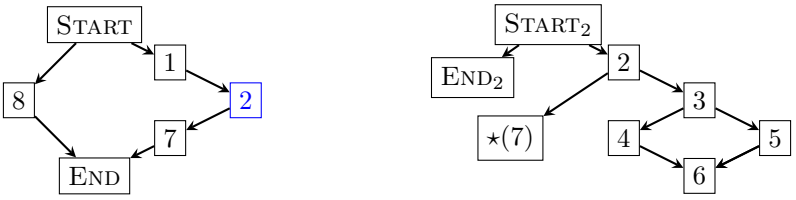

Figure 13: Control Flow Graph splitted with loop nesting relationships.

Then we have a forest of acyclic CFG, so we can apply the construction algorithm to each subgraph. The graphs are linked to each other by identical names of nodes for loop headers, and reference by $\star()$ for exit nodes.

The original algorithm handled loops by "ignoring back-edges", without giving more precision on how much they ignored it.

4.3 Predicated code

Our primary goal was to improve the construction algorithm for symbolic predicates coming from CFG structure. However, a major interest of the PPG is that it also accommodates predicate used in predication, which are not directly linked to the CFG structure. As described in [6], the shape of predicate assignment instructions automatically generates partition relations between predicates. The same article gives an algorithm to add these predicates to the PPG.

5 Application of PPG

We will now shortly describe some applications of the PPG, and compare it to others control sensitive intermediate representations we have studied.

5.1 Code motion with PPG

The PPG is well suited for a use in code motion. Code motion is the action of moving instructions in the program. There are multiple goals ; for example by moving portions of code we can speed up the program if we succeed at using a hardware pipeline. However, it is often limited to basic block range.

Sometimes, instead of entering an `if-then-else` and doing a computation, we want to move the computation up and then just take the result in the conditional part of the code. This is called speculation, and it needs to have information on control flow to go out of basic block boundaries.

The PPG allows us to easily query if when we move an instruction, it is possible and useful. For example it would not be clever to move an instruction from one side on an `if-then-else` to the other. We can check this by testing if the predicate of one position is a subset of the other predicate. This is equivalent to search a path in the PPG between the two nodes. Moreover, the PPG also allows us to get the complementary predicates where the instruction should be duplicated to keep program semantics, or where it will be added if we move it (depending on the direction of the inclusion). It can also help us to find merge nodes of SSA when adding new instructions in the program.

Unlike Gated-SSA whose control information is limited to the immediate dominator and the loops, the PPG is only limited by the loops (split in loop nesting forests). This is a good thing, as the inside of loops are not on the same levels than the outside (instructions can be executed many times inside, one time outside). So it would be a bad idea to move instructions in and out of loops.

5.2 Relation with Ψ-SSA

The PPG is already used in the construction of Ψ-SSA, but only as a tool to perform optimisations and not to hold control flow information. The first paper on Ψ-SSA [9] gives two examples of optimisations, the projection and the reduction of Ψ. The goal of these optimisations is to remove unnecessary arguments in Ψ-functions. The reduction removes arguments which will be overwritten by assignments later. The projection remove assignments which have no intersection with the predicate under which the variable is used. These optimisations reduce the size of the Ψ, but it is not guaranteed that the lost information was useless for code motion.

During the internship I went with Florian to a reunion with the designers of Ψ-SSA at ST microelectronics, and we talked about some issues we found in the papers which where not well explained. Their answers helped to understand the whole Ψ-SSA.

Each of these optimisations needs to track the domain of validity of values and so the domain of the predicate under which the value was defined. And we also need to be able to merge and compare domains. The article which defined the PPG [6] gives graph traversal algorithms to do that.

5.3 Relation with Gated SSA and Thinned Gated SSA

Gated-SSA add control flow information in a SSA-Graph. Unlike the PPG, it is at the same level than data flow information. It is only used to annotate ϕ-functions, and does not merge equivalent nodes. It is also limited to the immediate dominator scope. Its goal is to find predicates which led to an in-edge of a merge and to an input value of the ϕ-function. Gated-SSA has also hardships with loops, and finally they are enclosed in special nodes to keep loop information in the structure.

Nevertheless, it is interesting to see that is uses the CDG in its construction (in the paper of Paul Havlak [7]). It would be interesting to use the PPG to build Gated-SSA γ graphs.

I will not describe it here, but I have made a short algorithm to make the out-of-Gated-SSA translation, which was not provided in articles [6, 8].

We have not described Thinned Gated SSA, which is the structure Paul Havlak proposes in [7]. In fact, is it only a Gated-SSA graph after some transformations, like the projection and reduction in Ψ-SSA.

Conclusion

This internship was satisfying. I have learnt some methods to do research that I had not got in the L3 internship which was less satisfying. I really thanks Florian for the complete bibliography he has given to me, which was really helpful. I mostly worked with Florian, as Alain has a very tight timetable, so it was difficult to set a reunion.

One of the reason the goal of the internship was reduced was that it took too much time for me to understand the notions behind various representations. A probable reason is the average quality of the articles on the subject. They are often not precise, and sometimes just wrong and you must search for a paper who correct it when it exists. Another cause is that I had seen notions widely used like dominance in the M1 compilation course, however due to the high speed of the course I did not have the time to fully understand them. The main thing I have earned in this internship is a good understanding of these properties, and of the complexity of intermediate representation designing.

Nevertheless, I was able to get interesting results, like the designing of a new algorithm to build the PPG. I think the main result was to highlight the interests of the PPG. I first saw it as a tool, and at the end I have shown that it is a very promising representation for control dependencies.

If I have had more time left, I would have looked further into the application of the PPG. It would be very interesting to find an efficient algorithm to create a CFG from a PPG, which would optimise the number of jumps and basic blocks according to control and data dependencies. It would enable algorithms able to fully restructure the control flow of programs. For example we could use an SSA-Graph with a PPG, with the PPG holding control dependencies and thus removing the need for a CFG, and the SSA-Graph giving the data dependencies needed to build the output CFG.

References

[1] Philip L. Campbell, Ksheerabdhi Krishna, and Robert A. Ballance. Refining and defining the program dependence web. Technical Report 93-6, Department of Computer Science, The University of New Mexico, 1993.

[2] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 245–, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[4] François de Ferrière. Improvements to the psi-ssa representation. In *Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, SCOPES '07, pages 111–121, New York, NY, USA, 2007. ACM.

[5] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using ssa-graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 31–40, New York, NY, USA, 2008. ACM.

[6] David M. Gillies, Dz-ching Roy Ju, Richard Johnson, and Michael Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 114–125, Washington, DC, USA, 1996. IEEE Computer Society.

[7] Paul Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–499, London, UK, 1994. Springer-Verlag.

[8] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.

[9] Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 172–181, Washington, DC, USA, 2001. IEEE Computer Society.