Bachelor Thesis

Decompilation of LLVM IR

by Simon Moll

Saarland University
Faculty of Natural Sciences and
Technology I
Department of Computer Science

UNIVERSITÄT
DES
SAARLANDES

Supervisor:

Prof. Dr. Sebastian Hack, Universität des Saarlandes,
Saarbrücken, Germany

Reviewers:

Prof. Dr. Sebastian Hack, Universität des Saarlandes,
Saarbrücken, Germany

Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes,
Saarbrücken, Germany

Thesis submitted:

March 1, 2011

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,
March 1, 2011

_____
(signature)

Abstract

Recently, in many important domains, high-level languages have become the code representations with widest platform support surpassing any low-level language in their area with respect to completeness and importance as exchange format (e.g. OpenCL for data-parallel computing, GLSL/HLSL for shader programs, JavaScript for the web). The code representations of many actively-developed compiler frameworks [JVM,LLVM,FIRM] are designed for generating low-level machine code. They do, however, offer a broad range of language front-ends and program optimizations. In the scope of this thesis, we implemented a backend that decompiles the intermediate representation (IR) of the Low-Level Virtual Machine (LLVM) framework into a high-level code representation. The approach taken preserves functional program semantics and uses program transformations such as Controlled Node Splitting to restructure arbitrary Control-Flow Graphs. We implemented backends for both OpenCL and GLSL programs, that can decompile LLVM-Bitcode with some constraints (mostly due to unsupported data-types). The final evaluation shows that the performance of decompiled OpenCL matches that of reference OpenCL programs in most cases.

Contents

Chapter 1

Introduction

Over the past years high-level languages have become the only vendor agnostic program representation in some areas. For example in data-parallel programming of graphics hardware, there is predominantly NVIDIA's CUDA and more recently OpenCL to name a few. However, only the latter is a standardized language that is not bound to a specific vendor's platform. Also OpenCL is a high-level language and users are sought to program it directly and not to generate it from some other code representation.
On the other hand there are several compiler frameworks, which internally use code representations that in this picture are situated somewhere in between high-level languages and low-level machine code. They are designed to be independent of specific source languages and target machines. However, in the usual flow of a compiler, program code is exclusively lowered, i.e. it is only translated to representations that are closer to low-level target machines. On their way down the pipeline, programs loose the structure that was initially imposed on them by the source languages. These intermediate languages have several advantages. Conservatively, compilers translate from a specific source language A into the target machine code B. With intermediate languages, however, language front-ends translate programs in language A to the intermediate representation and there are back-ends for translating that into the B-machine code. So it is only necessary to supplement a generator towards or out of the representation to gain the support for all the available source languages or target architectures.

This thesis addresses the issue of turning high-level languages into feasible targets for compiler frameworks. We finally achieved this by deploying reverse compilation techniques and transformations that eliminate control-flow patterns which are not expressible in the target language. Using the developed decompilation algorithm, we implemented LLVM-Backends for OpenCL and GLSL (The OpenGL Shading Language).

1.1 Requirements

In the undertaking of this thesis, we aimed to comply to several requirements.

- *The back-end should be restricted as little as possible to specific high-level languages.*
  This was realized by separating the recovery of high-level control-flow from the generation of target language code. However, the target languages need to be imperative.

- *Decompilation should never fail due to unsupported control-flow.*
  Our algorithm enforces structural attributes on the intermediate representation. In contrast to forensic decompilers, the deployed transformations only preserve the functional semantics of the program. We conjecture that with our approach any Control-Flow Graph can be restructured completely, i.e. without relying on unstructured control-flow primitives (e.g. `GOTO`).

1.2 Outline

We begin with the introduction of the required terminology and related work in the area of decompilation and graph restructuring. The concept section elaborates on the characteristics of decompileable control-flow and transformations for establishing them and concludes with the discussion of the actual decompilation algorithm. We give an overview of the implementation and the design choices that were made in the core decompiler, but also in the implemented back-ends for OpenCL and GLSL. After a discussion of the evaluation results of OpenCL programs generated from LLVM-bitcode at different optimization levels, we finally conclude with an outlook on the unresolved issues we will address in future work.

Chapter 2

Background

2.1 Control-Flow Graph

A Control-Flow Graph (CFG) is a common model for representing the control flow in a function. It is a weakly connected directed graph where every node is reachable from a designated **entry node**. That is, for every node there is a path from the entry node to it.

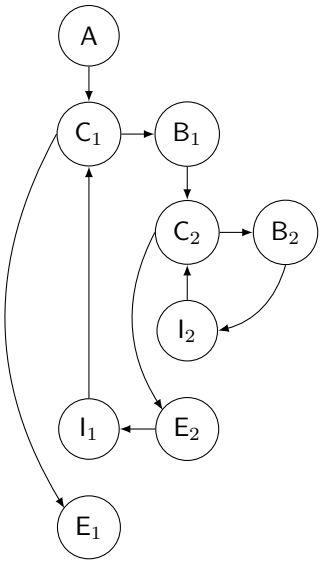


Figure 2.1: A Control-Flow Graph.

Common Terminology (from [1])

- A node x **dominates** a node y if every path from the entry node to y passes through x (written $x \succeq y$).

- A node x **post dominates** a node y if every path from y to an exit node passes through x (written $x \succeq_p y$).
- The **Dominance Frontier** of a node x ($DF(x)$) is the set of all branches (u, v) in the CFG, where x dominates u, but x does not dominate v.
- A node x **immediately dominates** a node y (written $x = ID(y)$), if x is the last dominator of y on any path from the entry node to y that is not equal to y.

In addition to common control-flow graph terminology we will introduce the following extended definitions.

Extended Terminology

- a node x **regulary dominates** a node y in a context described by a set of nodes C, (written $x \succeq^C y$), iff x dominates y in the CFG without nodes in C.
- The **regular Dominance Frontier** of a node x ($DF_C(x)$) in a context described by a set of nodes C, is the set of all branches (u, v) in the CFG, where x regulary dominates u, but x does not regulary dominate v.

2.1.1 Loop Tree

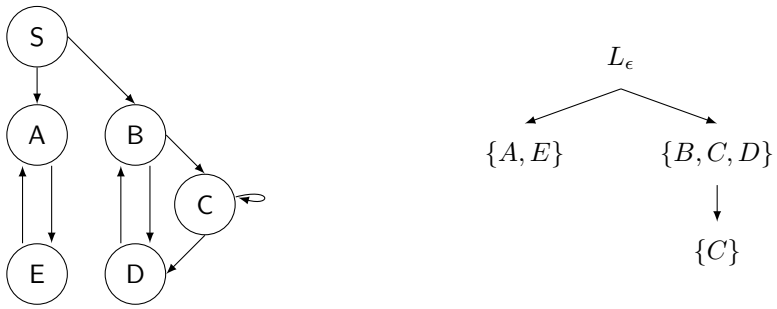


Figure 2.2: A CFG and its loop tree.

A **loop tree** identifies loops in the CFG and their relation to each other, i.e. the nesting of loops. There is not a single definition of loop trees, but several different methods of generating such a structure (in our case the method used by LLVM).
Given a loop, we define its **headers** to be the nodes of the loop, that have edges from nodes that are not in the loop going to them.

2.1.2 Region

Given a set of nodes S we define a **region** at a node x to be the set $img(DF_S(x))$. We call elements of the set S the exits of the region.

2.2 Static Single Assignment Form

The intermediate representation used in the scope of this thesis uses Static Single Assignment (SSA) form to model data flow. In SSA form [11], each instruction that returns a value writes to its own designated variable. Also a variable is only used in nodes that are dominated by the instruction that assigns to it.
There is exactly one single static assignment for each variable in the entire function. However, SSA form introduces an additional instruction. PHI-nodes are instructions that have an operand variable for each predecessor node of their parent node. If a PHI-instruction is executed, it returns the value of the variable that corresponds to the predecessor node. By using PHI-instructions as operands, instructions can select variables that were defined in predecessor nodes dependening on the path that let to the current node.

2.3 Node Splitting

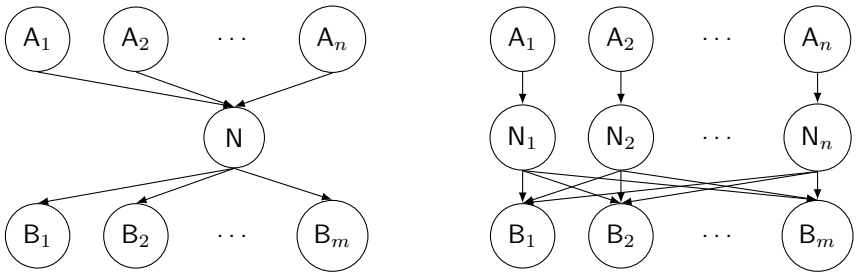


Figure 2.3: effect of a node split.

Node splitting is an operation on CFGs that aims at modifying the control flow while maintaining functional semantics. When a node is split it gets cloned, such that there is a designated node for each incoming edge of the split node (e.g. in fig. 2.3 one N_i for every A_i that branches to N). The resulting nodes only have a single predecessor each. The outgoing edges are copied with the split node and remain unchanged. Node Splitting is used for both making a CFG reducible as well as restructuring acyclic control flow.

2.4 LLVM

The Low-Level Virtual Machine (LLVM [8]) is a compiler framework that uses its own intermediate representation (IR). This IR denotes functions as CFGs and models data-flow (except for memory operations) in SSA form. The nodes of a LLVM CFG will be referred to as Basic Blocks.

A **Basic Block** is a finite list of instructions where the last instruction is always a **terminator instruction**, which denotes control flow in between basic blocks. In the scope of this thesis, we will refer to the basic blocks as nodes.
Table 2.1 shows the types of terminator instructions available in LLVM.

RETURN (x)	exit node returning the value of x.
JMP, (B_{next})	unconditional branch to node B_{next}.
BRANCH (x, B_{true}, B_{false})	2-way conditional branch to B_{true} if x is true, otherwise to B_{false}.
SWITCH (x, B_1, . . B_n)	n-way conditional branching to B_i depending on x.

Table 2.1: LLVM - Terminator Instruction types.

Chapter 3

Related Work

3.1 Reverse Compilation

In her dissertation, Cifuentes [4] developed a decompiler pipeline that converts x86 assembler to high-level C programs. The program is intended for forensic purposes and so it preserves the original structure and semantics of the decompiled program. This rules out program transformations such as node splitting or the introduction of new variables which only respect functional program semantics. Instead the decompiler resorts to GOTOs for expressing unstructured control-flow.

Lichtblau [9] introduced a graph grammar for CFG restructuring. The grammar will eventually contract any control-flow graph to a single node. However this only holds if a branch from the CFG is removed whenever the grammar lacks a fitting rule, i.e. a GOTO-statement is used instead. The proposed graph grammar explicitly includes pre and post-checked loops.

The doctoral thesis of Van Emmerik [5] deals with the decompilation of machine code programs using SSA form. Just as in compilers, the SSA Form is acknowledged as a suitable intermediate data-flow representation in decompilers. He discusses several SSA back-translation techniques and evaluates them with respect to the readability of the resulting programs.

3.2 Control-Flow Graph Restructuring

Williams and Ossher [12] discuss various methods for graph restructuring. They mention a range of restructuring strategies from those that are based on the introduction of auxiliary variables to strategies in which node splitting is involved. Furthermore they detail the notion of unstructured control flow by breaking it down to a set of properties of unstructured CFGs.

Chapter 4

Concept

It is the job of a compiler front-end to translate a high-level program into the intermediate representation. This is usually done by expressing high-level control-flow primitives in the source language with a pattern of low-level primitives. These patterns naturally constitute a graph grammar with one rule for each high-level control-flow primitive. Therefore any CFG generated by such a front-end can be translated back into a high-level program using pattern-matching rules. There exists at least one sequence of pattern applications that will succeed.

In this section we will introduce the high-level control flow primitives that the implemented decompilation algorithm recognizes. For arbitrary CFGs, however, the high-level pattern matching process may fail due to several reasons.

- The CFG could originate from a different front-end with an incompatible graph grammar. That is, the source language grammar may be capable of producing CFGs, that the decompiler grammar can not reproduce.
- Optimizations that modify the CFG may degrade any structure of the control-flow introduced by compiler front-ends.

The pattern matching algorithm is an integral part of the decompiler pipeline. But to achieve the operational stability required in real-life scenarios, it is necessary to create conditions under which the decompilation succeeds reliably. To this end we firstly identify the structural properties of decompile-able CFGs. Secondly, we discuss techniques for making the loops of arbitrary CFGs decompile-able. Thirdly, we introduce the actual decompilation process which resolves acyclic control-flow issues on the fly.

4.1 Structured Control Flow

4.1.1 Structured High-Level Primitives

The structured tree representation that the high-level primitives constitute is built on top of the underlying CFG. That is each primitive in the tree attaches to a node of the CFG. Figure 4.1 shows the CFG of the n-body test case and its high-level representation.

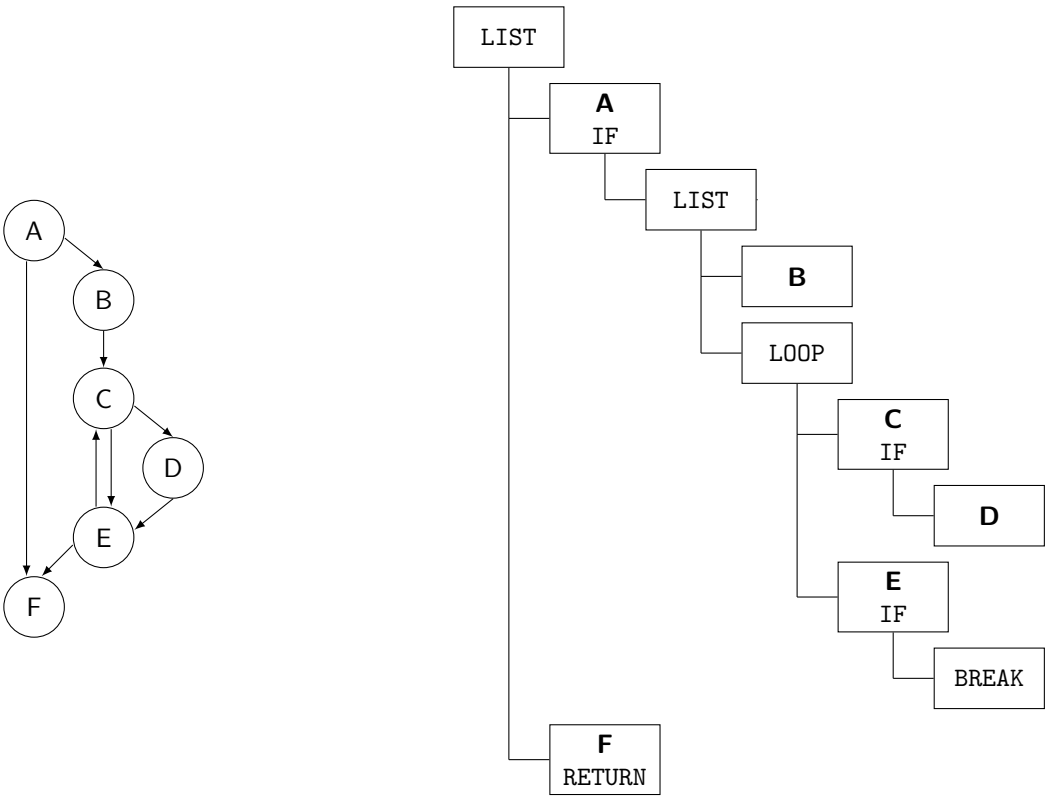


Figure 4.1: The "N-body" CFG and its high-level representation.

A noteworthy exemption from common imperative languages is the lack of a GOTO statement. The target languages (GLSL, OpenCL) do not support this primitive. Without a GOTO statement (or a similar construction) it is not possible to model arbitrary branches within the program. So for our purposes the CFG needs to be completely structured.

List of high-level Primitives

This list contains all types of high-level primitives used by the back-end. We left out pre and post-checked loops from the language. The provided primitives are sufficient for expressing them.

- `RETURN`
  End of function.
- `BREAK`
  Branch to the parent loop exit.
- `CONTINUE`
  Branch to the loop header.
- `SEQ`
  Unconditional branch to the "next" node.
- `LOOP`
  Infinite loop.
- `IF`
  2-way conditional branch. Used for both "if" and "if..else" constructs.

4.1.2 Properties of unstructured Control-Flow Graphs

A tree of high-level control-flow primitives translates to a CFG by mapping each node to a graph pattern. These patterns are only capable of expressing a subset of all possible CFGs. So for every CFG that cannot be produced from a tree, pattern matching fails for any application order. If there exists a tree for the CFG, it is called **structured**, if not it is called **unstructured** respectively.
To define restructuring CFG transformations appropriately, we first need to find a compact set of attributes that describe structured CFGs. The reproducibility by the graph grammar is such a characterization. However this definition does not yield a productive way for converting CFGs in a decompileable form. Instead, legal CFGs should be described by a small set of attributes with efficient transformations for establishing them. If that is the case then applying the transformations in sequence on an arbitrary CFG will convert it in a structured form. That is, if no subsequent transformation affects already established CFG attributes.

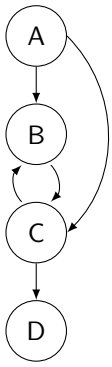


Figure 4.2: Irreducible control-flow.

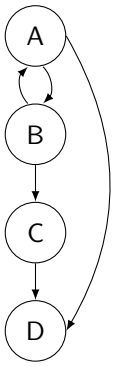


Figure 4.3: A CFG with loop-crossing branches.

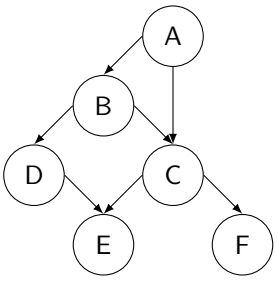


Figure 4.4: Abnormal selection paths.

- **Irreducible Control-Flow (fig. 4.2)**
  There is no different way of entering a high-level loop then by proceeding with its body node (i.e. it is not possible to branch to some more deeply nested node within the loop's body). Therefore in any structured CFG loops can only be entered at a single header block. This property is called reducibility. Conversely, CFGs where some loops have multiple headers are called irreducible. As there is only a single header `LOOP`-primitive, the decompiler language can not express irreducible control-flow.

- **Loop-crossing branches (fig. 4.3)**
  Consider acyclic sub CFGs that are completely contained in a loop body. For the high-level primitives, the only way to model branches out of this loop is by using either BREAK or RETURN nodes. BREAKs are however only capable of describing branches to the loop exit block and thus to the parent loop.

- **Abnormal Selection Paths (fig. 4.4)**
  Consider acyclic sub CFGs of a function CFG which only contain nodes within the same parent loop. Abnormal selection paths are essentially patterns in these kind of sub CFGs that the decompiler language can not express.

These are the CFG attributes we found to spoil the CFG decompilation with the proposed language. We furthermore conjecture that the decompiler language recognizes any CFG that exhibits neither properties.

4.2 Loop Restructuring

4.2.1 Controlled Node Splitting

Node Splitting is an acknowledged technique for establishing control-flow reducibility. In fact, randomly splitting nodes in a CFG will eventually yield a reducible CFG. However, it is desirable to reduce the amount of copies in the process. To this end common transformations do not apply node splitting directly. Instead, they compute the **limit graph** which is again a CFG that only contains nodes from the original CFG that participate in irreducible control-flow (e.g. the limit graph of a reducible CFG is a single node). We will denote the limit graph transformation as T. Now node splitting S is only applied on nodes that are contained in the limit graph $T(G)$ until a single node is left. (for more details see e.g. Hecht *et Al.* [6])

As Corporaal and Janssen [7] have shown splitting some nodes in the limit graph is not beneficial for making it reducible. Controlled Node Splitting (CNS) rules out these nodes and deploys a heuristics for candidate nodes. We need to define the necessary terminology for discussing CNS (from [7]):

Definition 1. *A **Loop Set** in a CFG is a set of nodes of a cycle in the CFG.*

E.g. $\{B, C\}$ is a loop set in fig. 4.2

A Shared External Dominator set (SED-set) is a subset of a loop-set L with the properties that it has only elements that share the same immediate dominator and the immediate dominator is not part of the loop-set L. The SED-set of a loop-set L is defined as:

Definition 2. *The SED-set of a loop-set L is defined as*

$$\textit{SED-set}(L) = \{n_i \in L \mid ID(n_i) = d, d \notin L\}$$

Definition 3. *A Maximal Shared External Dominator set (MSED-set) K is defined as:*

$$\textit{SED-set } K \textit{ is maximal} \Leftrightarrow \nexists SED - set\ M, K \subset M$$

Definition 4. *Nodes in an SED-set of a flow graph can be classified into three sets:*

- *Common Nodes (CN): Nodes that dominate other SED-set(s) and are not reachable from the SED-set(s) they dominate.*
- *Reachable Common nodes (RC): Nodes that dominate other SED-set(s) and are reachable from the SED-set(s) they dominate.*

- *Normal Nodes (NN): Nodes of an SED-set that are not classified in one of the above classes. These nodes dominate no other SED-sets.*

```
SCCs ← computeSCCs(G)
for all S ∈ SCCs do
  R ← S
  for all h ∈ headers(S) do
    D ← computeDominanceRegion(h, S)
    R ← R − D
    if D − {h} = ∅ then
      candidates ← candidates ∪{h}
    else
      candidates ← candidates ∪ recurse in (D − {h})
    end if
  end for
end for
if R ≠ ∅ then
  candidates ← candidates ∪ recurse in R
end if
```

Figure 4.5: algorithm for finding split candidates in a graph G.

Controlled Node Splitting only splits nodes that are in SED-sets and are not RC-nodes. This gives a clear characterization for split candidates. It is however necessary to find a method of identifying these kind of nodes. To this end we use the algorithm stated in Figure 4.5. The algorithm starts by computing the strongly-connected components (SCCs) of the CFG. For each SCC the headers are those nodes that can be reached from outside of the SCC. Clearly, these nodes are included in SED sets. For each header h, the algorithm recurses on the set of nodes that the header dominates. If there is a SED set and thus another SCC in that region then because all these nodes are elements of the same SCC, h is reachable from a SED-set it dominates. This makes it a RC-node (see Def. 4) and an unsuitable split candidate. If however there is no SCC in the dominance region of h within S then h is either a common or normal node (h could still dominate a SCC that is not nested in S). Also it is necessary to inspect the part of the SCC which is not dominated by any header (R). Similar to reducible loops it could contain nested SCCs that again feature irreducible control flow.

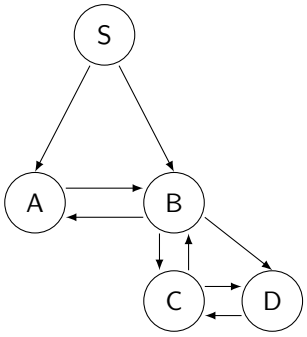


Figure 4.6: irreducible graph.

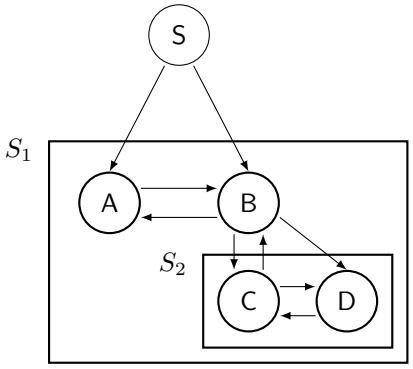


Figure 4.7: detected SED-nodes.

Figure 4.7 shows an example run of the algorithm. The algorithm detects the SED-nodes A,B,C and D. A and B are the headers of the SCC S_1. B dominates the SCC S_2 which makes it a RC-node. The headers C and D of the SCC S_2 and A of SCC S_1 respectively do not dominate anything and thus they are split candidates.

Code-growth

In theory, node splitting may result in exponential code-growth. The authors of the original paper [7] tested Controlled Node Splitting one some real-world programs that feature irreducible control-flow. For their test cases, they found the optimal splitting sequence to introduce in average 3.0% new nodes (relative to original number of nodes). Controlled Node Splitting with a heuristic does perform close to the optimum (3.1% new blocks).

4.2.2 Loop Exit Enumeration

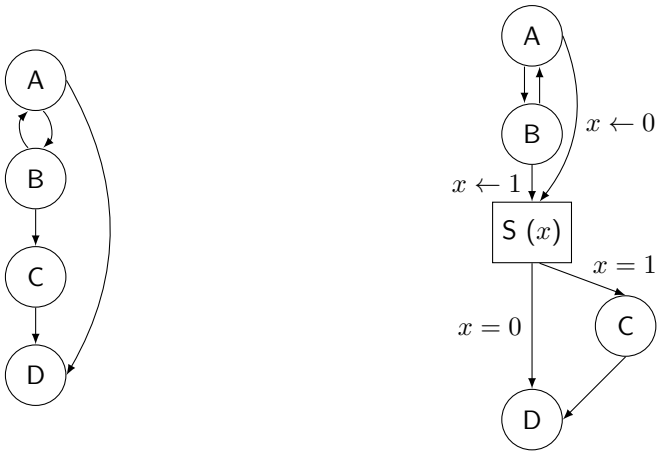


Figure 4.8: CFG before and after loop exit enumeration.

The Loop Exit Enumeration pass traverses the loop tree from inner-most to outer-most loop. For each processed loop it gathers all exit nodes of the loop body, that is nodes which are reachable from the loop header but are not contained in the loop. These exit nodes are enumerated (e.g. in fig. 4.2.2 the loop $\{A, B\}$ exits to nodes C and D). If there is at most one exit block, this loop is skipped as it is already single-exit. Alternatively a new block is created, that all branches from the loop body to identified exit blocks get redirected to. A new PHI-instruction in the joined exit node identifies the exiting nodes with the number of the node they formerly branched to. In a switch in the joined exit node this number then indexes to the corresponding loop exit. To cope with the n-way switches, a subsequent transformation converts them into 2-way conditionals.
As there now exists a single joined exit node instead of several, the loop becomes single-exit. Also as processing a loop only introduces new exits to its parent loop, the processing order from inner to outer most loop ensures that the single-exit property of any nested loop is preserved.

Running Time

The number of created exit blocks is bound to be less than or equal to the number of loops in the CFG because at most one additional block is created for every loop.

4.3 Decompilation Algorithms

After the loop-related control-flow has been restructured by CNS and Loop Exit Enumeration, the acyclic portions of the CFG may still be unstructured. This task of restructuring can be phrased as follows: given an acyclic CFG and a set of structured control-flow patterns, transform the CFG such that the given patterns can reproduce it, while preserving functional semantics. In many cases (structured CFGs) no transformation will be required at all. However if the CFG is unstructured the transformation should degrade it as little as possible.

4.3.1 Abstract Acyclic Control-Flow Primitive

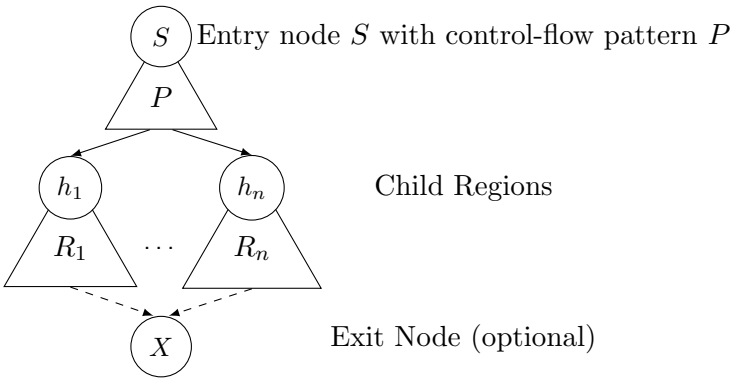


Figure 4.9: Abstract high-level control-flow primitive.

In contemporary programming languages there is a wide range of acyclic control-flow primitives. We do however recognize that many of them share some structural properties. This yields the concept of the abstract acyclic high-level primitive (see fig. 4.9)

- **Single entry node**
  Similar to reducible loops a structured control-flow pattern has a designated entry node (node S). This node dominates the part of the CFG which makes up the specific control-flow pattern of the primitive (P).
- **Child Regions**
  The primitive branches to a fixed set of disjoined regions (regions R_1 to R_n). The set of exits of a region R with entry node h is described by $img\ DF_{C'}(h)$, where C' is some set of ignored exits. The primitive region, starting at node S dominates all its child regions.

- **Single Exit Node**
  The joined set of exits (**Exit Set**) of all regions (R_1 to R_n) contains at most a single node. The notion of the exit of a region is described by edges leaving the dominated region of its header. Therefore the rule can be restated stated as:

$$|\bigcup_{i}^{n} img(DF_{C_i}(h_i))| \leq 1$$

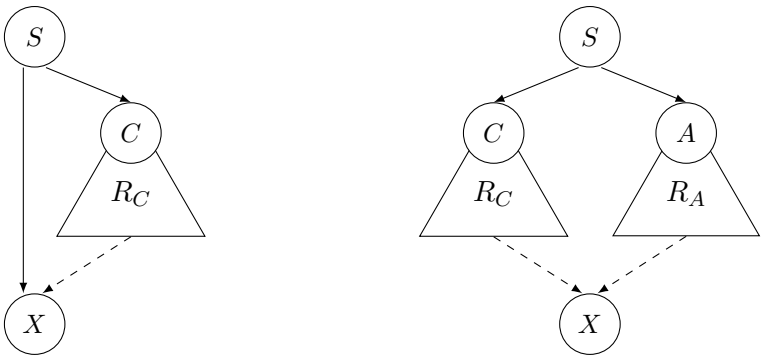


Figure 4.10: IF and IF..ELSE control-flow pattern.

The implementation at hand supports `IF` and `IF..ELSE` primitives (patterns seen in fig. 4.10). Note, that the `IF..ELSE` pattern will match any 2-way conditional in the graph. The resulting regions may however not satisfy the exit node property.

The general concept of an abstract high-level primitive applies to e.g. Short-Circuit-`IF`s as well (also note the similarity to loop related control-flow patterns). To add high-level Switches the "Single Exit Node"-rule has to be relaxed because regions would correspond to labels in the switch and control-flow may cross labels (i.e. if there is no `BREAK` in front of labels).

4.3.2 Decompilation Algorithm for structured CFGs

The algorithm starts on the function's body region. It checks which supported primitive can generate the control-flow pattern at its entry node. Eventually, it will find a matching pattern and extract the entry nodes to the child regions of the control-flow primitive. The algorithm then descends recursively on the child regions and extracts their high-level node.

Definition of Regions within the Extraction

A region in a CFG is defined by its entry node and a set of anticipated exit nodes. If the regions resides in a loop, these exits include the header and the exit node of that loop. This is because within that loop branches to the header or to the outside can be expressed at any point using CONTINUE or BREAK (at this point, loops are single-entry and single-exit).

Additionally, there is a natural notion of an exit node. E.g. an execution of the body region may leave an `IF`-primitive to a succeeding primitive. Also, the body region of a `LOOP` always exits to the loop header as this primitive loops infinitely.
The context information required in order to define the region at a node and to identify nested child loops is given by:

- The anticipated **exit node** of the region
- The **parent loop**
- Parent loop header (**continue node**)
- Parent loop exit (**break node**)

The extraction context of the function's entry node has all its fields set to null. That is, it starts on the function body and thus BREAK or CONTINUE is not possible. The definition of regions of the abstract acyclic control-flow primitive is parameterized by a set of anticipated exit nodes. For the decompilation algorithm, this set contains all defined nodes of the extraction context, i.e. fields that are not set to null.

By its definition structured Control-Flow Graphs can be decompiled without transforming them. That is, there is a high-level primitive that matches the control-flow pattern at the entry node of a region. Also, the resulting child regions satisfy the single exit node property. Therefore, the parent high-level primitive can be built from its recursively decompiled child nodes.
A region may, however, contain a sequence of primitives, such that each primitive exits on the entry node of its successor. Necessarily the exit sets of listed primitive, but the last one, contain a single exit node.

Decompilation of Primitive Sequences

When a region is decompiled the algorithm does not only parse the primitive at the entry node. Instead it greedily parses primitives at the returned exit node until either no exit node is returned or the returned exit node is the exit node of the extraction context. The extraction context of that is used unchanged for extracting the listed primitives. If the obtained sequence of primitives is longer than one, the decompiler constructs a `LIST` node from it. Otherwise the sole parsed primitive is returned.

Decompilation of structured Loops

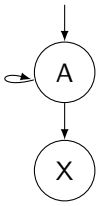


Figure 4.11: simple loop.

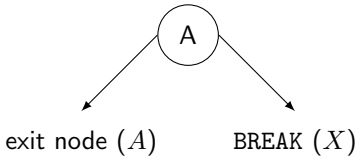


Figure 4.12: loop body branches within the extraction context.

The entry node may be the header of a nested loop and therefore any control-flow at that node occurs in the loop body. Consequently even if an acyclic primitive matches the immediate control-flow pattern at the entry node, the loop takes precedence. The extraction context for the loop body is based on the parent context with the following changes:

- **Exit node**
  The exit node is equivalent to the header of the loop.

- **Continue node**
  Also the loop header.
- **Break node**
  The unique exit node of the loop.
- **Parent loop**
  The detected loop.

Note, that there is a peculiar behavior in the treatment of loops. If the decompiler recursed on the loop body without taking into account that it just entered a loop, then it would detect that the entry node was equivalent to the exit node within the extraction context, so it would just return `CONTINUE`. To mend this issue the decompiler temporarily disables this code path. Also the entry node is equivalent to the exit node. This is, however, unproblematic because the parser enforces that at least one primitive is parsed.

Decompilation of 2-way Nodes

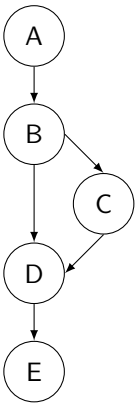


Figure 4.13: non-trivial IF-case.

The current implementation features two primitives for modeling 2-way control-flow patterns, the `IF` and `IF..ELSE` primitive. The algorithm has to choose among them and, in the `IF` case, needs to identify the successor node which is the entry node to the child region of the primitive. We ignore the branching condition at this stage because branch targets can be swapped by negating the condition and so both successors could potentially be used as body regions for the `IF`-primitive.
A 2-way node can be described by an `IF`-primitive, if

- **Trivial case** (see e.g. fig. 4.12)
  One of the branches leads to the exit node of the extraction context. The non-trivial successor is the entry node to the body without changing the extraction context.

- **Join case** (e.g. fig. 4.13)
  None of the branches is trivial and one node is an exit of the region of the other. The node which is not an exit makes the entry node to the body. The other is set as exit node in the extraction context.

Note, that this characterization is not exhaustive, e.g. if none of the exits is trivial and the exit set is empty (that is the child regions only leave using `RETURN`,`BREAK` or `CONTINUE`), both `IF` and `IF..ELSE` could express the control-flow appropriately (in fact, then there is no control-flow between the regions).

The decompiler creates an `IF..ELSE` if it will not use an `IF`-primitive. The extraction context for both child regions is set to be the single exit node of the primitive, if any.

Remaining Cases

- `BREAK` / `CONTINUE`
  Created if the processed node equals the corresponding nodes in the extraction context.

- `RETURN`
  Created for nodes without successor.

- **Unconditional Terminator**
  If the decompiler does not enter a loop with this entry node and also that node has only a single successor, then the node is encapsulated in a `SEQ` primitive.

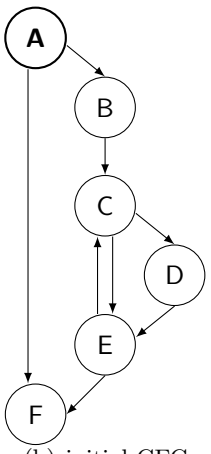
4.3.3 Example: OpenCL - NBody kernel

In the following, we discuss a run of our algorithm on the "n-Body" CFG seen in Figure 4.1. This CFG is taken from an actual test case that we will detail in the Implementation section. On the left side of the CFG, we show the extraction context as a table. In the middle, we depict the subgraph as defined by the extraction context and the entry node of the region. The entry node is printed in bold. On the right hand side, we show a table with the extraction context for the child regions. When talking about the dominance frontier, we always mean the S-regular dominance frontier, where S is the set of all defined nodes in the extraction context.

Initial state

$B_{continue}$	
B_{break}	
B_{exit}	
Loop	

(a) extractor context



(b) initial CFG.

$B_{continue}$	
B_{break}	
B_{exit}	F
Loop	

(c) extractor context for child region at B.

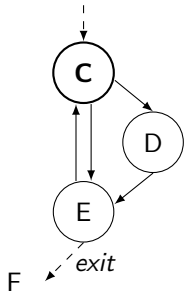
Figure 4.14: IF matching at node A.

We begin with the initial state. The extraction context is empty (Table 4.14(a)) and the algorithm start on the function's entry node A. The region entails the entire CFG (see Figure 4.14(b)). The algorithm detects an `IF`-primitive because the successor node F is element of the dominance frontier of B. Thus we enter the child region at node B. The extraction context at the child region (Table 4.14(c)) is the same as the parent's except for the exit node which is set to node F.
The next iteration will parse an `SEQ`-primitive at node B because this node branches unconditionally to node C. So a primitive sequence is parsed with the region at node C being the next element of the `LIST`-primitive.

Parsing of node C

$\mathrm{B}_{continue}$	
B_{break}	
B_{exit}	F
Loop	

(a) extractor contex at node C.



(b) region at node C.

$\mathrm{B}_{continue}$	C
B_{break}	F
B_{exit}	C
Loop	$\{C, D, E\}$

(c) loop-body extractor context at C.

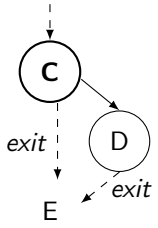
Figure 4.15: LOOP matching at node C.

When the node C is parsed, the algorithm will detect that node C is the header of the loop. So the algorithm proceeds recursively on the loop body. It sets the extraction context's break node to the unique exit node of the loop. Both the continue node and the exit node are set to the loop's header node which results in the extraction context seen in Table 4.17(a).

Parsing of the loop body at node C

$\mathrm{B}_{continue}$	C
B_{break}	F
B_{exit}	C
Loop	$\{C, D, E\}$

(a) extractor context at node C.



(b) loop-body region at node C.

$\mathrm{B}_{continue}$	C
B_{break}	F
B_{exit}	E
Loop	$\{C, D, E\}$

(c) extractor context for body D.

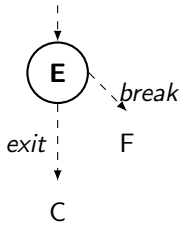
Figure 4.16: IF matching at node C.

The successor node E of node F is the exit node in the extraction context, so the trivial case `IF`-parsing occurs. The body node E is parsed with the same extraction context. Note, that according to the extraction context (Table 4.16(a)) the continue node is being parsed C. However, the algorithm does not parse `CONTINUE`-nodes if it parses the entry node of the loop body.

Parsing of node E

$\mathrm{B}_{continue}$	C
B_{break}	F
B_{exit}	C
Loop	$\{C, D, E\}$

(a) extractor context at E.



(b) region at node E.

$\mathrm{B}_{continue}$	C
B_{break}	F
B_{exit}	C
Loop	$\{C, D, E\}$

(c) child extractor context.

Figure 4.17: IF matching at node E.

Finally the algorithm parses node E. Again the trivial case of `IF`-primitive parsing occurs because node C is an exit node in the extraction context. Now if the body node F is parsed, the algorithm detects that F is the break node in the current context.

4.3.4 Generalized Decompilation Algorithm for CFGs with structured Loops

We already discussed techniques for restructuring the loops of arbitrary CFGs. In order to be able to decompile arbitrary control-flow, we will now extend the aforementioned decompilation algorithm on CFGs with structured Loops. Note, that if the extraction algorithm will find a matching primitive even when processing unstructured portions of the CFG. This is because there is a pattern for every kind of terminator instruction in the graph (unconditional branch, 2-way conditional).
The resulting child regions, however, could violate the single exit node property of abstract high-level primitives. In the extended algorithm, whenever the decompiler detects invalid child regions, the control-flow graph is transformed using a so-called **solver procedure**.

Node-Splitting based Solver Procedure

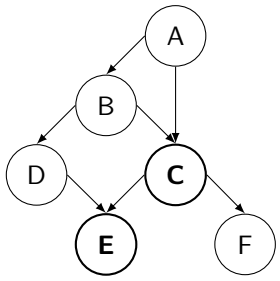


Figure 4.18: Unstructured CFG.

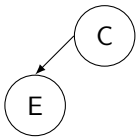


Figure 4.19: Join Graph.

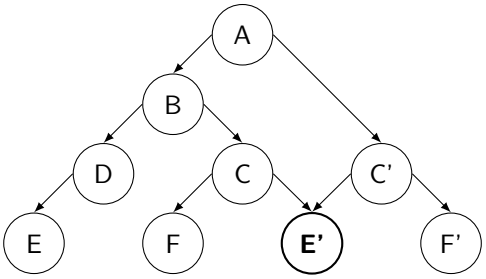


Figure 4.20: split sequence E, C, F.

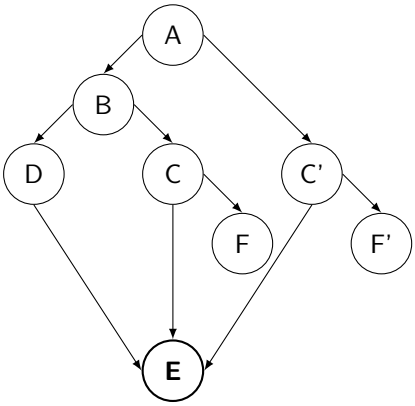


Figure 4.21: split sequence C, F.

The algorithm deploys a solver procedure, whenever a high-level primitive matches the control-flow at its entry node, but the resulting regions violate the single exit property. In other words the exit set of the child regions is greater than one (bold nodes in fig. 4.18).

4.3.5 Node Splitting Solver Procedure

In the following argumentation, we assume that in the example CFG (fig. 4.18 an `IF..ELSE` is detected with child regions at the nodes B and C. The parser does not fix an exit node and so the solver procedure may produce an arbitrary valid exit set. When the exit set has reached size one, it picks the single node of the set and makes it the exit node of all child regions.

If a node in the exit set is split, the regions will dominate their respective copy of this node and instead the node's successors will enter the set. Splitting arbitrary nodes from the exit set is, however, not a viable strategy. As Figure 4.20 depicts, splitting node E of the exit set, causes an unnecessary node split.

An non-empty exit set and the pairwise reachability of its nodes define an acyclic graph, the Join Graph (seen in fig. 4.19). We realize the exit set as a list with unique elements and an ordering constraint: if node B is reachable from node A, B is placed after A. If a node is split, its successors are put at the front of the list and rinse down behind the last node that reaches them.
The algorithm proceeds by iteratively splitting the front node of the list until it has at most a single element. This ensures, that firstly only unreachable elements of the join graph are split. Secondly by putting split successors at the front, confluent nodes in the initial Join Graph are preserved.

This behavior is crucial for minimizing the number of node splits. Consider all terminating paths within the parent region that start at the initial exit set. Assume there is a node (the exit node) for which the following holds. If none of these paths contain nodes that can be reached from the exit node, then the algorithm will eventually terminate on the exit set, which only contains that exit node.
This is true because, if there was a node that is reachable from both a predecessor of the exit node and a node of the initial exit set, then the exit node will be split first. Conversely, if there exist paths that terminate without passing through the exit node, during all iterations their nodes will be placed in front of the exit node. Therefore finally a join graph is reached, where all paths pass through the exit node. At this point, the exit node becomes a common post-dominator in the current join graph. During the next

iterations nodes on all paths to that common post-dominator will be split. So eventually the exit set of the final iteration will only contain the exit node.

We ignored loops in the description of the solver procedure. The algorithm, however, extends to graphs with structured loops. Whenever a node of the join graph that is the header of a loop is to be split, the entire loop is split instead.

4.3.6 The Case of the Mandatory Exit Node

For the sake of the argument we assumed, that the algorithm would detect an `IF..ELSE` primitive in the example CFG (fig. 4.18), when the algorithm would really match an `IF` primitive with mandatory exit node C and a single child region at node B. In this case, the algorithm uses a slight variation of the general solver algorithm. Firstly, the mandatory exit node gets removed from the exit set, i.e. the stack, before each iteration. Secondly, the algorithm iterates until the exit set is empty (e.g. fig. 4.22).

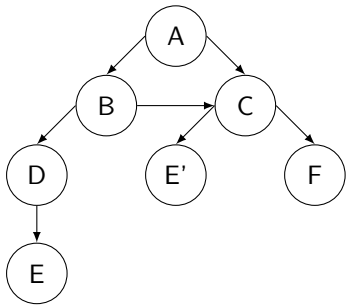


Figure 4.22: Resolved IF matching at the entry node A.

Chapter 5

Implementation

We implemented the decompiler pipeline in C++ based on LLVM 2.8. All transformations and the extraction were realized as passes within the LLVM pass infrastructure. The target language back-ends for OpenCL and GLSL share the entire pipeline up to the serialization pass. However, even that stage is not tied to a target language. The serialization pass only interacts through interfaces with target specific parts of the program.

5.1 Decompilation Pipeline Overview

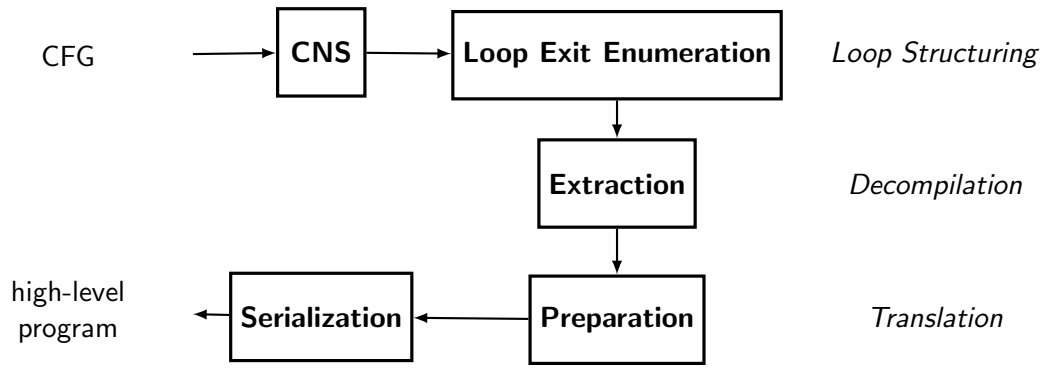


Figure 5.1: LLVM-Pass sequence of the decompiler.

The translation pipeline serializes a LLVM Bitcode module into a textual representation. Initially, the loop simplify pass of LLVM is run. This is necessary, so we can use the Loop Tree implementation of LLVM. Therefore, the decompiled CFGs differ slightly from those that were input. We describe the passes of the pipeline in the next sections.

5.2 Controlled Node Splitting Pass

As has been discussed in the concept section, our algorithm uses an implementation of Controlled Node Splitting to make CFGs reducible. The only graph transformations CNS applies to the actual CFG is node splitting and computing the limit graph. Therefore it is desirable to operate on a graph representation that supports these transformation efficiently. Furthermore the implementation should not only support heuristic driven node splitting, but also should allow finding the optimal split sequence. For big functions this is a computationally intensive task, which involves keeping numerous CFGs in memory at once.

Internal Graph Representation

Controlled Node Splitting only affects the control-flow of the function that is described by its Control-Flow Graph. The function representation of LLVM, however, also stores the contents, i.e. the instructions, with the function's CFG. Also, whenever a basic block is copied, by necessity all instructions are cloned along with it.

All of this inflicts an unnecessary computational burden when applying Controlled Node Splitting to a function. Therefore we decided to use out own data-structure that abstracts from basic blocks to nodes with adjacency lists. All operations, in particular node splitting, are then conducted on this internal representation. To be able to translate the CFG back to the representation of LLVM all nodes are labeled with a pointer to the basic block they represent. When cloning a node, its label is duplicated along with it.
Firstly, all blocks in the CFG are enumerated. The assigned index points into an array where a pointer to the original basic block is kept. All methods of the graph class take a bitmask for masking out nodes by index. This feature is frequently used to define the sub graph during the traversal of nested SCCs.

5.3 Loop Exit Enumeration Pass

The loop exit enumeration pass operates on the LLVM loop tree. After all exit nodes are collected a new basic block only containing a PHI instruction and a switch instruction is created. This basic block becomes the joined exit block of the loop. Note that the extraction framework does not implement switch instructions. The unswitch pass provided by LLVM is invoked which transforms all switches to an equivalent cascade of branches.

5.4 Extraction Pass

The extraction pass implements the generalized decompilation algorithm (see Section 4.3.4). It extracts trees of high-level primitives from the functions' CFGs. In the following, we will discuss the general implementation of the algorithm and the design choices that we made.

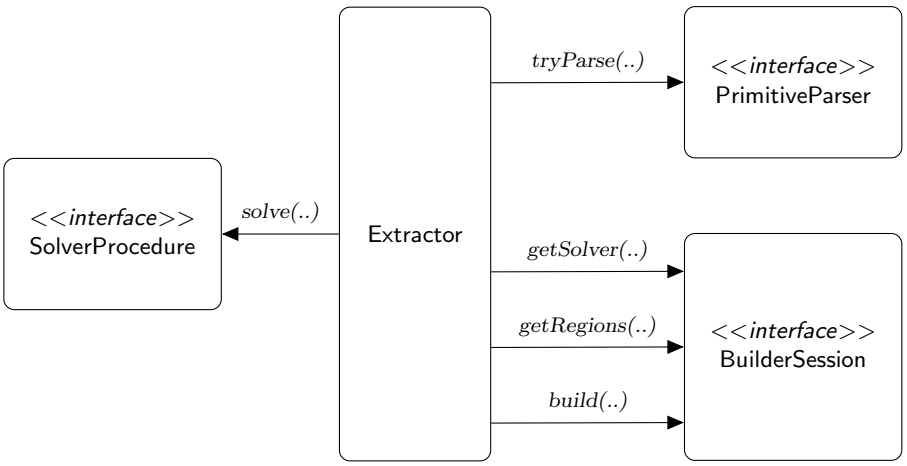


Figure 5.2: Extraction Pass class interactions.

Figure 5.2 visualizes the main classes and their interactions during the extraction of the high-level primitives. The `Extractor` class is a LLVM `ModulePass` and controls the entire process, i.e. it contains the main extraction function. The `PrimitiveParser` and `BuilderSession` interfaces divide functionality that is specific to certain high-level primitive types from the general extraction procedure. Each implementation of the `PrimitiveParser`-interface is responsible for detecting the pattern of specific high-level primitives. If a `PrimitiveParser`-object accepts the control-flow pattern at a node it sets up a `BuilderSession`-object. This object defines the child regions for this matching and a `SolverProcedure`. Also it is used to create the detected high-level primitive when all child regions have been decompiled.

5.4.1 Main Extraction Function

The main extraction function is given an entry node and an extraction context. It returns the extracted high-level primitive and the exit node of that primitive (if any).

1. Initially the function checks whether the entry node is a continue node or break node in the current extraction context. In that case, it returns

either a `CONTINUE` or `BREAK` primitive.

2. The `tryParse` method is iteratively called on a sequence of `PrimitiveParser` objects until the first returns a `BuilderSession` object.

3. The `Extractor` gets the solver procedure and the child regions of the matched primitive from the `BuilderSession` object.

4. The `NodeSplitting` procedure is invoked and enforces the single exit node property on the selected child regions.

5. The extractor recurses on the child regions' entry nodes and obtains one high-level primitive for each of them.

6. The `{build}` method of the `BuilderSession` object finally constructs a high-level primitive from the extracted child primitives.

5.4.2 Primitive Parsers

Currently, there exist two implementations of the `PrimitiveParser`-interface. We list them in the same sequence as the extraction function calls the `tryParse`-method on them.

1. **LoopParser**
   The `LoopParser` detects structured loops in the CFG. Its `tryParse` method returns a `BuilderSession`-object which constructs `LOOP`-primitives from them. If the entry node is the header of a loop that is not reducible or has multiple exit nodes `tryParse` aborts the decompilation.

2. **IfParser**
   If the `IfParser` is invoked on a node with a 2-way terminator it chooses to extract either a `IF` or `IF..ELSE` primitive from it. Depending on its choice, it returns an `IfBuilder` or `IfElseBuilder` that will create the intended primitive after the control-flow was restructured. In both cases, `getSolver` returns the `SolverProcedure`-implementation of the node splitting solver procedure.

5.5 Preparation Pass

This minor pass assures that the names of all instructions and types in the module's type symbol table are valid with respect to the target language. Instructions are named using a generic pattern (the function name, followed by the index of the node and the index of the instruction in that node).

```
...
%compute_2_2 = mul i32 %compute_0_2, %compute_2_0
%compute_2_3 = add i32 %compute_0_0, %compute_2_2
...
```

Figure 5.3: Named instructions in the n-Body test case.

5.6 Serialization Pass

The serialization pass takes the high-level representation of a LLVM-Module and translates it to a textual representation. It traverses the high-level trees of the module and issues a series of write-requests for the contents of the primitives and the primitives themselves. The implementation offers an interface for concrete receivers of these write-requests (`SyntaxWriter`). A given `Backend` object acts as a factory for creating the receiving objects. So it ultimately determines what the resulting textual representation will look like.

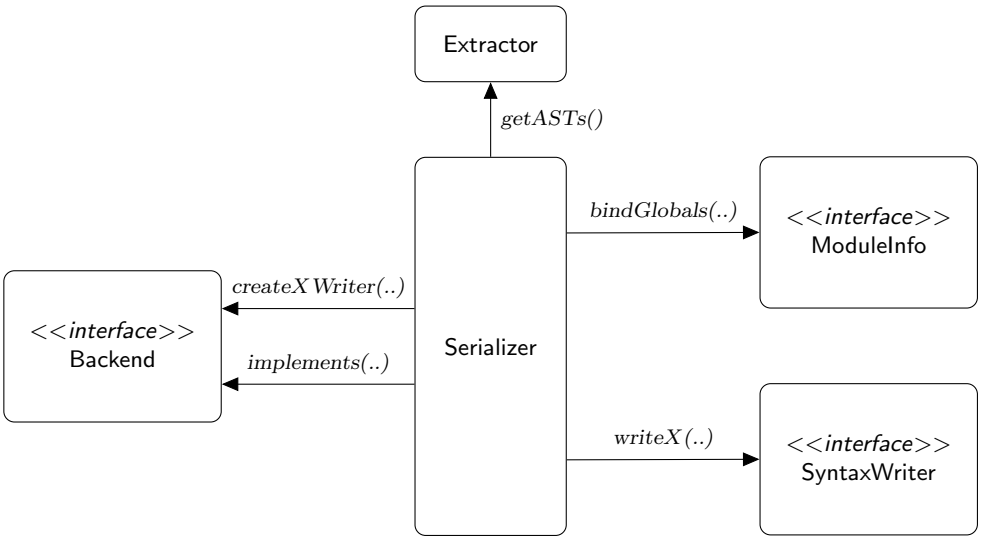


Figure 5.4: Serializer Pass class interactions.

The `Serializer` pass takes the high-level representation of a LLVM-Module from the `Extractor`. It gets an initial `SyntaxWriter` object from the target language specific `Backend` implementation and uses it to spill all symbol declarations (global variables, functions, named types) of that module. During this process, the `implements` method of the `Backend` is queried for each global symbol. If the method returns true, then the `Serializer` will skip this function. After the module prologue has been written, the `Extractor` proceeds by iteratively serializing the body nodes of all functions in the module. This is realized with the visitor pattern, i.e. the `Extractor` traverses the function's high-level representation and calls appropriate `write`X`(..)` methods on the current `SyntaxWriter` object.
The `Backend` implementation acts as a factory for three different kinds of `SyntaxWriter` objects. This is necessary, so e.g. the implementation can treat special functions in the module differently. Also this mechanism is

Figure 5.5:

used in both backends for the syntactical grouping of statement lists into block statements.

5.6.1 Syntax Writers

Table 5.1 categorizes the major `write`*X*`(..)` methods of the `SyntaxWriter` interface. The `Serializer` calls these methods during its traversal of the LLVM-Module.

Module scope
writeFunctionDeclaration(llvm::Function * func, ..)
writeFunctionHeader(llvm::Function * func, ..)
High-level primitives
writeIf(const llvm::Value * condition, bool negateCondition, ..)
writeElse()
writeLoopContinue()
writeLoopBreak()
writeInfiniteLoopBegin()
writeInfiniteLoopEnd()
writeReturnInst(llvm::ReturnInst * retInst, ..)
Instructions
writeInstruction(llvm::Instruction * inst, ..)
writeAssign(..)
Function scope
writeVariableDeclaration(..)
writeFunctionPrologue(llvm::Function * func, ..)

Table 5.1: write*X* - methods of the *SyntaxWriter* interface.

Stack of Syntax Writers

Three different kinds of factory methods in the `Backend` are used for creating `SyntaxWriter`. The `SyntaxWriter` objects are stacked on top of each other to represent the nesting of scopes during serialization.

createModuleWriter(ModuleInfo & modInfo, ..);
createFunctionWriter(SyntaxWriter * modWriter, llvm::Function * func);
createBlockWriter(SyntaxWriter * writer);

Table 5.2: create*X*Writer - methods of the *Backend* interface.

Module scope writer

An initial `SyntaxWriter`-object is created using the `createModuleWriter`-method of the backend. This object persists until the serialization of the module has finished. Its constructur can be used to write global symbol declarations (see fig. 5.6).

```
void compute(
        __global float4*,
        __global float4*,
        int,
        float,
        float,
        __local float4*,
        __global float4*,
        __global float4*);
```

Figure 5.6: Forward declaration written at module scope (n-Body example).

Function scope writer

When the `Serializer` encounters a function it obtains a writer with the `createFunctionWriter`-method. This object is used for writing the function header. The function body is then serialized with a writer obtained from `createBlockWriter`. The call sequence can be seen in Figure 5.7.

function call	**generated syntax**
FunctionWriter:: → writeFunctionHeader(..)	 __kernel compute (..)
<< constructor >> BlockWriter	{
BlockWriter:: → writeFunctionPrologue()	 float4 compute_4_3; float4 compute_4_4; ... float4 compute_7_0_in; float4 compute_7_0;
. . function contents . .	
<< desctructor >> BlockWriter	}

Figure 5.7: serialization of a function (n-Body example).

While traversing the function's high-level representation the serialization

pass pushes block writers on a stack to open a new syntactical scopes. The top writer always writes to a location that is nested within all the scopes on the stack. Finally, when the module writer gets destroyed or if the writer stack is empty, the module is completely serialized. However, by explicitly requiring the notions of module, function and block scopes the serialization API requires the target languages to have a certain structure.

5.6.2 ModuleInfo class

Generally the `ModuleInfo` implementation is responsible for storing additional information of a module that is specific to the target language back-end. E.g. both provided back-ends use them to identify functions that require special treatment during the serialization (kernel functions / shader stages, respectively). Also they are queried to obtain identifier bindings for global symbols in the module.

5.6.3 Identifier Scopes & Bindings

Identifier scopes bind LLVM value objects to strings. They are passed with every `write`X call on the `SyntaxWriter` objects so that the methods can look up the identifiers of operand values and destination variables. In the serialized programs there is exactly two kinds of identifier scopes: the global scope of the module and per-function scopes. The global identifier scope is created once for each module. It binds all global variables in the module to identifiers.
The serializer only outputs local declarations when writing the function prologue. So there are no nested identifier scopes within a function. Also, there is exactly one identifier scope for each function.

5.6.4 PHI-Node Elimination

We use a naïve algorithm for eliminating PHI-nodes during program serialization. There are two identifiers for each PHI-node in the program. Whenever a branch is taken, we create an assignment to the input identifier (x`_in`) for every PHI-node in the target node. When the target node itself is serialized, we iterate over all its PHI-nodes and create an assignment from the input identifier to the identifier bound to the PHI-node (x `=` x`_in`).

5.6.5 Example: n-Body Kernel

The figures below all show the same section of the n-Body test program during different stages of the decompilation. The listing in figure 5.8 is taken

from the initial bitcode file. Figure 5.9 shows its high-level tree representation. The final product can be seen in figure 5.10 (for complete listings, refer to appendix A).

```
for.body29:
  %compute_4_0 =
     phi i32
       [ %compute_4_26, %for.body29 ],
       [ %compute_3_0, %for.body29.preheader ]
  %compute_4_1 =
     phi <4 x float>
       [ %compute_4_25, %for.body29 ],
       [ %compute_2_1, %for.body29.preheader ]

    [ . . . ]

   br i1 %compute_4_27, label %for.end.loopexit, label %for.body29

for.end.loopexit:
  %compute_5_0 =
     phi <4 x float>
       [ %compute_4_25, %for.body29 ]
   br label %for.end
```

Figure 5.8: Excerpt from the n-Body bitcode.

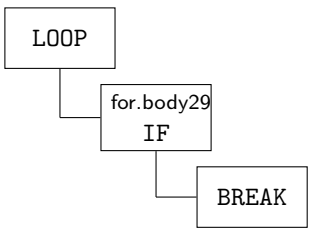


Figure 5.9: Part of the decompiled n-Body tree.

```
while(true)
{
    compute_4_0 = compute_4_0_in;
    compute_4_1 = compute_4_1_in;

    [..]

    if (compute_4_27)
    {
        compute_5_0_in = compute_4_25;
        compute_5_0_in = compute_4_25;
        break;
    }
    else
    {
        compute_4_0_in = compute_4_26;
        compute_4_1_in = compute_4_25;
    }
}
```

Figure 5.10: Generated OpenCL code.

5.7 Back-ends

5.7.1 OpenCL Back-end

The OpenCL programming language is syntactically very close to C. Noteworthy derivations from plain C include the notion of different memory spaces, vector types, annotations for kernel functions and additional opaque data types. Our implementation adheres to the OpenCL 1.0 specification [2].

Engineering Details

The memory hierarchy OpenCL operates on was mapped to designated address spaces in LLVM. The module info class essentially only contains methods for identifying kernel functions and stores the output stream. The backend's factory methods are rudimentary wrappers for creating new instances of one of the tree writer classes. Leaving the embracing prologue and finalization outputs aside all writers share the same behavior, except the function writer that accounts for kernel functions by case distinction when asked to emit a function header.

Primitive Mappings

LLVM Type	OpenCL Type	Remarks
`float`,`i32`,`i16`,`i8`,`i1`	float, int, short, char, bool	
`<`I `x float>`	float$I$	$I \in \{2, .., 16\}$
`<`I `x i1>`	int$I$	no vector of boolean values
`<`I `x i8>`	char$I$	
`<`I `x i16>`	short$I$	
`<`I `x i32>`	int$I$	
T `addrspace(1)*`	__global $S(T)$ *	T must not contain a pointer type
T `addrspace(2)*`	__constant $S(T)$ *	
T `addrspace(3)*`	__local $S(T)$ *	
T `*`	$S(T)$ *	
T `addrspace(100)*`	S(T)	opaque type wrapper space
`{` T_1, .. , T_n `}`	struct { D_{T_1}(x1); .. D_{T_n}(xn); }	
`[`n `x` T`]`	$S(T)$ [n]	
`opaque`	event_t, sampler_t, image2d_t, image3d_t	identified by their OpenCL name

Figure 5.11: OpenCL Type mappings $S(T)$ ($D_T(x)$ denotes declarations).

The particular address space indices where chosen to match those of the LLVM OpenCL front-end that comes with the AMD Stream SDK v2.2. This is except the special address space 100 which in the following will be referred to as the no-pointer address space. Pointer values in that address space will be treated as being the objects they point to. Additionally, memory operations on these pointer values are undefined. This enables e.g. declaring local opaque values and passing them by value.
This way types of the target language that are unknown to the LLVM type system can be represented in bitcode. Alternatively, they could have been associated with equivalently structured LLVM types. For example an LLVM struct of 16 floats could get reinterpreted as a four by four matrix. There is rationale against this. Firstly, providing a direct mapping clarifies what target language code will be generated when using a specific type in LLVM. Secondly, only types that are not likely to be subject of structural decomposition will be mapped to target language types (e.g. the matrix example).

5.7.2 GLSL Backend

```
@color = addrspace(2) global <4 x float> zeroinitializer , align 16
@outFragment = addrspace(4) global <4 x float> zeroinitializer, align 16
@result = global i32 0, align 4

define zeroext i1 @shadeFragment
  (<4 x float> %fragVertex, <4 x float> %fragNormal, i32 %inA, i32 %inB)
  nounwind noinline
{
entry:
  %tmp = load <4 x float> addrspace(2)* @color
  store <4 x float> %tmp, <4 x float> addrspace(4)* @outFragment
  %add = add i32 %inB, %inA
  store i32 %add, i32* @result
  ret i1 true
}

define void @shadeVertex
  (<4 x float> %normal, <4 x float> %vertex, i32 %vertIn, i32 %otherIn)
{
entry:
  %call = tail call <4 x float> @flat_f4(<4 x float> %vertex)
  %call2 = tail call <4 x float> @smooth_f4(<4 x float> %normal)
  %call4 = tail call i32 @flat_u(i32 %vertIn)
  %call6 = tail call i32 @flat_u(i32 %otherIn)
  %call7 = tail call zeroext i1 @shadeFragment
      (<4 x float> %call, <4 x float> %call2, i32 %call4, i32 %call6)
  ret void
}

declare <4 x float> @flat_f4(<4 x float>)

declare <4 x float> @smooth_f4(<4 x float>)
```

Figure 5.12: Basic OpenGL shader bitcode

The OpenGL Shading Language is also syntactically a C-based language. The data-flow between processing stages does, however, lack a counter part in the LLVM Bitcode. Additional complications include the existence of multiple output streams, where there is one for each stage. The design choices for coping with these challenges where made with future extendibility in mind. Our back-end implements parts of the OpenGL Shading Language v3.3 [3].

Engineering Details

Bitcode structure of OpenGL Shader Programs

In the following, we will discuss the basic structural attributes of OpenGL

shader programs specified in LLVM Bitcode. (example seen in fig. 5.12, capital case macros represent e.g. address space annotations).

With regard to the OpenGL rasterization pipeline a LLVM module can implement single processing stages up to functions for the entire pipeline. To enforce that source Bitcode for the GLSL Back-end defines processing stages (shader types) at most once, each gets identified with a reserved function name (`shadeFragment`,`shadeVertex`).

A fragment shader can terminate execution by accepting or discarding its result (`return discard` statements in GLSL). There is no such thing as a discarding terminator in LLVM-Bitcode. So instead the returned value is reinterpret as either accepting the fragment (true $\rightarrow$ `return`) or discarding it (false $\rightarrow$ `discard`).

In the OpenGL rasterization pipeline there is data-flow between consecutive processing stages. E.g. the vertex shader operates on a per-vertex basis and can store custom data at each vertex. These values are interpolated over the geometry. Whenever the fragment shader is invoked on a surface sample of the geometry it has access to the interpolated values at that location. We represent this passing of data between stages with function calls. Special intrinsic functions define the interpolation behavior for each value. If a value is passed directly, the interpolant is also unspecified in the resulting GLSL program and default interpolation is assumed.

Processing Stage-aware Serialization

In applications, OpenGL Shader Programs are usually built from a set of shader source files, which are compiled separately and then linked together. There is a shader source file for each processing stage. Moreover it is not possible to specify more than one processing stage in a single source file. In our Bitcode representation this is, however, possible. So the writers have to unravel the processing stages of a module into their designated output streams, such that the resulting shader files will make the correct program, when linked together. This involves for example writing shared symbols to all shader files that use them and serializing shader functions only into their designated stream.

Supported Processing Stages

For this thesis, support for vertex and fragment shaders has been implemented. The overall framework is, however, extensible for additional processing stages such as geometry or tessellation shaders. Their implementation requires new reserved function names and target streams in the

module info class. Also processing stages may bring a new set of intrinsics with them.

Primitive Mappings

LLVM Type	**GLSL Type**	**Remarks**
`float`,`i32`,`i16`,`i8`,`i1`	float, int, short, char, bool	

`<`I `x float>`
`<`I `x i1>`
`<`I `x i8>`
`<`I `x i16>`
`<`I `x i32>`
	vec$I$
	bool$I$
	char$I$
	short$I$
	int$I$
		$I \in \{2, 3, 4\}$

T `addrspace(1)*` T `addrspace(3)*`	uniform $S(T)$; out $S(T)$;	T must be global. fragment shader output.
T `addrspace(100)*`	S(T)	opaque type wrapper space
`{` T_1, .. , T_n `}`	struct { D_{T_1}(x1); .. D_{T_n}(xn); }	
`[`n `x` T`]`	$S(T)$ [n]	
`opaque` `opaque`	mat$i$xj sampler1D, sampler1DArray	$i, j \in \{2, 3, 4\}$

Figure 5.13: GLSL Type mappings $S(T)$ ($D_T(x)$ denotes declarations).

5.8 Limitations

5.8.1 General Limitations

Any language feature of LLVM that is not natively supported in OpenCL or GLSL will either abort the decompilation process or produce invalid programs with respect to the specification. Common aborting features include include:

- unsupported types (strings, function pointers, invalid sized vectors).
- unknown address spaces.
- dynamic memory allocation.
- dynamic vector element access (`ExtractElementInst`).

The following features will be decompiled, but the resulting programs may not compile:

- structs (container types) with pointer type elements.
- recursive function calls.

5.8.2 OpenCL-specific Limitations

Like all other OpenCL specific intrinsics memory barriers are realized as function calls in bitcode. Node splitting can duplicate such calls and two paths that passed through the same block before, may not join in that node afterwards.

5.8.3 GLSL-specific Limitations

The OpenGL Shading Language does not support any kind of pointer type. The use of pointer types in Bitcode will immediately abort the serialization process.

Chapter 6

Evaluation

All tests where conducted on a Intel Core 2 Quad (Q9400) machine with 4 GB of RAM and a NVIDIA GeForce 9600 GT graphics card (driver version 195.36.24). We took the OpenCL test cases from the AMD Stream SDK v2.2 and ported them to C augmented with our intrinsics. The kernels were then compiled to LLVM bitcode using the Clang C front-end. All the OpenCL samples in the SDK come with a CPU reference computation, that we used to verify the correctness of the synthesized kernels. As the kernel CFGs were already structured they were unaffected by the restructuring transformations. We evaluated the running time of the original kernels in comparison with those generated from unoptimized (O0) and optimized (O3) LLVM-Bitcode modules. Problem sizes were increased all over the board to obtain more expressive results on the kernel runtime.

6.1 OpenCL Results

6.1.1 Running Time of structured Control-Flow Graphs

kernel (n)	runtime (*original*)	runtime (O0)	runtime (O3)	relative *o.* / O3
	[ms]	[ms]	[ms]	[%]
NBody (65536)	1271	1271	1333	95.3
FFT (2^{24})	106	106	106	100.0
FloydWarshall (512)	780	780	440	177.3
BlackScholes (64)	3	3	3	100.0
DCT (1024)	57	57	58	98.3
Binomial Option (1024)	33	33	33	100.0

Table 6.1: OpenCL runtime performance.

kernel	basic blocks	decompile time
	no.	[ms]
NBody	9	9
FFT	39	62
FloydWarshall	4	6
BlackScholes	2	8
DCT	16	10
Binomial Option	23	6

Table 6.2: Bitcode complexity and decompile time (O0).

The results in Table 6.1 show the average runtime of the OpenCL test kernels in ten iterations (original and synthesized (build command `clang++ -emit-llvm -m32` with `-O0` and `-O3`)). The graphics hardware used for testing has a 32-bit architecture. This is communicated to the front-end by passing the `-m32` command line option. Bitcode sizes in number of basic blocks and the required decompilation times are shown in Table 6.2.

6.1.2 Correctness

All kernels except the FFT-testcase passed perfectly, i.e. the computed values were equal to those of the CPU reference computation. As for FFT, we detected minor deviations from the ground truth when optimizations were disabled (O0). Further analysis revealed a maximal difference of 0.01 in 0.01% of the computed values (2^{24} in total) over ten randomized iterations.

6.1.3 Performance Impact of LLVM Optimizations

Inspection of the NBody kernel functions showed, that LLVM optimizations eliminate multiple loads to the same location and reuse the value from the first load instead. This caused the suboptimal running time for the NBody-O3 case, as the kernel performed equally to the original when the load was replicated manually.
On the other hand, the same reuse of loaded values could significantly improve the kernel running time of the FloydWarshall kernel. Overall the impact of optimizations was low.

6.2 Interpretation

Our results suggests that naïve PHI-Node elimination and a simple infinitive loop primitive with BREAK and CONTINUE statements is sufficient for

lossless decompilation of structured CFGs. This is most likely due to optimizations occurring in the NVIDIA OpenCL compiler. We assume that this compiler internally uses standard techniques such as SSA form and CFGs that eliminate any artifacts introduced by decompilation. At the time of writing no other tool chain was available so it must be speculated how the kernels would perform, e.g. on AMD graphics hardware.

Chapter 7

Conclusions

With our decompiler back-end for LLVM we have shown that it is possible to decompile LLVM Bitcode into high-level programs. The evaluation provides evidence that this process is lossless for structured Control-Flow Graphs. In the case of unstructured CFGs restructuring transformations can generate a structured program with equivalent functional semantics. We conjecture that our pipeline is able to decompile any unstructured CFG, with at most 2-way conditionals, into a completely structured high-level representation.

Chapter 8

Future Work

8.1 Improvements to the Decompiler

The implemented high-level primitives have proven to be sufficient for expressing structured programs. In order to reduce block duplication in more complex cases, it may be beneficial to add support for additional primitives such as switches and short-circuit boolean expressions. As mentioned in the discussion of abstract high-level primitives, general switches may specify different mandatory exit nodes for each child region, i.e. if there is no break to delimit in front of case labels. To handle these cases, the concept of solver procedures and high-level primitives needs to be generalized further.

We currently use node splitting to cope with unstructured acyclic control-flow. Eventhough it did not occur in the test cases, this strategy may lead to a significant increase of code size and more divergent control-flow. GPU-architectures, however, operate more efficiently on uniform control-flow patterns, i.e. divergence among threads in a group may provoke serial scheduling. Block predication could possibly avoid this.

We build our own the high-level representation on top of LLVM CFGs. However, the Clang compiler front-end framework for LLVM defines a well established AST-format. Translating our representation into the Clang AST format would open up several opportunities. Targeting Clang-ASTs would make an actual decompiler with respect to the LLVM framework. Also the serialization pass could then operate on Clang ASTs directly.

Findings while implementing Controlled Node Splitting suggest the algorithm for CNS could be made more efficiently. The improved algorithm would operate on the “irreducible loop tree“ that we build for detecting node candidates. This has minor priority, however, unless the decompiler will be applied to bigger programs than the fairly brief kernel functions and shaders.

8.1.1 Sign Recovery

LLVM does not distinguish signed and unsigned integer values by type (e.g. both `int` and `unsigned int` become `i32`). Instead if there is an integer operation that dependends on its argument being signed or not, then there are two instances of that function (one interpreting the argument as signed, the other as unsigned respectivelly). We account for this in the case of builtin LLVM instructions (e.g. the cast of an unsigned integer to a float type) by reinterpreting the argument at the call site before passing it to the function. We intend to improve this by also specifying the argument signs of target language intrinsics. Also, a sign recovery pass could recover sign information, such that most of the sign-casts are eliminated. This would enhance the readability of the decompiled programs.

8.1.2 Back-end Feature Completeness

The GLSL back-end is not yet feature complete in terms of the language specification. We will implement all intrinsics and stage functions in the OpenGL 4.1 core profile (e.g. geometry shaders / tesselation).

8.1.3 Evaluation of Control-Flow Restructuring

The OpenCL kernels are necessarily structured, so we can't use our back-end directly to measure the impact of restructuring CFG transformations on the kernel/shader running time. It would only be possible to compare structured programs obtained generated with different restructuring methods (e.g. predication, node splitting, etc.). With the recent PTX-Backend [10] for LLVM, however, we could directly assess the effect of restructuring transformations because this back-end also generates code from unaltered unstructured CFGs. We could measure the actual performance impact of CFG restructuring on graphics hardware.

8.2 GPU-centric Optimizations For LLVM

As has been discussed in the evaluation section, some LLVM optimizations do actually make memory access patterns worse for GPUs. Although LLVM bitcode features address spaces, their concept is limited to disjoined memory locations. Critically there is no notion of SIMT function executions and therefore LLVM optimizes the code for multi-threaded execution on the

CPU at best.
In contrast, the GPU threading model is fairly complex with threads being grouped hierachally. At every level of the hierachy there is shared memory that can be accessed from any group member. This scales up to global memory that is freely accessible from any thread at any time. Also memory operations are only efficient if all members of a group simultaneously access addresses that are linear in their index within that group. All of this is not yet included in LLVM.

In the case of OpenGL, shader program performance could be improved by moving computations to earlier processing stages and using interpolated results instead (e.g. compute some value per-vertex and let the hardware interpolate it for each fragment). Although this is lossless for linear operations, more aggressive optimizations may be possible when accepting some (controlled) loss in visual quality. In spirit of Loop-Independent Code Motion (LICM), such a class of optimizations could be referred to as Stage-Independent Code Motion (SICM). Other optimizations include e.g. storing precomputed values in textures or dynamic approximation by Level-Of-Detail.

Chapter 9

References

Bibliography

[1] On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.

[2] *OpenCL 1.0 Specification (revision 48, October 6, 2009)*, 2009.

[3] *The OpenGL Shading Language (Language Version 3.30, Document Revision 6)*, 2010.

[4] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, 1994.

[5] Michael Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, 2007.

[6] Matthew S. Hecht. *Flow analysis of computer programs / Matthew S. Hecht*. North Holland, New York :, 1977.

[7] Johan Janssen and Henk Corporaal. Controlled node splitting. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 44–58, London, UK, 1996. Springer-Verlag.

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Ulrike Lichtblau. Decompilation of control structures by means of graph transformations. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development*, volume 185 of *Lecture Notes in Computer Science*, pages 284–297. Springer Berlin / Heidelberg, 1985.

[10] H. Rhodin. A ptx code generator for llvm, October 2010.

[11] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[12] M. H. Williams and H. L. Ossher. Conversion of unstructured flow diagrams to structured form. *The Computer Journal (1978) 21 (2)*, 1978.

Appendix A

Code Listings for the N-Body Simulation Example

A.1 Original N-Body OpenCL Program

```
/*
 * For a description of the algorithm and the terms used, please see the
 * documentation for this sample.
 *
 * Each work-item invocation of this kernel, calculates the position for
 * one particle
 *
 * Work-items use local memory to reduce memory bandwidth and reuse of data
 */


__kernel

void

nbody_sim(

    __global float4* pos ,

    __global float4* vel,

    int numBodies,

    float deltaTime,

    float epsSqr,

    __local float4* localPos,
    __global float4* newPosition,
    __global float4* newVelocity)

{
    unsigned int tid = get_local_id(0);

    unsigned int gid = get_global_id(0);

    unsigned int localSize = get_local_size(0);


    // Number of tiles we need to iterate

    unsigned int numTiles = numBodies / localSize;


    // position of this work-item

    float4 myPos = pos[gid];
```

```
    float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

    for(int i = 0; i < numTiles; ++i)
    {
        // load one tile into local memory
        int idx = i * localSize + tid;
        localPos[tid] = pos[idx];

        // Synchronize to make sure data is available for processing
        barrier(CLK_LOCAL_MEM_FENCE);

        // calculate acceleration effect due to each body
        // a[i->j] = m[j] * r[i->j] / (r^2 + epsSqr)^(3/2)
        for(int j = 0; j < localSize; ++j)
        {
            // Calculate acceleartion caused by particle j on particle i
            float4 r = localPos[j] - myPos;
            float distSqr = r.x * r.x  +  r.y * r.y  +  r.z * r.z;
            float invDist = 1.0f / sqrt(distSqr + epsSqr);
            float invDistCube = invDist * invDist * invDist;
            float s = localPos[j].w * invDistCube;

            // accumulate effect of all particles
            acc += s * r;
        }

        // Synchronize so that next tile can be loaded
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    float4 oldVel = vel[gid];

    // updated position and velocity
    float4 newPos = myPos + oldVel * deltaTime + acc * 0.5f * deltaTime * deltaTime;
    newPos.w = myPos.w;

    float4 newVel = oldVel + acc * deltaTime;

    // write to global memory
    newPosition[gid] = newPos;
    newVelocity[gid] = newVel;
}
```

A.2 N-Body Bitcode Module

This is the LLVM-bitcode module of the n-Body test case right before the extraction pass is run (the loop simplify pass of LLVM and our preparation pass have been run). The code was generated with Clang from a C-program (build line: `clang++ -emit-llvm -m32 -O2`). The module header and comments behind instructions were removed from the original bitcode. Also line breaks were inserted where necessary to make the listing fit on the page.

```
define void @compute(
    <4 x float> addrspace(1)* nocapture %pos,
    <4 x float> addrspace(1)* nocapture %vel,
    i32 %numBodies,
    float %deltaTime,
    float %epsSqr,
    <4 x float> addrspace(3)* nocapture %localPos,
    <4 x float> addrspace(1)* nocapture %newPosition,
    <4 x float> addrspace(1)* nocapture %newVelocity) {
entry:
  %compute_0_0 = tail call i32 @get_local_id(i32 0) readnone
  %compute_0_1 = tail call i32 @get_global_id(i32 0) readnone
  %compute_0_2 = tail call i32 @get_local_size(i32 0) readnone
  %compute_0_3 = udiv i32 %numBodies, %compute_0_2
  %compute_0_4 =
    getelementptr inbounds <4 x float> addrspace(1)* %pos, i32 %compute_0_1
  %compute_0_5 = load <4 x float> addrspace(1)* %compute_0_4
  %compute_0_6 = icmp eq i32 %compute_0_3, 0
  %compute_0_7 = bitcast <4 x float> zeroinitializer to <4 x float>
  br i1 %compute_0_6, label %for.end84, label %bb.nph134

bb.nph134:
  %compute_1_0 =
    getelementptr inbounds <4 x float> addrspace(3)* %localPos, i32 %compute_0_0
  %compute_1_1 = icmp eq i32 %compute_0_2, 0
  %compute_1_2 = bitcast i32 0 to i32
  %compute_1_3 = bitcast <4 x float> zeroinitializer to <4 x float>
  br label %for.body

for.body:
  %compute_2_0 =
    phi i32 [ %compute_1_2, %bb.nph134 ], [ %compute_6_2, %for.end ]
  %compute_2_1 =
    phi <4 x float> [ %compute_1_3, %bb.nph134 ], [ %compute_6_0, %for.end ]
  %compute_2_2 = mul i32 %compute_0_2, %compute_2_0
  %compute_2_3 = add i32 %compute_0_0, %compute_2_2
  %compute_2_4 = getelementptr <4 x float> addrspace(1)* %pos, i32 %compute_2_3
  %compute_2_5 = load <4 x float> addrspace(1)* %compute_2_4
  store <4 x float> %compute_2_5, <4 x float> addrspace(3)* %compute_1_0
  tail call void @barrier_l()
  br i1 %compute_1_1, label %for.end, label %for.body29.preheader

for.body29.preheader:
  %compute_3_0 = bitcast i32 0 to i32
  br label %for.body29

for.body29:
  %compute_4_0 =
    phi i32
      [ %compute_4_26, %for.body29 ],
      [ %compute_3_0, %for.body29.preheader ]
  %compute_4_1 =
    phi <4 x float>
      [ %compute_4_25, %for.body29 ],
      [ %compute_2_1, %for.body29.preheader ]
  %compute_4_2 =
    getelementptr <4 x float> addrspace(3)* %localPos, i32 %compute_4_0
  %compute_4_3 = load <4 x float> addrspace(3)* %compute_4_2
  %compute_4_4 = fsub <4 x float> %compute_4_3, %compute_0_5
  %compute_4_5 = extractelement <4 x float> %compute_4_4, i32 0
  %compute_4_6 = fmul float %compute_4_5, %compute_4_5
  %compute_4_7 = extractelement <4 x float> %compute_4_4, i32 1
  %compute_4_8 = fmul float %compute_4_7, %compute_4_7
  %compute_4_9 = fadd float %compute_4_6, %compute_4_8
  %compute_4_10 = extractelement <4 x float> %compute_4_4, i32 2
  %compute_4_11 = fmul float %compute_4_10, %compute_4_10
  %compute_4_12 = fadd float %compute_4_9, %compute_4_11
```

```
  %compute_4_13 = fadd float %compute_4_12, %epsSqr
  %compute_4_14 = tail call float @sqrt_f(float %compute_4_13) readnone
  %compute_4_15 = fdiv float 1.000000e+00, %compute_4_14
  %compute_4_16 = fmul float %compute_4_15, %compute_4_15
  %compute_4_17 = fmul float %compute_4_16, %compute_4_15
  %compute_4_18 = extractelement <4 x float> %compute_4_3, i32 3
  %compute_4_19 = fmul float %compute_4_18, %compute_4_17
  %compute_4_20 =
    insertelement <4 x float> undef, float %compute_4_19, i32 0
  %compute_4_21 =
    insertelement <4 x float> %compute_4_20, float %compute_4_19, i32 1
  %compute_4_22 =
    insertelement <4 x float> %compute_4_21, float %compute_4_19, i32 2
  %compute_4_23 =
    insertelement <4 x float> %compute_4_22, float %compute_4_19, i32 3
  %compute_4_24 = fmul <4 x float> %compute_4_23, %compute_4_4
  %compute_4_25 = fadd <4 x float> %compute_4_1, %compute_4_24
  %compute_4_26 = add nsw i32 %compute_4_0, 1
  %compute_4_27 = icmp eq i32 %compute_4_26, %compute_0_2
  br i1 %compute_4_27, label %for.end.loopexit, label %for.body29

for.end.loopexit:
  %compute_5_0 = phi <4 x float> [ %compute_4_25, %for.body29 ]
  br label %for.end

for.end:
  %compute_6_0 =
    phi <4 x float>
      [ %compute_2_1, %for.body ],
      [ %compute_5_0, %for.end.loopexit ]
  tail call void @barrier_1()
  %compute_6_2 = add nsw i32 %compute_2_0, 1
  %compute_6_3 = icmp eq i32 %compute_6_2, %compute_0_3
  br i1 %compute_6_3, label %for.end84.loopexit, label %for.body

for.end84.loopexit:
  %compute_7_0 = phi <4 x float> [ %compute_6_0, %for.end ]
  br label %for.end84

for.end84:
  %compute_8_0 =
    phi <4 x float>
      [ %compute_0_7, %entry ],
      [ %compute_7_0, %for.end84.loopexit ]
  %compute_8_1 =
    getelementptr inbounds <4 x float> addrspace(1)* %vel, i32 %compute_0_1
  %compute_8_2 = load <4 x float> addrspace(1)* %compute_8_1
  %compute_8_3 = insertelement <4 x float> undef, float %deltaTime, i32 0
  %compute_8_4 =
    shufflevector
      <4 x float> %compute_8_3,
      <4 x float> undef,
      <4 x i32> zeroinitializer
  %compute_8_5 = fmul <4 x float> %compute_8_2, %compute_8_4
  %compute_8_6 = fadd <4 x float> %compute_0_5, %compute_8_5
  %compute_8_7 =
  fmul
  <4 x float> %compute_8_0,
  <float 5.000000e-01, float 5.000000e-01, float 5.000000e-01, float 5.000000e-01>
  %compute_8_8 = fmul <4 x float> %compute_8_7, %compute_8_4
  %compute_8_9 = fmul <4 x float> %compute_8_8, %compute_8_4
  %compute_8_10 = fadd <4 x float> %compute_8_6, %compute_8_9
  %compute_8_11 =
    shufflevector
      <4 x float> %compute_8_10,
      <4 x float> %compute_0_5,
      <4 x i32> <i32 0, i32 1, i32 2, i32 7>
  %compute_8_12 = fmul <4 x float> %compute_8_0, %compute_8_4
  %compute_8_13 = fadd <4 x float> %compute_8_2, %compute_8_12
  %compute_8_14 =
    getelementptr inbounds
      <4 x float> addrspace(1)* %newPosition,
      i32 %compute_0_1
  store <4 x float> %compute_8_11, <4 x float> addrspace(1)* %compute_8_14
  %compute_8_16 =
    getelementptr inbounds
      <4 x float> addrspace(1)* %newVelocity,
      i32 %compute_0_1
  store <4 x float> %compute_8_13, <4 x float> addrspace(1)* %compute_8_16
  ret void
}

declare i32 @get_local_id(i32) readnone
```

```
declare i32 @get_global_id(i32) readnone
declare i32 @get_local_size(i32) readnone
declare void @barrier_l()
declare float @sqrt_f(float) readnone
```

A.3 Generated OpenCL program

This file (`./nbody.cl`) was built using the command "`. opencl nbody`" in the root directory of the project. Line breaks were inserted manually where necessary to make the listing fit on the page. Also empty lines were removed.

```
void compute(
    __global float4*,
    __global float4*,
    int,
    float,
    float,
    __local float4*,
    __global float4*,
    __global float4*);

__kernel void compute(
    __global float4* pos,
    __global float4* vel,
    int numBodies,
    float deltaTime,
    float epsSqr,
    __local float4* localPos,
    __global float4* newPosition,
    __global float4* newVelocity)
{
   float4 compute_4_3;
   float4 compute_4_4;
   float compute_4_5;
   float compute_4_6;
   float compute_4_7;
   float compute_4_8;
   float compute_4_9;
   int compute_2_0_in;
   int compute_2_0;
   bool compute_1_1;
   float4 compute_2_1_in;
   float4 compute_2_1;
   int compute_0_0;
   int compute_0_1;
   int compute_0_2;
   int compute_0_3;
   float4 compute_0_5;
   bool compute_0_6;
   int compute_2_2;
   int compute_2_3;
   float4 compute_2_5;
   int compute_4_0_in;
   int compute_4_0;
   float4 compute_4_1_in;
   float4 compute_4_1;
   float4 compute_8_0_in;
   float4 compute_8_0;
   float4 compute_8_2;
   float4 compute_8_3;
   float4 compute_8_4;
   float4 compute_8_5;
   float4 compute_8_6;
   float4 compute_8_7;
   float4 compute_8_8;
   float compute_4_10;
   float compute_4_11;
   float compute_4_12;
   float compute_4_13;
   float compute_4_14;
   float compute_4_15;
   float compute_4_16;
   float compute_4_17;
   float compute_4_18;
   float compute_4_19;
   float4 compute_4_20;
   float4 compute_4_21;
   float4 compute_4_22;
   float4 compute_4_23;
   float4 compute_4_24;
   float4 compute_4_25;
   int compute_4_26;
   bool compute_4_27;
   float4 compute_6_0_in;
```

```
float4 compute_6_0;
int compute_6_2;
bool compute_6_3;
float4 compute_8_9;
float4 compute_8_10;
float4 compute_8_11;
float4 compute_8_12;
float4 compute_8_13;
float4 compute_5_0_in;
float4 compute_5_0;
int compute_1_2;
float4 compute_1_3;
int compute_3_0;
float4 compute_0_7;
float4 compute_7_0_in;
float4 compute_7_0;

compute_0_0 = get_local_id(0x00000000);
compute_0_1 = get_global_id(0x00000000);
compute_0_2 = get_local_size(0x00000000);
compute_0_3 = (as_uint(numBodies) / as_uint(compute_0_2));
compute_0_5 = (pos)[compute_0_1];
compute_0_6 = (compute_0_3==0x00000000);
compute_0_7 = (float4)(0.0f);
if (! compute_0_6)
{
   compute_1_1 = (compute_0_2==0x00000000);
   compute_1_2 = 0x00000000;
   compute_1_3 = (float4)(0.0f);
   compute_2_0_in = compute_1_2;
   compute_2_1_in = compute_1_3;
   while(true)
   {
      compute_2_0 = compute_2_0_in;
      compute_2_1 = compute_2_1_in;
      compute_2_2 = (compute_0_2*compute_2_0);
      compute_2_3 = (compute_0_0+compute_2_2);
      compute_2_5 = (pos)[compute_2_3];
      (localPos)[compute_0_0] = compute_2_5;
      barrier(CLK_LOCAL_MEM_FENCE);
      if (! compute_1_1)
      {
         compute_3_0 = 0x00000000;
         compute_4_0_in = compute_3_0;
         compute_4_1_in = compute_2_1;
         while(true)
         {
            compute_4_0 = compute_4_0_in;
            compute_4_1 = compute_4_1_in;
            compute_4_3 = (localPos)[compute_4_0];
            compute_4_4 = (compute_4_3-compute_0_5);
            compute_4_5 = compute_4_4.s0;
            compute_4_6 = (compute_4_5*compute_4_5);
            compute_4_7 = compute_4_4.s1;
            compute_4_8 = (compute_4_7*compute_4_7);
            compute_4_9 = (compute_4_6+compute_4_8);
            compute_4_10 = compute_4_4.s2;
            compute_4_11 = (compute_4_10*compute_4_10);
            compute_4_12 = (compute_4_9+compute_4_11);
            compute_4_13 = (compute_4_12+epsSqr);
            compute_4_14 = sqrt(compute_4_13);
            compute_4_15 = (1.0f/compute_4_14);
            compute_4_16 = (compute_4_15*compute_4_15);
            compute_4_17 = (compute_4_16*compute_4_15);
            compute_4_18 = compute_4_3.s3;
            compute_4_19 = (compute_4_18*compute_4_17);
            compute_4_20.s0 = compute_4_19;
            compute_4_21 = compute_4_20;
            compute_4_21.s1 = compute_4_19;
            compute_4_22 = compute_4_21;
            compute_4_22.s2 = compute_4_19;
            compute_4_23 = compute_4_22;
            compute_4_23.s3 = compute_4_19;
            compute_4_24 = (compute_4_23*compute_4_4);
            compute_4_25 = (compute_4_1+compute_4_24);
            compute_4_26 = (compute_4_0+0x00000001);
            compute_4_27 = (compute_4_26==compute_0_2);
            if (compute_4_27)
            {
               compute_5_0_in = compute_4_25;
               compute_5_0_in = compute_4_25;
               break;
            }
            else
```

```
                {
                    compute_4_0_in = compute_4_26;
                    compute_4_1_in = compute_4_25;
                }
            }

            compute_5_0 = compute_5_0_in;
            compute_6_0_in = compute_5_0;
        }
        else
        {
            compute_6_0_in = compute_2_1;
        }
        compute_6_0 = compute_6_0_in;
        barrier(CLK_LOCAL_MEM_FENCE);
        compute_6_2 = (compute_2_0+0x00000001);
        compute_6_3 = (compute_6_2==compute_0_3);
        if (compute_6_3)
        {
            compute_7_0_in = compute_6_0;
            compute_7_0_in = compute_6_0;
            break;
        }
        else
        {
            compute_2_0_in = compute_6_2;
            compute_2_1_in = compute_6_0;
        }
    }

    compute_7_0 = compute_7_0_in;
    compute_8_0_in = compute_7_0;
}
else
{
    compute_8_0_in = compute_0_7;
}
compute_8_0 = compute_8_0_in;
compute_8_2 = (vel)[compute_0_1];
compute_8_3.s0 = deltaTime;
compute_8_4 = (float4)(compute_8_3.s0000);
compute_8_5 = (compute_8_2*compute_8_4);
compute_8_6 = (compute_0_5+compute_8_5);
compute_8_7 = (compute_8_0*(float4)(0.5f, 0.5f, 0.5f, 0.5f));
compute_8_8 = (compute_8_7*compute_8_4);
compute_8_9 = (compute_8_8*compute_8_4);
compute_8_10 = (compute_8_6+compute_8_9);
compute_8_11 = (float4)(compute_8_10.s012, compute_0_5.s3);
compute_8_12 = (compute_8_0*compute_8_4);
compute_8_13 = (compute_8_2+compute_8_12);
(newPosition)[compute_0_1] = compute_8_11;
(newVelocity)[compute_0_1] = compute_8_13;
return;
}
```