

**CODE GENERATION AND ADAPTIVE CONTROL
DIVERGENCE MANAGEMENT FOR LIGHT WEIGHT
SIMT PROCESSORS**

A Thesis
Presented to
The Academic Faculty

by

Meghana Gupta

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College Of Computing

Georgia Institute of Technology
May 2016

Copyright © 2016 by Meghana Gupta

**CODE GENERATION AND ADAPTIVE CONTROL
DIVERGENCE MANAGEMENT FOR LIGHT WEIGHT
SIMT PROCESSORS**

Approved by:

Professor Sudhakar Yalamanchili, Advisor
College Of Computing
Georgia Institute of Technology

Professor Hyesoon Kim
College Of Computing
Georgia Institute of Technology

Professor Santosh Pande
College Of Computing
Georgia Institute of Technology

Date Approved: 25 April 2016

ACKNOWLEDGEMENTS

I want to express my sincere gratitude to my advisor Dr.Sudhakar Yalamanchili, for accepting me into the CASL research group and supporting me throughout my degree program. His guidance was invaluable to my research. I am deeply indebted to his unwavering support and guidance. I look up to him immensely, and aspire to be like him.

I am also very grateful to Chad Kersey. He has patiently answered all my questions about the HARP architecture and ISA. His feedback was extremely helpful in the HARP compiler development.

I am also very grateful to Dr.Hyesoon Kim and my colleagues in the PNM research group, they have listened intently to all my presentations about the HARP compiler and given invaluable feedback throughout the course of 2 years.

My special thanks to Blaise Tine and Joo Hwan Lee for chipping in compiler development and integrating the LLVM Test Infrastructure.

Finally, I am very fortunate to have a supporting and close-knit family. I am very thankful to my parents and sister who have supported and encouraged my every endeavour, and are my constant source of strength.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
LIST OF ABBREVIATIONS	viii
SUMMARY	ix
I INTRODUCTION	1
II BACKGROUND	8
2.1 The HARP Project	8
2.1.1 HARP Architecture	8
2.1.2 HARP ISA	9
2.1.3 HARP Toolchain and Execution Model	10
2.1.4 Control divergence management in HARP	12
2.2 LLVM compiler infrastructure	15
2.2.1 LLVM Intermediate Representation (IR)	15
2.2.2 LLVM Design Philosophy	16
2.2.3 Documentation and Community	17
III THE HARP COMPILER	18
3.1 Overview	18
3.2 The HARP backend	20
3.2.1 LLVM Target Independent Code Generation-HARP backend Co-design	20
3.2.2 Instruction Selection	22
3.2.3 Register Allocation	26
3.2.4 Frame Lowering	28
3.2.5 Asm Printer	29
3.2.6 Peephole Code Generator	29

3.2.7	Control divergence Pass	30
3.3	Integration of HARP backend	30
3.3.1	LLVM Build System	30
3.3.2	Cross Compilation Support in Clang	31
IV	COMPILER SUPPORT FOR CONTROL DIVERGENCE MAN-	
	AGEMENT	32
4.1	Split-Join Infrastructure	32
4.1.1	Factors to be addressed for split-join insertion	33
4.1.2	Algorithm for split-join insertion	38
4.2	Predication	41
4.2.1	Predication Support in HARP backend	41
4.2.2	Limitations of LLVM If-conversion pass	42
4.3	Decision framework for control divergence management	43
4.3.1	Static analysis driven decision framework for control diver-	
	gence management	44
4.3.2	Profile guided decision framework for control divergence man-	
	agement	47
V	EXPERIMENTS AND RESULTS	51
5.1	Evaluation of code generation	52
5.2	Evaluation of decision framework for control divergence management	53
VI	CONCLUSION	57
VII	FUTURE WORK	58
REFERENCES	60

LIST OF TABLES

1	<code>harp32</code> and <code>harp64</code> target configurations	19
2	Pseudo instructions and their equivalent HARP instruction sequences that cannot be represented in SSA form	30
3	List of tests to evaluate code generation of HARP compiler	52

LIST OF FIGURES

1	HARP Architecture	2
2	Schematic diagram of Split-Join Execution	5
3	Conversion of control dependence to data dependence via If-conversion	5
4	HARP toolchain	10
5	A typical execution of application program on HARP	11
6	Split-Join Example	13
7	Basic LLVM Architecture	15
8	HARP compiler	18
9	HARP backend	20
10	HARP Target Classes	21
11	LLVM - HARP backend co-design for Instruction Selection	22
12	LLVM - HARP backend co-design for register allocation	26
13	Control flow graph marked with IPDOM for common control flow structures	33
14	Control flow graph of the binsearch kernel during different stages of compilation	34
15	Control flow graph of the binsearch kernel marked with ipdom	35
16	Control flow graph of binsearch kernel after aggressive optimizations .	36
17	Dynamic instruction count bloat with unstructured to structured optimization	37
18	Effect of turning off aggressive compiler optimizations on kernel dynamic instruction count	37
19	LLVM-HARP backend Co-design for If-conversion	41
20	Ratio of conditional branches in <code>harpbench</code> that can be predicated by LLVM If-conversion pass	43
21	Divergence ratio of <code>harpbench</code> programs executed with a 16 warp, 16 lane configuration	53
22	Kernel dynamic instruction count of different control divergence management techniques	54

LIST OF ABBREVIATIONS

SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SPMD	Single Program Multiple Data
GPU	Graphics Processing Units
HARP	Heterogeneous Architecture Research Prototype
PIM	Processing In Memory
HMC	Hybrid Memory Cube
LLVM	Low-Level Virtual Machine
IR	Intermediate Representation
IPDOM	Immediate Post-Dominator
PC	Program Counter
SP	Stack Pointer
FP	Frame Pointer
CFG	Control Flow Graph
RISC	Reduced Instruction Set Computing
VLIW	Very Large Scale Instruction Word
RTL	Register-Transfer Logic
KDIC	Kernel Dynamic Instruction Count

SUMMARY

A multitude of graph algorithms and machine learning algorithms are being written for Graphics Processing Units to exploit the performance and energy efficiency SIMT-style (Single Instruction Multiple Thread) execution brings to the table. These application paradigms process large amounts of data, and have several dynamic irregularities, limiting the effectiveness of caches. This leads to large volumes of data-movement. The large costs associated with data-movement necessitate architectural changes for such workloads, bringing attention to Processing In Memory (PIM) architectures. PIM architectures reduce the volume of data movement, and internally provide high memory-level parallelism. The result is a positive impact on performance and energy.

In order to explore the design space of in-memory SIMT architectures, an architecture prototype called the Heterogeneous Architecture Research Prototype (HARP) has been developed. The HARP architecture consists of light-weight SIMT cores placed in 3D stacked DRAMS. We utilize the instance of 3d memory cubes referenced to as the Hybrid Memory Cube (HMC), developed by Micron. The HARP architecture is driven by the HARP ISA, a SIMT ISA.

The main objective of this thesis is developing a compiler and associated optimizations for the HARP architecture. The HARP compiler lowers programs written in C into the HARP ISA. It is built on top of the LLVM open source compiler infrastructure, which provides high quality tools for building compilers. Following are the main contributions of the thesis:

- **Addition of extensions to C language to support SIMT execution model:** Extensions have been added to the C language to identify HARP kernel functions, `warp Id` and `lane Id`. These extensions aid code generation, to apply analyses and transformations specific to data-parallel kernels.

- **Code Generation for the HARP ISA:** A new compiler backend has been developed in the LLVM compiler infrastructure to support code generation for the HARP ISA. The HARP backend implements target specific support required by standard code generation phases. It also implements a second layer of instruction selection for some idiosyncratic HARP instruction sequences that cannot be represented in the LLVM intermediate representation during traditional instruction selection.
- **Compiler support for control divergence management:** The HARP architecture provides two kinds of control divergence management techniques to handle divergent branches. They are, *hardware stack based control divergence management* and *predication*. Both of these techniques require code generation support from the HARP compiler. New compiler analyses and transformations have been developed to generate code to implement these functionalities.
- **Decision framework to choose control divergence management:** Both control divergence management techniques supported by the HARP architecture have unique advantages and disadvantages. It is advantageous to use hardware stack based control divergence management for conditional branches that are likely to be unanimous and predicate conditional branches that are likely to be nonunanimous. New decision frameworks guided by static analysis and profile information have been developed to choose the appropriate control divergence management technique by analyzing the conditional branches during compilation.

CHAPTER I

INTRODUCTION

Graphics Processing Units (GPUs) have evolved tremendously over the last two decades. From special purpose hardware with limited flexibility, GPUs have evolved into highly programmable general purpose computing devices. This evolution has been guided by the motivation of bringing performance and energy efficiency of GPUs to a wide range of applications. Changes to the GPU architecture for efficient mapping of general purpose programs, and the development of programming models like CUDA [21] and OpenCL [22] have paved the way for this evolution. As a result, diverse applications are being written for GPUs, to exploit the massive parallelism these architectures have to offer.

Graph processing algorithms are among the new applications targeted for GPUs. These algorithms work on large amount of data and have highly irregular memory access patterns, leading to large volumes of data-movement. In order to reduce the costs associated with data-movement for such workloads, new architectural approaches are required. Processing In Memory (PIM) architectures are an attractive solution for such workloads. PIM architectures not only reduces the volume of data-movement, but also provides access to a high memory-level parallelism which SIMT (Single Instruction Multiple Threads) architectures can effectively exploit. Additionally, longer latencies of PIM architectures can be overcome by fast context switches supported by SIMT architectures. Due to these considerations, a new in-memory SIMT architecture called Heterogeneous Research Architecture Prototype (HARP) [15, 17, 16] has been developed.

The HARP architecture consists of multiple light-weight SIMT cores placed in

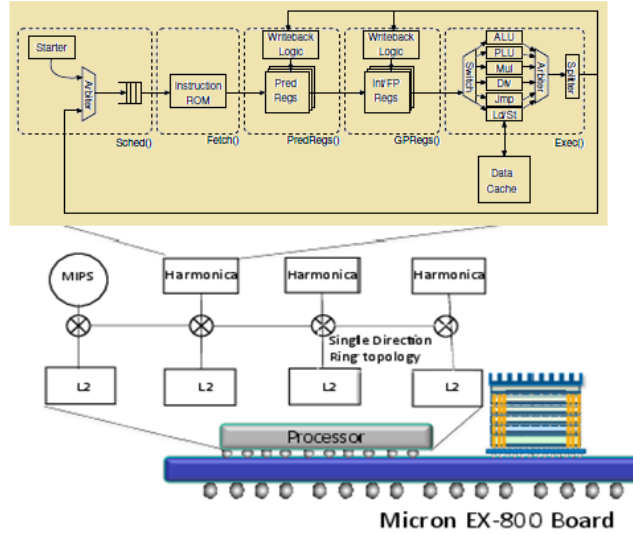


Figure 1: HARP Architecture

a 3d stacked memory. In our analyses and implementation we target the Hybrid Memory Cube [12], developed by Micron. Figure 1 shows the schematic diagram of the HARP architecture. It implements the HARP ISA, a RISC-like ISA with instructions for synchronization and SIMT control. The HARP instruction stream can represent parallel execution by multiple threads over independent register sets. The HARP architecture was designed such that parameters like number of SIMT cores, SIMD width, number of registers, etc., are customizable and supported by tool chains that can generate synthesizable Register-Transfer level (RTL) design. This parameterizability enables flexible research.

In the current execution model for the HARP architecture, a single thread begins execution of a kernel. After initial setup, this thread launches multiple warps which execute the kernel function in SIMT style. This is different from OpenCL / CUDA execution model where a host launches data-parallel kernels and launch of warps and threads are handled transparently by the runtime. The programs for the current execution model for the HARP architecture are written in C, and use the API provided by the HARP Runtime Library to launch warps, threads, etc. This approach mitigates

the need for focusing on software issues of host-device interaction and development of complex software toolchain consisting of driver, runtime, etc. Eventually, the goal of the HARP project is to execute OpenCL programs on the HARP architecture, so that in-memory processing benefits are available for stock GPU applications written in OpenCL.

The main objective of this thesis is, building a compiler for the aforementioned execution model for HARP ISA. The HARP compiler is built on top of the open source LLVM compiler infrastructure [18], and uses the open source Clang [1] C frontend, commonly used with LLVM-based compilers. Apart from the standard details of code generation, the HARP compiler handles code generation of certain idiosyncrasies in the HARP ISA by implementing a second layer of instruction selection. It also provides compiler support necessary of control divergence management. The main contributions of the HARP compiler are discussed as follows:

- **Addition of extensions to the C language to support SIMT execution model:** Extensions have been added to the C language to identify kernel functions, `warp Id` and `lane Id`. These extensions are lowered to metadata by the Clang frontend. This metadata is used during code generation to apply analyses and transformations specific to data-parallel kernels.

- **Code Generation for the HARP ISA:**

A large portion of this thesis focuses on building the HARP backend in the LLVM compiler infrastructure to generate code for the HARP architecture. The LLVM compiler infrastructure provides target independent implementations of different code generation phases such as instruction selection, register allocation, etc. These target independent implementations query target specific details through pre-defined interfaces. Like other typical backends, the HARP backend implements target specific support required by the standard code generation phases. It also implements a second layer of instruction selection for some

idiosyncratic HARP instruction sequences that cannot be represented in the LLVM intermediate representation during traditional instruction selection.

- **Compiler support for control divergence management**

Control divergence occurs when parallel threads executing in lockstep want to execute different directions of a conditional branch. Traditional solutions to control divergence management include *hardware stack based control divergence management* and *predication*. Both these techniques are supported by the HARP architecture, and require code generation support from the compiler. A brief overview of these techniques are described as follows.

- *Hardware stack based control divergence management*: The HARP architecture provides hardware stack based divergence management [13] via `split` and `join` instructions. The `split` instruction is inserted before a conditional branch, and the `join` instruction is inserted at the reconvergence point. The `split` instruction is predicated with the same predicate as the conditional branch. On encountering a `split` instruction, PC (program counter) and the active mask are pushed onto the hardware stack, threads with a true predicate are allowed to continue, while the other threads are masked. When the `join` instruction is encountered, PC and the active mask is popped of the hardware stack, and previously masked out lanes are now allowed to continue. When the `join` instruction is encountered the second time, control flow falls through and all threads start executing in lockstep again. Figure 2 shows the schematic diagram of split-join execution.

Aggressive compiler optimizations and certain control flow structure properties make split-join insertion more complex than it may appear. The HARP compiler implements several transformations to make the control

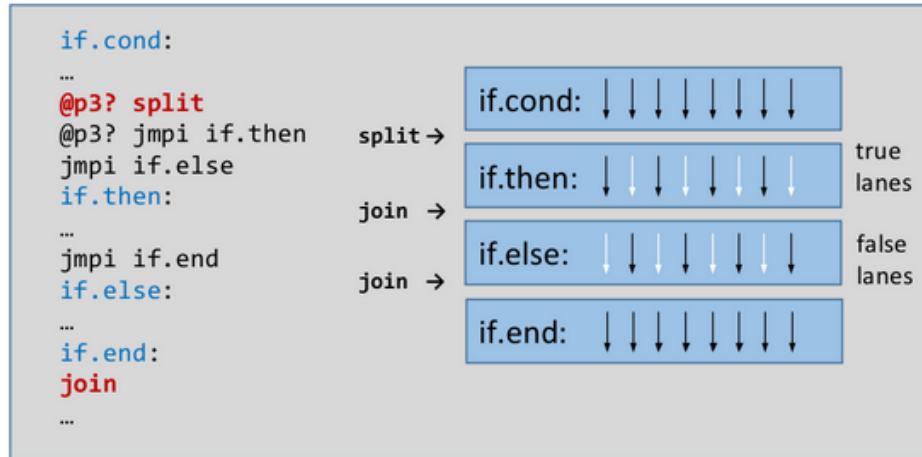


Figure 2: Schematic diagram of Split-Join Execution

flow graph amenable for split-join insertion.

- *Predication*: Predicated execution involves conditional execution of instructions. Instructions are executed normally when the predicate value is true and when the predicate value is false, it is prevented from updating state. The compiler generates code for predicated execution by a technique called If-conversion [10, 20]. If-conversion converts control dependencies into data dependencies. Figure 3 shows a predicated if-then-else statement.

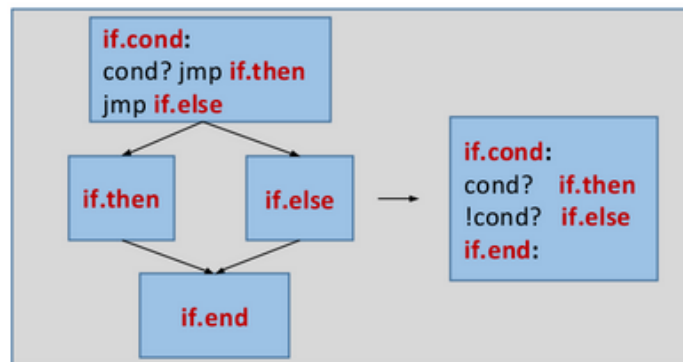


Figure 3: Conversion of control dependence to data dependence via If-conversion

HARP ISA provides several predicate manipulation instructions and predicate registers for predicated execution. Most instructions in HARP can

be predicated. The HARP compiler uses LLVM's If-conversion pass for predicating branches. The If-conversion pass is target independent like other code generation algorithms and calls functions in the target to check legality, profitability and predication. The HARP backend adds support for these functions.

- **Decision framework to choose control divergence management technique:**

Both hardware stack based control divergence management and predication have unique advantages and disadvantages. On a unanimous branch, the hardware stack based control divergence management will execute only one direction of the branch. However, predication will execute both directions of the branch, slowing down performance. On a nonunanimous branch, both directions of the branch will be executed by the hardware stack based mechanism as well as predication. However, it is beneficial to generate predicated code for such branches, because hardware stack based control divergence management requires additional instructions such as `jmp`, `split`, `join`. The HARP compiler implements a decision framework for choosing one control divergence management technique over the other driven by static analysis and dynamic profile information. The static analysis guided decision framework follows def-use chains of operands in the conditional branch to see if it is dependent on a thread variant value, depending on which, it classifies the conditional branches. Dynamic profile guided decision framework determines the nature of the conditional branches in the instrumented run, and uses this information to classify the conditional branches. Based on the analyses the decision framework inserts `split` and `join` instructions for branches that are likely to be unanimous, and performs If-conversion in other cases.

The thesis is organized as follows. Chapter 2 describes the details of the HARP architecture along with the two control divergence management techniques it supports. It also gives a brief overview of the LLVM compiler infrastructure on which the HARP compiler is built. Chapter 3 describes the HARP backend design in detail. Chapter 4 discusses the compiler support required for control divergence management. In Chapter 5, the test framework for evaluating code generation and experimental results of the different divergence management techniques are presented. Finally, Chapter 6 provides some concluding remarks and Chapter 7 presents avenues for future work.

CHAPTER II

BACKGROUND

The main focus of the thesis is in building a compiler for the HARP architecture. This chapter describes the aspects of the HARP project relevant for HARP compiler development. We also present details of the LLVM compiler infrastructure, which is utilized to construct the HARP compiler.

2.1 The HARP Project

The Heterogeneous Architecture Research Prototype (HARP) provides a parametric framework to generate HARP code and associated microarchitecture implementations. HARP was designed for design exploration of PIM architectures. An extensive number of tools have been developed for the HARP project ranging from a C++ based hardware design library to a software toolchain consisting of an assembler, linker and emulator [16]. The entire HARP toolchain is parametric, supporting customizable architecture features such as the number of cores, number of registers, word length etc. Details of the HARP architecture, including the HARP ISA, execution model, and the support for control divergence management are discussed in the next sections.

2.1.1 HARP Architecture

The HARP baseline architecture is shown in Figure 1. It consists of multiple SIMT cores placed in a 3d stacked DRAM. The SIMT cores are called Harmonica. Each Harmonica core consists of multiple scalar pipelines. The scalar pipelines within a Harmonica core are driven by a single PC, and execute in SIMT fashion. A data parallel function, when launched on the HARP architecture executes in multiple groups of threads called warps. The threads in a warp are mapped to a Harmonica core,

and execute in SIMT style. The number of Harmonica cores, scalar pipelines are reconfigurable owing to the parametric HARP framework.

2.1.2 HARP ISA

The HARP ISA drives the SIMT cores of the HARP Architecture. The properties of HARP ISA are described below:

- **Simple:** HARP ISA is a simple, RISC like ISA. It consists of only 64 opcodes, which include both register and immediate operand versions. The instructions are written with destination register first. The ISA consists of general purpose registers and predicate registers. General purpose register names are prefixed with % symbol and predicate registers are prefixed with @ symbol. Example :

```
addi %r0, %r0, #3
```

- **SIMT oriented:** HARP ISA is inherently SIMT. The instruction stream can represent execution by multiple parallel threads driven by a single PC, executing on independent register sets. The ISA provides instructions important for SIMT control :
 - Instructions to handle control divergence
 - Instructions to launch warps, lanes
 - Barriers
- **Fully predicated:** Most HARP instructions can be predicated. A predicated instruction is executed only when the predicate controlling the instruction is true. The ISA provides predicate registers and instructions to manipulate them. A predicated instruction is written as :

```
@p0? addi %r0, %r0, #3
```

- **Customizable:** The number of general purpose registers, predicate registers, and instruction encodings are customizable. An Architecture Identifier string (ArchID) specifies the customizable architecture features, and drives the parametric toolchain consisting of the assembler, linker, emulator. Example: ArchID: 4w8/8/16/16:

Field	Meaning
4	4-byte (32-bit) registers and addresses
w	Word-based (32-bit) fixed-width instruction encoding
8	8 general-purpose registers per lane
8	8 predicate registers per lane
16	16 SIMD lanes
16	16 warps (thread groups)

2.1.3 HARP Toolchain and Execution Model

The HARP toolchain consists of a set of tools that convert application programs into a HARP binary. Figure 4 shows the schematic diagram of the HARP toolchain.

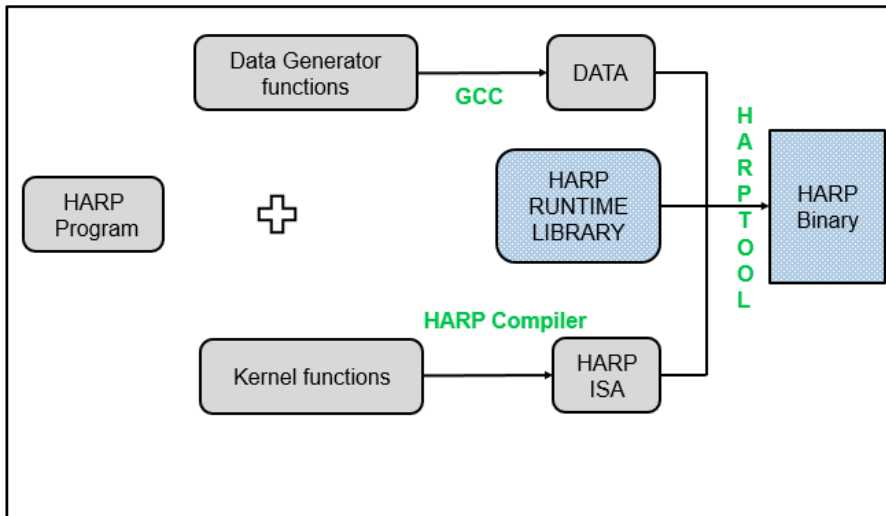


Figure 4: HARP toolchain

Application programs for HARP, are currently written in C, with HARP specific extensions. New HARP specific attributes are added to C language which helps

identify kernel function, `lane Id` and `warp Id`. These attributes aid the HARP compiler in code generation. The application programs are written using the API provided by the HARP runtime library for spawning warps, lanes, etc.

Application programs written for HARP consist of kernel functions and functions to generate and setup input data. The data-generator functions are compiled using a host compiler like `gcc`, and execute to produce input data which will be later used by the kernel functions. Kernel functions are compiled by the HARP compiler. `harptool` is a set of tools consisting of assembler, linker and emulator which take HARP ISA and input data, and creates the HARP binary, which can be executed on the HARP emulator.

Figure 5 shows a schematic diagram of a typical execution of the application program written for HARP. Execution of the kernel function begins with the `start()` function. The `start()` function calls the HARP Runtime library's `spawn_warp()` function, passing a function pointer to the function that all warps should begin execution with. This function calls the HARP Runtime library's `call_par()` function, which launches parallel threads of the warp. All lanes in the warp begin SIMT style execution of the application code which is passed as a function pointer to `call_par()`.

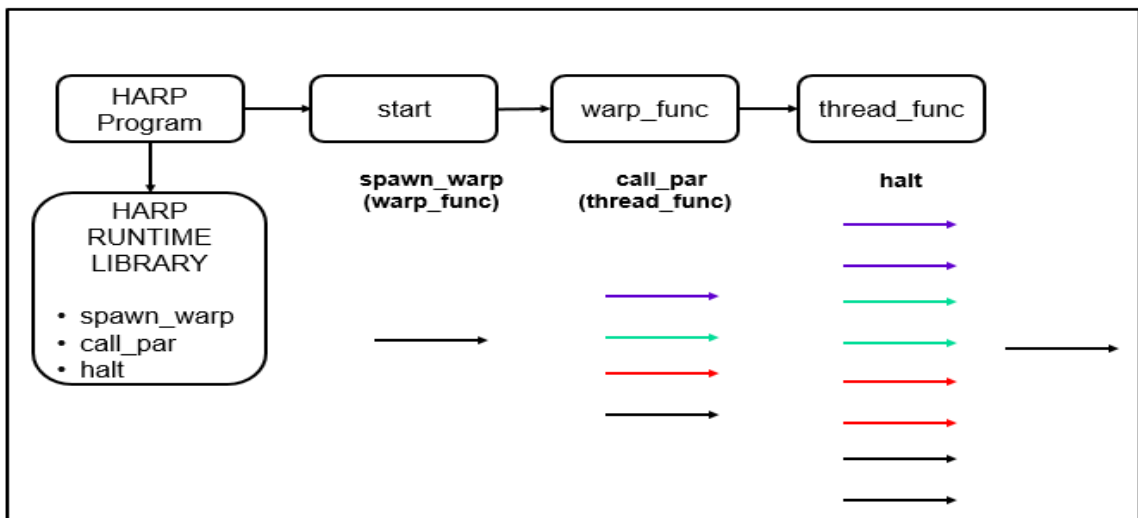


Figure 5: A typical execution of application program on HARP

2.1.4 Control divergence management in HARP

In SIMT execution, when lanes in a warp executing in lockstep encounter a conditional branch, some lanes can take a direction different from the rest. This is called control divergence. On a divergent branch, the execution of both sides of the branch are serialized until a reconvergence point is reached, where all lanes resume execution in lockstep. Efficient control divergence management is extremely important for data parallel workloads, because of its large impact on performance and energy. It is a widely researched area, with long established techniques and optimizations in both hardware and software [13, 14, 25, 19]. Traditional solutions to branch divergence management include *hardware stack based divergence management* and *predication*. The HARP architecture supports both these techniques and are discussed in the next sections.

2.1.4.1 Split-Join Infrastructure

The HARP architecture provides hardware stack based divergence management via `split` and `join` instructions. The `split` instruction is inserted before a conditional branch, and the `join` instruction at the reconvergence point. The `split` instruction is predicated with the same predicate as the conditional branch. On encountering a `split` instruction, PC (program counter) and the active mask are pushed onto a hardware stack, threads with true predicate are allowed to continue, and the other threads are masked. When the `join` instruction is encountered, the PC and the active mask are popped from the hardware stack, and previously masked lanes are now allowed to continue. When the `join` instruction is encountered the second time, control flow falls through and all threads start executing in lockstep again. Aggressive compiler optimizations and certain control flow structure properties make automatic split-join insertion by the compiler complicated. The HARP compiler performs multiple transformations on the control flow graph of the kernel to make it amenable to

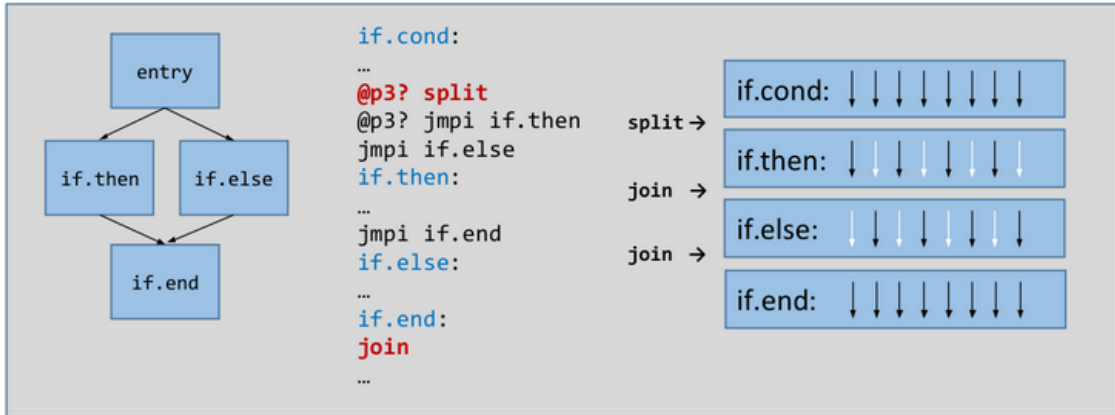


Figure 6: Split-Join Example

split-join insertion. This is discussed in detail in Section 4.1.

2.1.4.2 Predication

Predicated execution involves the conditional execution of instructions. Instructions are executed normally when the predicate value is true. When the predicate value is false, the instruction cannot update state. The compiler generates code for predicated execution by a technique called as if-conversion [10, 20]. It eliminates conditional branches altogether by converting control dependencies into data dependencies. All non loop conditional branches can be if-converted. Predication has been widely discussed in the literature. It was traditionally used in the CPU world to predicate hard to predict branches, and in the VLIW (Very Large Instruction Word) world, to achieve full VLIW length instruction packing.

HARP provides several predicate manipulation instructions and predicate registers for predicated execution. Most instructions in HARP can be predicated. HARP compiler support for predication is discussed in detail in Section 4.2.

The following figure shows an example of a predicated if-then statement.

```

                                ld %r0, %r0, #0
                                subi %r0, %r0, #1
                                isneg @p3,%r0
                                notp @p2, @p3
if (b[lid] > 0) {                @p2? ld %r0, %fp, #-12
    b[lid]++;                    @p2? shli %r0, %r0, #2
}                                @p2? ld %r1, %fp, #-16
                                @p2? add %r0, %r1, %r0
                                @p2? ld %r1, %r0, #0
                                @p2? addi %r1, %r1, #1
                                @p2? st %r1, %r0, #0

```

2.1.4.3 Handling branch divergence due to loops

Divergent loops are those in which different lanes of the warp can execute different number of iterations. The immediate post-dominator (IPDOM) of a loop is at the loop exit block. Inserting a `split` instruction before the loop conditional branch and a `join` instruction at the IPDOM can result in multiple splits with a single join. This can leave unpopped entries in the hardware stack, and can result in incorrect results. Different hardware solutions are possible to handle multiple splits and a single join. However in the current version of HARP, divergent loops are addressed by using the `lane_or()` function of the HARP Runtime Library. The `lane_or()` function does a logical OR of the loop predicate for all lanes in the warp. Therefore, the loop is entered in all lanes even if only one lane has a `true` loop predicate. The loop is executed in every lane for the same number of iterations, and therefore no control divergence management is needed. An example is shown below:

```

while (lane_or(w_id, l_id, !finished)) {
    if (!finished) {
        if (*begin == val) {
            finished = 1;
            found = 1;
        } else {
            finished = (begin + 1) == end;
        }
        begin++;
    }
}

```

2.2 LLVM compiler infrastructure

The HARP compiler is built on top of the LLVM compiler infrastructure. The LLVM compiler infrastructure [18] is an open-source compiler framework, which provides high quality tools needed to build compilers. It consists of LLVM Core libraries that provide target independent optimizations and code generation support for varied targets such as X86, MIPS, SPARC etc. Clang [1] is the most widely used compiler frontend with LLVM-based compilers. It supports high level languages such as, C, C++, Objective C and OpenCL. Figure 7 shows the schematic diagram of the LLVM toolchain.

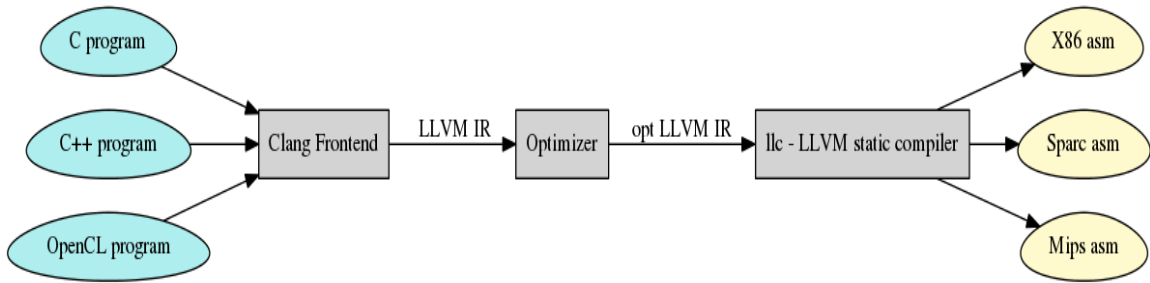


Figure 7: Basic LLVM Architecture

As shown in Figure 7, the Clang frontend lowers programs written in high level language to LLVM IR, LLVM’s intermediate representation. The `opt` tool, runs several scalar, loop and interprocedural optimizations transforming the LLVM IR generated by the frontend to a more efficient implementation. Finally the `llc` tool, transforms the optimized LLVM IR into the target ISA. The following sections describe the details of the intermediate representation, design philosophy and community support of the LLVM compiler infrastructure.

2.2.1 LLVM Intermediate Representation (IR)

LLVM IR is the intermediate representation used by the different tools in the LLVM compiler infrastructure. The LLVM IR has low-level instruction set, similar to a

RISC-style machine. However, the LLVM IR supports a high-level type system similar to a high-level programming language like C so that sufficient information is available for optimizers to aggressively optimize. The LLVM IR is in SSA (Single-Static Assignment) [11] form with infinite number of virtual registers. SSA form allows only single definitions of variables, with every variable defined before being used. For merging values from multiple control flow paths, phi (Φ) function is used. The properties of SSA form aid several compiler analyses and transformation passes. Following is a program in LLVM IR to compute factorial, illustrating the low-level IR, high-level types and the phi function.

```
define i32 @fact(i32 %n) #0 {
entry:
  br label %for.cond
for.cond:                                ; preds = %for.body, %entry
  %fact.0 = phi i32 [ 1, %entry ], [ %mul, %for.body ]
  %i.0 = phi i32 [ 1, %entry ], [ %inc, %for.body ]
  %cmp = icmp sle i32 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:                                ; preds = %for.cond
  %mul = mul nsw i32 %fact.0, %i.0
  %inc = add nsw i32 %i.0, 1
  br label %for.cond
for.end:                                  ; preds = %for.cond
  ret i32 %fact.0
}
```

2.2.2 LLVM Design Philosophy

LLVM is highly modular. It defines components clearly via separate libraries. The entire infrastructure is written keeping reusability in mind. LLVM is written in C++, and relies heavily on polymorphism and inheritance for adding new extensions. This design philosophy makes it easy to extend LLVM, be it new backends or new register allocation algorithms or new alias analyses. For this reason, LLVM has become widely popular among researchers and industry wide compiler writers, and is therefore, the defacto choice for the HARP compiler.

2.2.3 Documentation and Community

The LLVM compiler infrastructure has extensive documentation for users as well as developers. Detailed documentation on LLVM IR [4], target independent code generator [5], writing a new backend [8] and extending Clang [9] are publicly available. LLVM also has a large community of industry and research compiler writers. Several external tutorials and documentation about the LLVM compiler infrastructure exist [7]. These resources along with the LLVM forum for discussion [3] were invaluable resources for the HARP compiler development.

CHAPTER III

THE HARP COMPILER

This chapter discusses the design and implementation details of the HARP compiler. The chapter first presents an overview of the components of the HARP compiler, and then discusses the details of code generation, which is a significant portion of this thesis.

3.1 Overview

The schematic diagram of the HARP compiler is shown in Figure 8. Like other compilers, the HARP compiler has the following components:

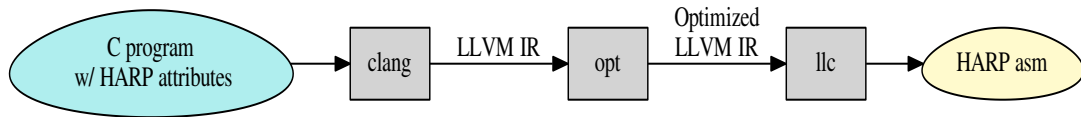


Figure 8: HARP compiler

- **Frontend:** Clang [1] is used as the compiler frontend. The Clang frontend transforms the source program into an LLVM intermediate representation (LLVM IR), flagging syntax and semantic errors. New function and parameter attributes have been added as extensions to the C language, to identify kernel function, `lane Id` and `warp Id`. These attributes are lowered into metadata which will be used by the code generator. An example program with newly added attributes is shown below:

```
__attribute__((harp_kernel)) int scan_par(  
    __attribute__((warp_id)) int w_id,  
    __attribute__((lane_id)) int l_id, int val) {  
    ...  
}
```

- **Optimizer:** LLVM’s `opt` tool is used as the optimizer. The optimizer runs several scalar and loop optimizations transforming the LLVM IR generated by the frontend into a more efficient implementation.
- **Backend:** The backend is responsible for transforming LLVM IR into HARP ISA. The HARP backend targets 2 popular HARP ISA configurations, `harp32` and `harp64`. The HARP backend cannot target all possible combinations of configurations that the customizable HARP ISA can support. This is because LLVM builds static data structures describing the target’s properties such as register set, instruction mappings, etc., for code generation. This design is not suitable for a customizable instruction set like HARP, which requires target information to be populated on-the-fly based on the `ArchId` switch. Therefore, only `harp32` and `harp64` configurations are currently supported. Table 1 shows the distinctive details of these 2 configurations.

Table 1: `harp32` and `harp64` target configurations
All sizes and alignments shown are in bytes.

Target feature	<code>harp32</code>	<code>harp64</code>
Word length	4	8
Number of General Purpose Registers	8 (R0 - R7)	32 (R0 - R31)
Number of Predicate Registers	8 (P0 - P7)	32 (P0 - P31)
Register Size	4	8
Caller Saved Registers	R0 - R3	R0 - R15
Callee Saved Registers	R4 - R7	R16 - R31
Argument passing Registers	R0 - R3	R0 - R15
Return value Register	R0	R0
Size of char	1	1
Alignment of char	4	8
Size of short, int, float, double	4	8
Alignment of short, int, float, double	4	8

A significant amount of work in this thesis was devoted in building the HARP backend, details of which are discussed in the next section.

3.2 The HARP backend

The HARP backend goes through several phases for transforming the LLVM IR into HARP ISA implementation. Figure 9 shows a schematic diagram of the organization of the HARP backend. It has typical code generation phases like instruction selection, register allocation, etc., and some phases specific to HARP such as Peephole Codegen and a Control Divergence Pass. The next sections describe the design of interaction between LLVM and the HARP backend followed by detailed description of the code generation phases.

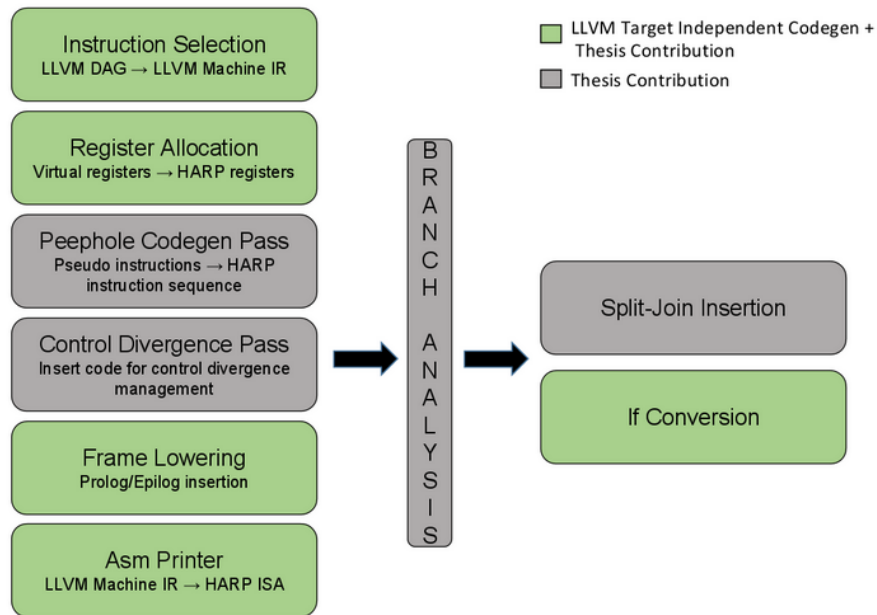


Figure 9: HARP backend

3.2.1 LLVM Target Independent Code Generation-HARP backend Co-design

The LLVM Codegen library implements target independent code generation algorithms like instruction selection, register allocation, etc. These algorithms query target specific properties via pre-defined interfaces. LLVM defines top-level abstract classes to describe the target machine, instruction set, register information, etc. These

classes define virtual functions that query target specific features. The backend implements classes that derive from the LLVM abstract classes, overriding the virtual functions as per the target. This design allows maximum code reuse and simplifies the addition of new backends. Figure 10 shows the classes implemented by the HARP backend, to describe HARP specific features.

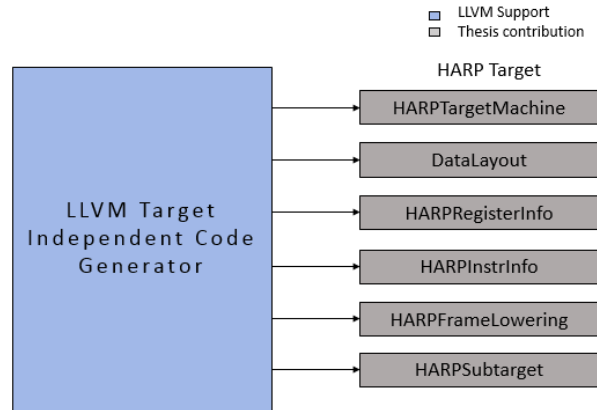


Figure 10: HARP Target Classes

Tablegen: Most of the target specific features in the HARP backend are written using the Tablegen [6] format. Target specific properties tend to have a large number of records with repeated information, Tablegen provides a concise way to describe these properties. Tablegen follows object-oriented principles (OOP) allowing refactoring of common code via inheritance. LLVM defines top-level abstract classes like `Register` and `Instruction` in the abstract target description. The backends derive from these top-level classes adding/overriding target-specific properties.

The HARP backend describes its instruction set, register set and calling conventions in the Tablegen format. The `llvm-tablegen` tool reads these target descriptions and auto-generates code encapsulating target properties in C++ files, the auto-generated data structures, functions, classes are used in the HARP target classes shown in Figure 10.

3.2.2 Instruction Selection

During this phase, instructions in the LLVM IR are mapped to HARP instructions. There are several phases in Instruction Selection during which multiple classes in the LLVM target independent code generator interact with the HARP backend. Figure 11 shows the different phases of Instruction Selection, along with the interacting LLVM - HARP backend class hierarchy. The transformations during each of the phases, along with the LLVM - HARP backend interaction is described in detail below:

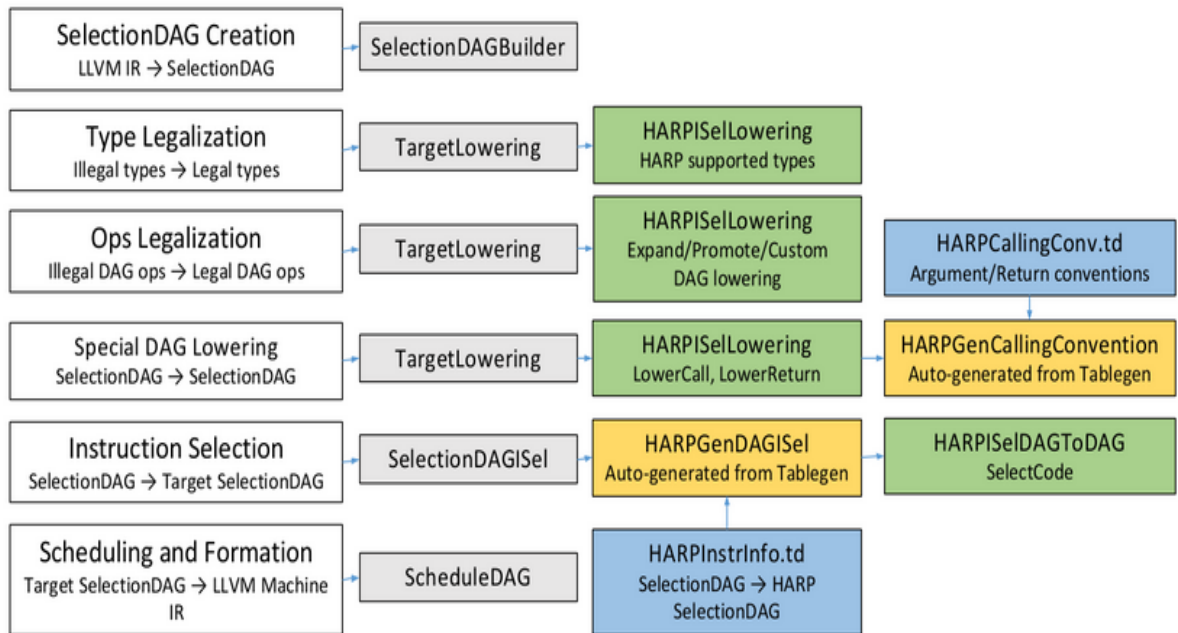


Figure 11: LLVM - HARP backend co-design for Instruction Selection

- **SelectionDAG Creation:** During this phase, LLVM's `SelectionDAGBuilder` transforms LLVM IR into a SelectionDAG. Nodes in the SelectionDAG represent LLVM IR operations. Edges may represent data flow or control flow. This representation is amenable to instruction selection algorithms which mainly perform DAG pattern matching.
- **SelectionDAG Lowering:** The SelectionDAG created can be illegal, consisting of unsupported types and operations on the target. During SelectionDAG

lowering, illegal SelectionDAG nodes are transformed to legal SelectionDAG nodes. There are multiple phases in SelectionDAG lowering, each phase queries target specific information via the virtual functions in the `TargetLowering` class. The HARP backend implements the `HARPISELowering` class which overrides these virtual functions providing a HARP specific implementation. This is explained in detail below:

- **Type Legalization:** The `HARPISELowering` class specifies what types HARP registers can hold. The target independent Type Legalizer queries this information via the `TargetLowering` interface, and transforms unsupported types into supported types via promoting/expanding the types.
- **Ops Legalization:** Not all LLVM SelectionDAG nodes are supported in the HARP backend. The `HARPISELowering` class specifies unsupported SelectionDAG nodes that can be transformed to supported SelectionDAG nodes, it also specifies how to do this transformation. The Ops Legalizer queries for such SelectionDAG nodes through the `TargetLowering` interface and performs the specified transformation. Following are some examples:
 - * **‘Expand’ Lowering:** When SelectionDAG nodes can be emulated with other SelectionDAG nodes, mapping for all SelectionDAG nodes are not supported by the backend. For example, LLVM defines `BR_CC` and `BRCND` to represent conditional branches. However, the HARP backend specifies mapping for only `BRCND` SelectionDAG node to HARP conditional branch. The `HARPISELowering` class specifies ‘Expand’ lowering for `BR_CC`, the Ops Legalizer then transforms a `BR_CC` SelectionDAG node into `BRCND` node.
 - * **‘Custom’ lowering:** SelectionDAG nodes on some targets need be

lowered in a target specific way. The `HARPISe1Lowering` class specifies ‘Custom’ lowering for such SelectionDAG nodes, and provides a target specific hook to lower them. Example, there is no direct mapping for a `GlobalAddress` DAG node in HARP, it has to be lowered to a sequence of HARP specific DAG nodes. Hence it is performed via ‘Custom’ lowering.

- **Special DAG Lowering:** Target specific lowering is required for some SelectionDAG nodes across all backends. The `TargetLowering` class defines virtual functions for lowering such SelectionDAG nodes. These virtual functions are redefined in the `HARPISe1Lowering` class. Example:
 - * **Lower Call:** The `HARPISe1Lowering` class redefines the `LowerCall` function in the `TargetLowering` class. The implementation replaces the LLVM `Call` DAG node with `jali` HARP DAG node. It also analyzes the arguments of the function call, and inserts code to pass arguments in registers or stack locations as per the HARP calling convention.
 - * **Lower Return:** The `HARPISe1Lowering` class redefines the `LowerReturn` function in the `TargetLowering` class. The implementation replaces LLVM `Return` DAG node with `jmp` HARP DAG node. It also analyzes the return value, and inserts code to copy return values to register or stack location as per the HARP calling convention.
- **Instruction Selection:** Once the SelectionDAG consists of only target supported operations and types, the target independent code generator maps LLVM DAG nodes to HARP DAG nodes by DAG pattern matching. The mapping between LLVM DAG nodes to HARP DAG nodes is specified in the HARP target description in Tablegen format. The

`llvm-tablegen` tool auto-generates the DAG pattern matching implementation from this target description in the `HARPGenDAGISel` class. This auto-generated class derives from the LLVM `SelectionDAGISel` class and overrides the `SelectCode` virtual function which drives the Instruction Selection.

Most of the mappings in the target description are one-to-one. Since the HARP ISA is RISC-like similar to the LLVM IR. However some mappings in the target description are more advanced:

- * **Immediates:** Immediate versions of instructions can only be used if the immediate fits into the instruction encoding. Therefore, for every mapping of SelectionDAG node to HARP instruction with an immediate operand, a predicate is attached to check if the immediate fits in the target encoding. The pattern-matching algorithm will not select the immediate version of the instruction if this predicate fails. If the immediate does not fit, the pattern-matching algorithm selects the register operand version of the HARP instruction. The immediates that do not fit in the instruction encoding, are broken into smaller parts and constructed via `shift` and `or` instructions into registers.
- * **SETCC SelectionDAG nodes:** HARP does not provide composite comparison instructions like `seteq`, `setlt` etc. The SETCC SelectionDAG nodes are emulated with SUB instruction and predicate test instructions `ISNEG`, `ISZERO`, `RTOP`. This necessitates a large number of one-to-many mappings in the target description files.
- **Scheduling and Formation:** Finally, the `DAGScheduler` class schedules the HARP Machine DAG and generates LLVM Machine IR.

Some SelectionDAG nodes cannot be lowered directly to HARP, because HARP instruction sequence representation cannot be expressed in LLVM Machine IR which is in SSA form. Such SelectionDAG nodes are mapped to a pseudo instruction in the target description. The pseudo instructions are lowered to HARP instructions by the *Peephole Code Generator* discussed in Section 3.2.6.

3.2.3 Register Allocation

During this phase, the virtual registers in the LLVM IR are mapped to physical registers supported by the target. The LLVM Codegen library implements greedy register allocator [2] as the default. Like other target independent code generation algorithms, the greedy register allocation implementation is target independent. It queries target specific information via virtual functions in the abstract `TargetRegisterInfo` class. The HARP backend implements the `HARPRegisterInfo` class which overrides these virtual functions providing a HARP-specific implementation. Figure 12 shows a schematic diagram of the class hierarchy in LLVM - HARP backend co-design for register allocation.

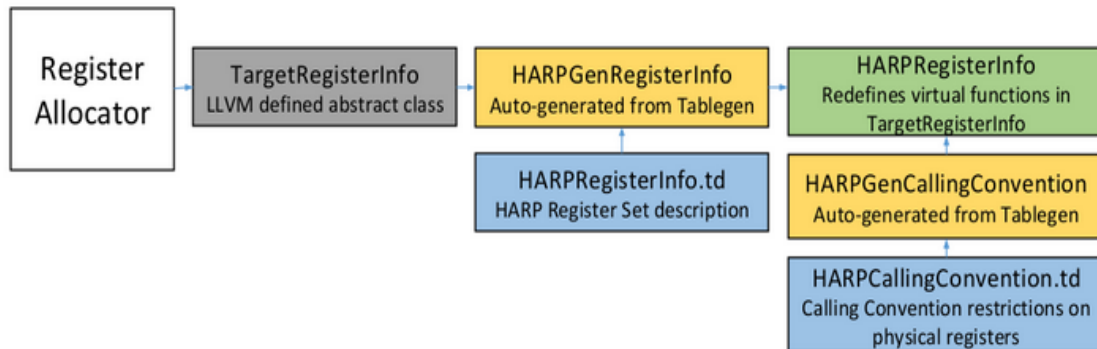


Figure 12: LLVM - HARP backend co-design for register allocation

The HARP register set is described in Tablegen. The register set description includes detailed information about the supported register classes, number of registers,

dwarf numbers for registers etc. From this description, the `llvm-tablegen` tool generates the `HARPGenRegisterInfo` class. This class derives from the `TargetRegisterInfo` class, and encapsulates details of the HARP register set. The `HARPRegisterInfo` class derives from the auto-generated `HARPGenRegisterInfo` class and overrides the virtual functions in `TargetRegisterInfo` inherited via this derivation. Specifically, it provides HARP specific restrictions on register allocation, support for spilling and frame index elimination. These are described below:

- **Restrictions on Register Allocator:** Not all registers in the register set are available for allocation. The register allocator has to respect target specific restrictions during allocation. Following are some of the restrictions in the HARP backend:
 - **Reserved Registers:** The HARP backend has dedicated registers for stack pointer, frame pointer and link register, they serve specific purposes and are reserved from register allocation. Along with these registers few other general purpose and predicate registers are reserved for code generation workarounds.
 - **Calling Convention Restrictions:** The HARP calling convention divides the first half of the register set into callee saved registers and the second half of the register set into caller saved registers. This information is specified in the `HARPCallingConvention` target description in Tablegen format. The `HARPRegisterInfo` class uses the auto-generated arrays from the calling convention target description to specify these restrictions.
- **Support for spilling:** The register allocator spills virtual registers to the stack when enough physical registers are not available. The `HARPRegisterInfo` class defines target hooks to load register from a stack slot, store register to a stack slot, etc., to support spilling. These functions are called by the spiller via the

TargetRegisterInfo interface.

- **Frame index elimination:** All accesses to stack locations are represented via abstract frame indexes in the LLVM IR. These abstract stack references should be translated to machine specific stack references after register allocation. The HARPRegisterInfo class overrides the eliminateFrameIndex virtual function to implement this translation. This function computes the stack offset of the frame index and replaces it with %fp + stack_offset. Following is an example:

```
STa %R32_0, <fi#1>, 0; --> STa %R32_0, %FP32, -16;
```

3.2.4 Frame Lowering

The LLVM target independent code generation phases call the virtual functions in the TargetFrameLowering abstract class to query information about the stack frame layout of the target. The HARP backend implements the HARPFrameLowering class which derives from the TargetFrameLowering class and overrides information about the stack frame layout as per HARP. Specifically, the HARPFrameLowering class does the following:

- Specifies the stack frame should grow down in HARP.
- Specifies the alignment of stack pointer, 4 bytes in harp32 and 8 bytes in harp64.
- Inserts Prolog/Epilog: The prolog is responsible for setting up the stack frame on function entry, and the epilog is responsible for deleting the stack frame on function exit. The HARPFrameLowering class defines target hooks to insert the following instruction sequences as prolog and epilog:

prolog:

```
st %fp, %sp, #-4
ori %fp, %sp, #0
subi %sp, %sp, #stacksize
```

epilog:

```
ori %sp, %fp, #0
ld %fp, %fp, #-4
```

Prolog is executed on function entry, and epilog on function exit. The frame pointer (`%fp`) points to the bottom of the call stack, and the stack pointer (`%sp`) points to the top of the call stack. On function entry, prolog code saves the frame pointer on the stack, and moves the frame pointer to the top of the stack. The stack pointer is incremented to the size of the callee's stack frame. On function exit, the epilog code moves the stack pointer to the bottom of the stack, and reloads the frame pointer from the saved location on the stack, thereby deleting the call frame.

3.2.5 Asm Printer

During this phase, the LLVM Machine IR is translated to HARP assembly text. The HARP backend implements the `HARPAsmPrinter` class that derives from the abstract `AsmPrinter` class, to specify HARP specific assembly printing. The assembly output of every HARP instruction is specified in the LLVM DAG to HARP DAG mapping target description. From this specification, `llvm-tablegen` tool generates functions and data structures to print out the final HARP assembly.

3.2.6 Peephole Code Generator

Mapping of some SelectionDAG nodes into HARP instruction sequences cannot be represented in SSA form. Such SelectionDAG nodes are mapped to pseudo instructions during Instruction Selection. The Peephole Code Generator pass lowers such pseudo instructions into equivalent HARP instruction sequences. It runs after register allocation, when the Machine IR is no longer in SSA form, and can therefore replace pseudo instructions with their HARP equivalents. Table 2 shows the pseudo instructions and their equivalent HARP instruction sequences that cannot be represented in SSA form.

Table 2: Pseudo instructions and their equivalent HARP instruction sequences that cannot be represented in SSA form

Pseudo instruction	HARP instruction sequence
<code>zero_extend %rX, @p0</code> zero extend from predicate register to gpr	<code>notp @p1, @p0</code> <code>@p0 ? ldi %rX, #1</code> <code>@p1 ? ldi %rX, #0</code>
<code>select %rX, %condReg, %trueReg, %falseReg</code> If <code>condReg == 1</code> , <code>%rX = %trueReg</code> else <code>%rX = %falseReg</code>	<code>notp @p1, @p0</code> <code>@p0 ? ori %rX, %trueReg, #0</code> <code>@p1 ? ori %rX, %falseReg, #0</code>
<code>loadImm %p0, #1</code> load immediate 1 into predicate register	<code>xorp @p0, @p0, @p0</code> <code>notp @p0, @p0</code>

Instruction sequences in Table 2 have multiple writes to the same register, therefore they cannot be represented in SSA form. The Peephole Code Generator traverses the Machine IR, looking for pseudo instructions, and replaces them with their HARP counterparts.

3.2.7 Control divergence Pass

Control divergence in the HARP architecture can be managed via *hardware stack based control divergence management* or *predication*. Both of these techniques require code generation support from the HARP compiler. The analyses and transformations needed to support the control divergence management techniques along with the decision framework to choose between the two are described in detail in Chapter 4.

3.3 Integration of HARP backend

In order to integrate the HARP backend with the rest of the LLVM Infrastructure, the LLVM Build System and the Clang frontend have to be extended.

3.3.1 LLVM Build System

Like other components of the LLVM compiler infrastructure, the LLVM Build System can be seamlessly extended for a new backend. The LLVM Build System has a top-level `configure` script that initializes the build system. The HARP target has to be

added to the `configure` script. The build system automatically generates Makefiles that build the HARP backend library and link it with the rest of the Codegen library, which drives the `llc` tool.

3.3.2 Cross Compilation Support in Clang

The LLVM IR generated by the Clang frontend is not completely target agnostic. The Clang frontend requires certain information about the target such as sizes and alignments of fundamental data types. By default, the Clang frontend generates LLVM IR for the target on which it is executed. However, for our purposes, we cross-compile on a host CPU like x86 to generate code for the HARP target. To support cross-compilation, the Clang frontend accepts a string called `Target Triple` which uniquely identifies the target. Based on the `Target Triple` switch, Clang instantiates appropriate `TargetInfo` class that specifies target properties needed for LLVM IR generation. `harp32` and `harp64` are added as target triples for the 32-bit and 64-bit configurations of the HARP target. The sizes and alignments of fundamental data types of the HARP target as described in Table 1 are added to the HARP target description in Clang.

CHAPTER IV

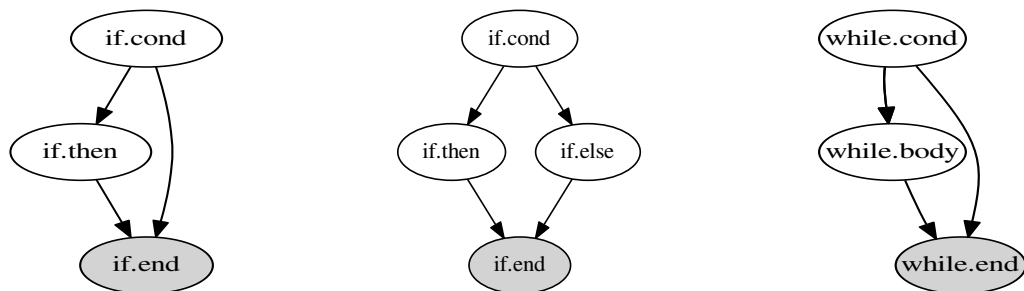
COMPILER SUPPORT FOR CONTROL DIVERGENCE MANAGEMENT

The control divergence management techniques supported by the HARP architecture are discussed in Section 2.1.4. This chapter describes the analyses and transformations required for code generation of the control divergence management techniques. It also describes a decision framework guided by static analysis and dynamic profile information for choosing between the control divergence management techniques.

4.1 Split-Join Infrastructure

The HARP backend inserts `split` and `join` instructions to handle control divergence for all potentially divergent conditional branches. Every conditional branch in the kernel is a potential divergent branch. Therefore the HARP backend inserts a `split` instruction before every conditional branch. The `join` instruction should be inserted at the nearest reconvergence point so that the warp runs at 100% utilization as early as possible. The nearest reconvergence point for a conditional branch is its immediate post-dominator (IPDOM).

A post-dominator is defined as follows: A basic block X post-dominates basic block Y (written as $X \text{ pdom } Y$) if and only if all paths from Y to the exit node go through X . A basic block X , distinct from Y , immediately post-dominates basic block Y if and only if $X \text{ pdom } Y$ and there is no basic block Z such that $X \text{ pdom } Z$ and $Z \text{ pdom } Y$. Figure 13 shows immediate post-dominator marked on some common control flow structures.



(a) CFG for if-then

(b) CFG for if-else

(c) CFG for while

Figure 13: Control flow graph marked with IPDOM for common control flow structures

4.1.1 Factors to be addressed for split-join insertion

Aggressive compiler optimizations and certain control flow structure properties make split-join insertion complex. The HARP backend performs multiple transformations on the control flow graph of the kernel to make it amenable for split-join insertion. It has to address the following factors while inserting split-join instructions:

- Presence of loops with multiple backedges:** For loops with multiple backedges, the IPDOM of the loop conditional branch is the block containing the conditional branch itself. The straight-forward approach of inserting splits before conditional branches and a join at the IPDOM will produce incorrect results in this case. A new IPDOM has to be created such that it does not dominate the loop header in addition to post-dominating it. The HARP backend therefore, transforms loops with multiple backedges to a single backedge. Algorithm 1 describes this in detail. Figure 14 shows an example of the `binsearch` kernel where this transformation is applied. Figure 14a shows the unoptimized control flow graph of the `binsearch` kernel. After standard compiler optimizations such as redundancy elimination and unreachable block elimination, control flow graph changes to Figure 14b. The optimized control flow graph with multiple

backedges is transformed to single backedge as shown in Figure 14c.

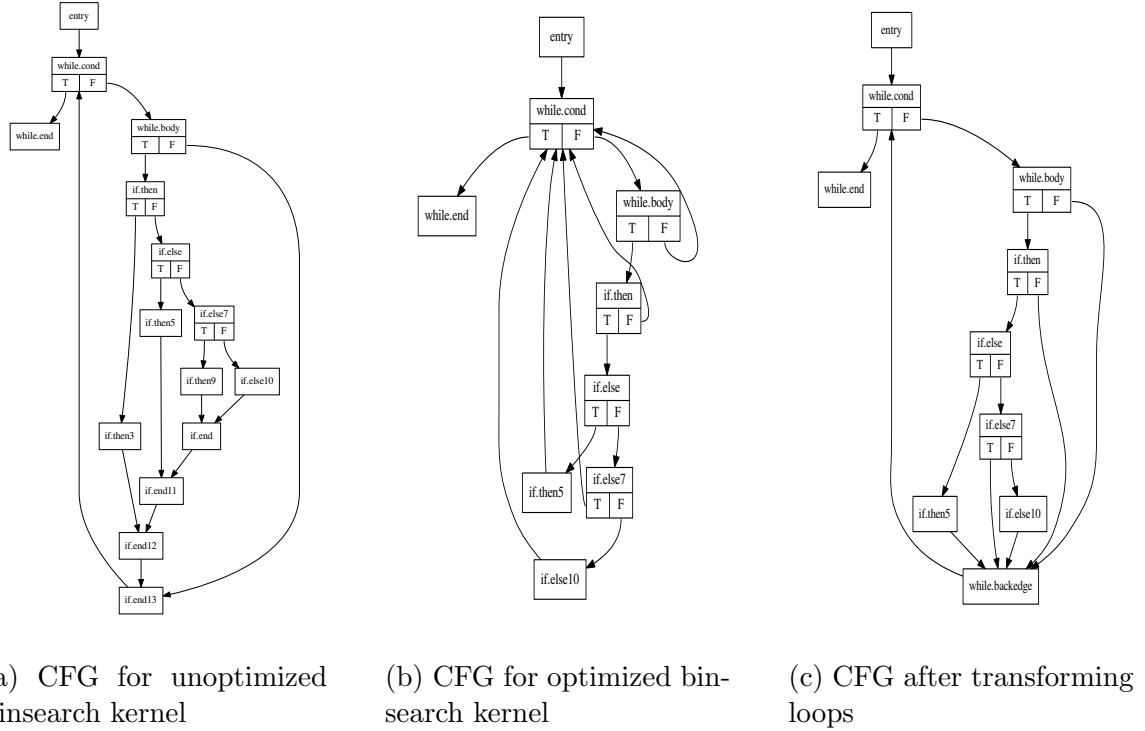
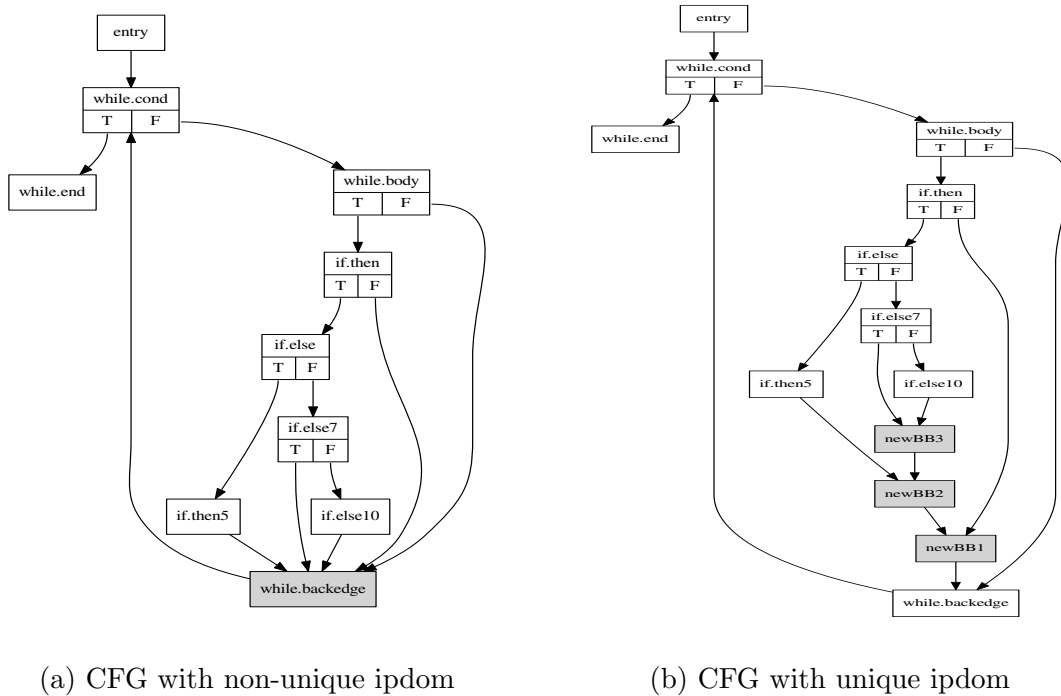


Figure 14: Control flow graph of the binsearch kernel during different stages of compilation

Fig.14a shows the unoptimized cfg of the binsearch kernel. Fig.14b shows the cfg after standard compiler optimizations such as redundancy elimination and unreachable block elimination. Fig.14b shows the transformed cfg after removal of multiple backedges.

- Presence of nested control flow with common IPDOM:** Nested control flow structures can have a common IPDOM. For the correct operation of `split` and `join` instructions, the HARP backend modifies the control flow graph to have a unique IPDOM for every conditional branch. This is needed because there is no way to associate a `join` instructions with its corresponding `split` instruction. Inserting `join` instruction at the common IPDOM for every conditional branch can result in executing more `join` instructions than necessary. Figure 15 shows an example. The conditional branches at `if.then`, `if.else`



(a) CFG with non-unique ipdom

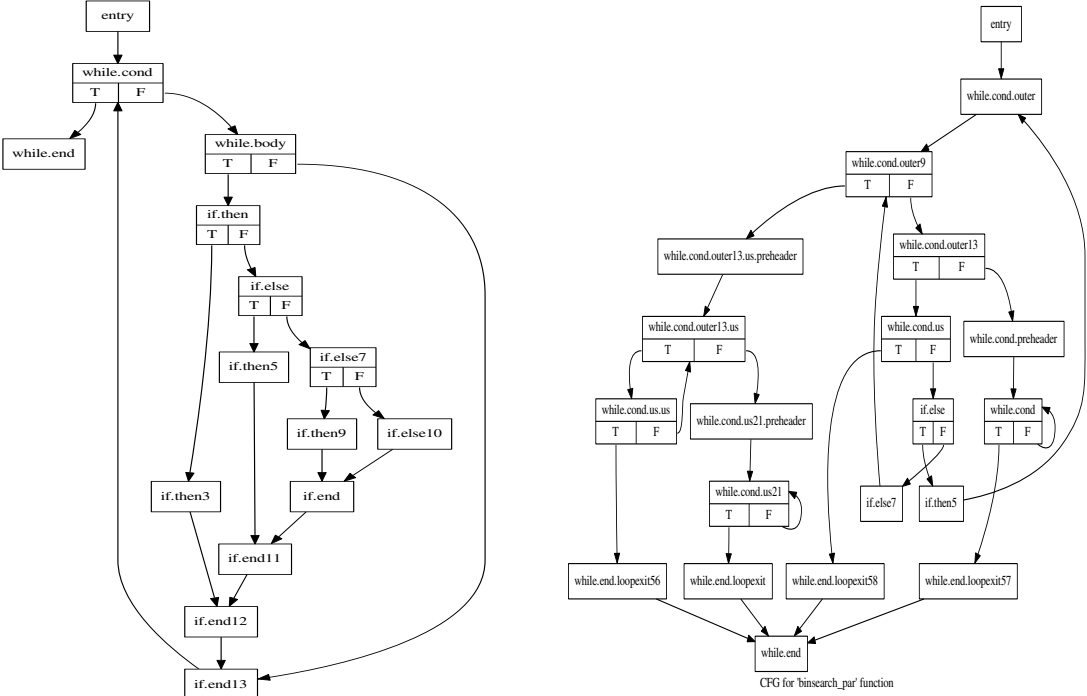
(b) CFG with unique ipdom

Figure 15: Control flow graph of the binsearch kernel marked with ipdom. Fig.15a shows the cfg with a common IPDOM. Fig.15b shows the transformed cfg with a unique IPDOM.

and `if.else7` have the IPDOM at `while.backedge`. Inserting join instructions for all of them at the common IPDOM `while.backedge` can lead to incorrect results. If the `if.then` branch is not taken by any of the lanes, only one join instruction needs to be executed, however the IPDOM `while.backedge` consists of 3 join instructions, as it is a common IPDOM for other nested conditional branches as well. Executing more than the required join instructions, leads to stack underflow. Therefore, the control flow graph is transformed such that every conditional branch has unique IPDOM as shown in Figure 15b.

- **Effect of aggressive compiler optimizations:** Aggressive optimizations transform the control flow graph in Figure 16a to Figure 16b. In Figure 16b, the basic block `if.else` has its IPDOM at `while.cond.outer9`. However the

IPDOM not only post-dominates, but also dominates the `if .else` block. Inserting `join` instruction at its IPDOM, will imply executing `join` instruction before the `split` instruction, which can lead to incorrect results. For a control flow graph to be amenable to split-join insertion, the IPDOM of a conditional branch should not dominate it. Having a structured control flow graph will ensure this. However transforming an unstructured control flow graph to a structured control flow graph leads to a bloat in dynamic instruction count which results in performance degradation. Figure 17 shows the increase in dynamic instruction count of single-threaded programs in the `harpbench` microbenchmark with the unstructured to structured transformation.



(a) CFG for unoptimized binsearch kernel (b) CFG for aggressively optimized binsearch kernel

Figure 16: Control flow graph of binsearch kernel after aggressive optimizations . Fig.16a shows the cfg with the unoptimized binsearch kernel. Fig.16b shows the cfg for an aggressively optimized binsearch kernel.

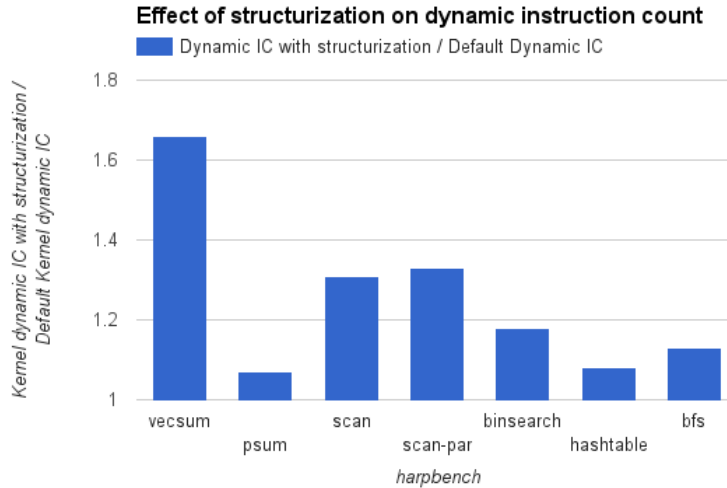


Figure 17: Dynamic instruction count bloat with unstructured to structured optimization

We do not need structured control flow graph. We require the IPDOM of conditional branches not to dominate them. Applying the unstructured to structured control flow graph transformation is unnecessary and detrimental to performance. Therefore, the HARP compiler turns off optimizations in the LLVM optimizer that can render the control flow graph intractable for split-join insertion. Figure 18 shows how the kernel dynamic instruction count changes with aggressive optimizations turned off for single-threaded programs in *harpbench*.

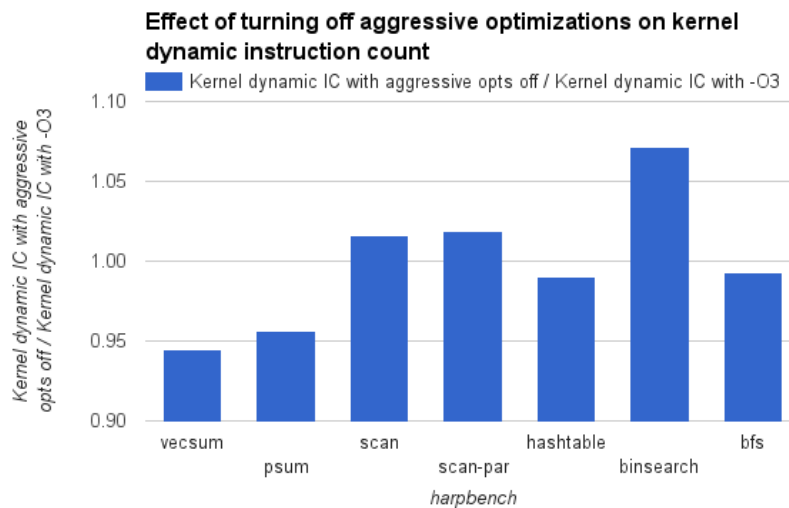


Figure 18: Effect of turning off aggressive compiler optimizations on kernel dynamic instruction count

Unlike unstructured to structured control flow graph transformation, turning off aggressive optimizations do not have a detrimental effect on the dynamic instruction count of programs in the `harpbench` microbenchmark suite. In fact, some programs benefit from turning off aggressive optimizations.

4.1.2 Algorithm for split-join insertion

The HARP backend inserts split-join instructions by addressing the challenges discussed in Section 4.1.1. The algorithm for inserting split-join instructions is described as follows.

- Algorithm 1 is applied to transform loops with multiple backedges to a single backedge. The algorithm creates a new loop exit block at the bottom of the loop. All backedges are then made to point to the newly created block instead of the loop header. Then a single backedge is added from the newly created block to the loop header. PHI instructions are updated due to the changes in the control flow graph, as per Algorithm 2.
- Algorithm 3 is applied to insert `split` and `join` instructions. For every conditional branch, the algorithm inserts `split` instruction before the conditional branch, and `join` instructions at the IPDOM. If a `join` instruction was already inserted at the IPDOM, a basic block is created to be the new IPDOM and the `join` instruction is inserted in the new IPDOM.
- For newly created IPDOMs, Algorithm 4 is applied. In depth first order both directions of the conditional branch are traversed, replacing control flow edges going into the old IPDOM to the newly created IPDOM.
- Algorithm 2 is applied, to transfer PHI instructions from the old IPDOM to the new IPDOM due to the changes in the control flow graph.

Algorithm 1 Algorithm for transforming loops with multiple backedges

```
for  $L \in \text{loops}(\text{kernel})$  do
  if  $\text{numBackEdges}(L) > 1$  then
     $\text{newBB} \leftarrow \text{CREATEBASICBLOCK}()$ 
     $\text{INSERT}(\text{newBB}, \text{GETBOTTOMBLOCK}(L))$ 
    for  $bb \in \text{predecessors}(\text{header}(L))$  do
      if  $bb \in L$  then
         $\text{REPLACESUCCESSOR}(bb, \text{header}(L), \text{newBB})$ 
      end if
    end for
     $\text{ADDSUCCESSOR}(\text{newBB}, \text{header}(L))$ 
     $\text{TRANSFERPHIS}(\text{header}(L), \text{newBB})$ 
  end if
end for
```

Algorithm 2 Algorithm to transfer PHI instructions

```
function  $\text{TRANSFERPHIS}(\text{oldBB}, \text{newBB})$ 
  for  $phi \in \text{phiInstructions}(\text{oldBB})$  do
     $\text{newPHI} \leftarrow \text{createPHIInstruction}()$ 
     $\text{changed} \leftarrow \text{false}$ 
    for all  $phiOp \in \text{operands}(phi)$  do
       $phiReg \leftarrow \text{getReg}(phiOp)$ 
       $phiBB \leftarrow \text{getBB}(phiOp)$ 
      if  $phiBB \notin \text{predecessors}(\text{oldBB})$  then
         $\text{addOperand}(\text{newPHI}, \langle phiReg, phiBB \rangle)$ 
         $\text{removeOperand}(phi, phiOp)$ 
         $\text{changed} \leftarrow \text{true}$ 
      end if
    end for
    if  $\text{changed}$  then
       $\text{addOperand}(phi, \langle \text{newPHI}, \text{newBB} \rangle)$ 
    end if
  end for
end function
```

Algorithm 3 Algorithm for split-join insertion

```
visited  $\leftarrow \emptyset$  ▷ set of IPDOM BBs in which join is inserted
ipdomMap  $\leftarrow \emptyset$  ▷ oldIPDOM  $\mapsto$  newIPDOM
for bb  $\in$  blocks(kernel) do
  if numSuccessors(bb) = 2 then
    condJump  $\leftarrow$  FINDFIRSTTERMINATOR(bb)
    splitInst  $\leftarrow$  CREATESPLITINST(getPredicateOperand(condJump))
    INSERT(splitInst, prev(condJump))
    succ1  $\leftarrow$  getSuccessor(bb, 0)
    succ2  $\leftarrow$  getSuccessor(bb, 1)
    ipdom  $\leftarrow$  FINDNEARESTCOMMONPOSTDOMINATOR(succ1, succ2)
    joinInst  $\leftarrow$  CREATEJOININST()
    if ipdom  $\notin$  visited then
      insert(joinInst, ipdom)
      visited  $\leftarrow$  visited  $\cup$  ipdom
    else
      newIpdom = CREATEBASICBLOCK()
      INSERT(joinInst, newIpdom)
      DFS(succ1, ipdom, newIpdom) ▷ In dfs order, update successors of BBs
      DFS(succ2, ipdom, newIpdom) ▷ from old IPDOM to new IPDOM
      if ipdom  $\notin$  ipdomMap then
        insertPt  $\leftarrow$  prev(ipdom)
      else
        insertPt  $\leftarrow$  prev(ipdomMap[ipdom])
      end if
      INSERT(newIpdom, insertPt)
      ipdomMap[ipdom]  $\leftarrow$  newIpdom
      TRANSFERPHIS(ipdom, newIpdom)
    end if
  end if
end for
```

Algorithm 4 Algorithm to update successors from old IPDOM to new IPDOM

```
visited  $\leftarrow \emptyset$  ▷ set of visited BBs
function DFS(start, oldIpdom, newIpdom)
  visited  $\leftarrow$  visited  $\cup$  start
  if oldIpdom  $\in$  successors(start) then
    REPLACESUCCESSOR(start, oldIpdom, newIpdom)
  end if
  for succ  $\in$  successors(start) do
    if succ  $\notin$  visited then
      DFS(succ, oldIpdom, newIpdom)
    end if
  end for
end function
```

4.2 Predication

The HARP backend uses LLVM’s standard If-conversion pass to generate predicated code. The If-conversion pass works iteratively on the control flow graph, looking for triangle and diamond patterns which correspond to if-then and if-else constructs, and “flattens” them with predication. The pass calls functions defined by the HARP backend to check profitability, legality and predication. Figure 19 shows the functions implemented in the HARP backend to support the target-independent If-conversion pass.

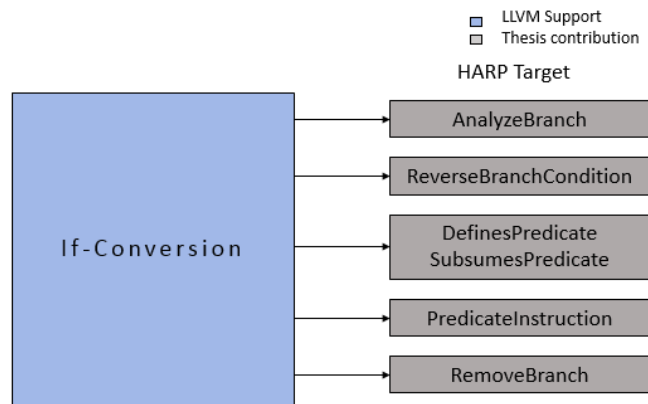


Figure 19: LLVM-HARP backend Co-design for If-conversion

4.2.1 Predication Support in HARP backend

Predication support in the HARP backend is provided via the operand class `PredicateOperand` defined by LLVM. Every predicable instruction in the HARP target description is defined with a first operand of type `PredicateOperand`. A `PredicateOperand` is an optional operand whose default value is an ‘always-execute’ value. The HARP backend sets the default value to `%p_fake`, a fake predicate register. During instruction-selection, if no value is supplied for the optional operand its default value is used. When the If-conversion pass calls into the HARP backend for predicated an instruction, the `%p_fake` register is replaced with the branch predicate. Finally, the `HARPAsmPrinter` skips printing the default `PredicateOperand` value for

unpredicated instructions. The example shown below shows the stages of the IR from code generation to final assembly:

source	codegen	ifcvt	asm
<pre>if b[lid] > 0 { b[lid]++; }</pre>	<pre>BB#0: %R0 = LD %Pfake, %R0, 0; %R0 = SUBI %Pfake, %R0, 1 %P3 = ISNEG %Pfake, %R0 JMP %P3, <BB#2> JMPuncond <BB#1> BB#1: %R1 = LD %Pfake, %R0, 0; %R1 = ADDI %Pfake, %R1, 1 ST %Pfake, %R1, %R0, 0; BB#2: RET %R7</pre>	<pre>BB#0: %R0 = LD %Pfake, %R0, 0; %R0 = SUBI %Pfake, %R0, 1 %P3 = ISNEG %Pfake, %R0 %P2 = NOTP %Pfake, %P3 %R1 = LD %P2, %R0, 0, %R1; %R1 = ADDI %P2, %R1, 1, %R1 ST %P2, %R1, %R0, 0; RET %R7</pre>	<pre>foo: ld %r0, %r0, #0 subi %r1, %r0, #1 isneg @p3, %r1 notp @p2, @p3 @p2? ld %r1, %r0, #0 @p2? addi %r1, %r1, #1 @p2? st %r1, %r0, #0 jmpr %ra</pre>

4.2.2 Limitations of LLVM If-conversion pass

The LLVM If-conversion pass works on the Machine IR after register allocation when the Machine IR is no longer in SSA form. Since the SSA representation is not predicate-aware and unsuitable to represent predicated code. Furthermore, the code generation algorithms in LLVM such as register allocation, liveness analysis, etc., are predicate agnostic. Running an If-conversion pass on register allocated code leads to several missed opportunities. This is because If-conversion sometimes involves inserting new instructions to compute predicates. On Machine IR before register allocation this can be achieved by merely creating new virtual registers. However, on register allocated code, it is more complex to create new live-ranges. The HARP backend currently works around this problem by inserting split-join instructions for conditional branches that cannot be if-converted.

Due to aforementioned reasons, we cannot achieve full if-conversion in the context of the HARP compiler. Figure 20 shows the ratio of conditional branches in `harpbench` that could be if-converted by the LLVM If-conversion pass. Only 58% of the conditional branches in `harpbench` are if-converted. This inhibits us from evaluating the benefits of full if-conversion vs split-join infrastructure for control divergence management.

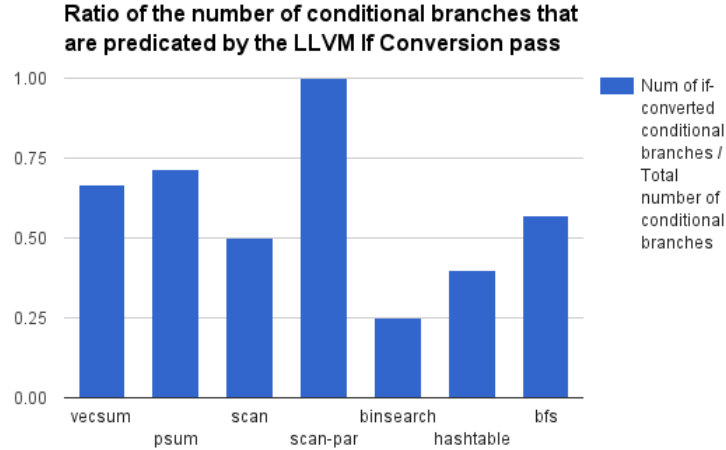


Figure 20: Ratio of conditional branches in `harpbench` that can be predicated by LLVM If-conversion pass

4.3 Decision framework for control divergence management

Both the split-join infrastructure and predication have unique advantages and disadvantages. On a unanimous branch, the split-join infrastructure will execute only one direction of the branch. However, predication will execute both directions of the branch, slowing down performance. On a nonunanimous branch, both directions of the branch will be executed by the split-join infrastructure as well as predication. However, it is beneficial to generate predicated code for such branches, because the split-join infrastructure requires additional instructions.

The decision framework applies conservative static analysis to decide if a conditional branch is likely nonunanimous or not. Section 4.3.1 describes this in detail. Some conditional branches cannot be classified to be unanimous or nonunanimous statically, either due to hard-to-analyze conditional expressions or due to arbitrary data functions with varying runtime behavior. Such conditional branches can be classified using dynamic profile information. Section 4.3.2 describes the profile guided decision framework. Based on the analyses, the decision framework inserts split-join instructions for conditional branches for those that are likely to be unanimous and

if-convert branches that are likely to be nonunanimous.

4.3.1 Static analysis driven decision framework for control divergence management

The kernel functions for the HARP architecture are written in C with HARP specific attributes. An example is shown below:

```
__attribute__((harp_kernel)) int scan_par(  
    __attribute__((warp_id)) int w_id,  
    __attribute__((lane_id)) int l_id, int val) {  
    ...  
}
```

The `lane Id` and `warp Id` are passed as function arguments to the kernel marked with `harp_kernel` attribute. Other arguments of the kernel can be thread invariant or not depending upon whether the kernel was called from a non-kernel function or another kernel function. This programming model simplifies static analysis to determine whether a conditional branch is nonunanimous or not.

The static analysis is guided by the following simple rules:

- A conditional branch dependent only on `warp Id` is unanimous.
- A conditional branch dependent on `lane Id` is nonunanimous.
- A conditional branch dependent on an arbitrary data function indexed by `lane Id` is indeterminate.

To determine whether a conditional branch is nonunanimous we follow the def-use chains of the operands of the conditional branch instruction. If we encounter a `lane Id`, the conditional branch is marked nonunanimous. Conditional branches marked as nonunanimous by static analysis are if-converted. However, if we encounter a lane indexed load, we do not classify such conditional branches as nonunanimous. This is because, static analysis conservatively classifies conditional branches as nonunanimous. If we optimistically classify conditional branch as nonunanimous, it can result

in performance degradation if the conditional branch is unanimous at runtime. Algorithm 5 describes this in detail.

If we encounter a conditional branch with only `warp Id` and constants, the conditional branch is marked unanimous. The static analysis that classifies conditional branches as unanimous is precise. Therefore, no control divergence management is required for such conditional branches. Even though programmers may not write statically thread-invariant conditional branches, some optimizations may create such branches. In the `harpbench` microbenchmarks, only 0.92% of the conditional branches are statically unanimous. Algorithm 6 describes how conditional branches are classified as unanimous statically. It is a very conservative algorithm, and far more powerful algorithms [24] exist.

Algorithm 5 Algorithm to identify nonunanimous conditional branches

```

function ISNONUNANIMOUS(branchInst)
  worklist  $\leftarrow$   $\emptyset$             $\triangleright$  set of instructions whose def-uses are being explored
  branchPredicate  $\leftarrow$  GETOPERAND(branchInst, 0)
  defInst  $\leftarrow$  GETVREGDEF(branchPredicate)
  worklist  $\leftarrow$  worklist  $\cup$  defInst
  while worklist  $\neq$   $\emptyset$  do
    inst  $\leftarrow$  worklist.remove()
    if ISLANEINDEXEDLOAD(inst) then return false
    end if
    for op  $\in$  operands(inst) do
      if op  $\in$  laneId then
        return true
      end if
      worklist  $\leftarrow$  worklist  $\cup$  GETVREGDEF(op)
    end for
  end while
  return false
end function

```

Algorithm 6 Algorithm to identify unanimous conditional branches

```
function ISUNANIMOUS(branchInst)
  worklist  $\leftarrow \emptyset$             $\triangleright$  set of instructions whose def-uses are being explored
  branchPredicate  $\leftarrow$  GETOPERAND(branchInst, 0)
  defInst  $\leftarrow$  GETVREGDEF(branchPredicate)
  worklist  $\leftarrow$  worklist  $\cup$  defInst
  while worklist  $\neq \emptyset$  do
    inst  $\leftarrow$  worklist.remove()
    for op  $\in$  operands(inst) do
      if op  $\in$  laneId  $\vee$  op  $\in$  threadVariantArg then
        return false
      end if
      opDef = GETVREGDEF(op)
      if opDef  $\in$  load  $\vee$  opDef  $\in$  phi then            $\triangleright$  hard-to-analyze instructions
        return false
      end if
      worklist  $\leftarrow$  worklist  $\cup$  opDef
    end for
  end while
  return true
end function
```

The decision framework performs if-conversion only on the conditional branches marked as nonunanimous by the static analysis. For other conditional branches in the kernel, split-join instructions are inserted.

The static analysis used to determine whether a conditional branch is nonunanimous can consist of false positives. This can be due to complex conditional expressions such as `lane_id < k`. This conditional branch will be classified as nonunanimous in our current static analysis framework irrespective of the value of the constant `k`. More sophisticated static analysis algorithms may help cut down such false positives. The static analysis is also incomplete. It cannot identify all nonunanimous conditional branches. It can only classify conditional branches that are directly dependent on `lane Id` as nonunanimous. Conditional branches such as `a[lane_id] > 0` however, cannot be classified using this static analysis. For conditional branches involving such expressions, dynamic profile information can help identify whether the conditional branch is unanimous or nonunanimous. This is discussed next.

4.3.2 Profile guided decision framework for control divergence management

Static analysis cannot classify all conditional branches, specifically conditional branches that are functions of loads indexed by `lane Id`. Such conditional branches are quite prevalent in programs written for data-parallel architectures. In order to make a well informed decision for generating code for branches that are hard to analyze statically, the HARP backend uses dynamic profile information. Dynamic profile information collected across all threads summarizes branch behavior, which is used in guiding the code generation of control flow structures.

In order to decide between split-join infrastructure and predication using dynamic profile information, the HARP backend instruments the kernel, collects profile counters, and then analyzes the profile information to decide whether to predicate or insert split-join instructions. This is explained in detail next.

- **Instrumentation:** During the instrumentation phase, the HARP backend creates a 3d array of profile counters that can be indexed by the branch number, `lane Id` and `warp Id`. Instrumentation code that increments the appropriate index in the 3d array is added for conditional branches. On running the instrumented kernel, the profile counters are recorded in a file, which will be read during the profile guided code generation phase. Algorithm 7 describes the instrumentation in detail.
- **Profile guided analysis:** During the profile-guided analysis phase, the HARP backend reads the profile counters from the instrumented run into a 3d array that can be indexed by branch number, `warp Id` and `lane Id`. Algorithm 8 describes it in detail.

Algorithm 7 Algorithm to instrument conditional branches

```
counterNo  $\leftarrow$  0
for  $bb \in blocks(kernel)$  do
  if  $numSuccessors(bb) = 2$  then
    if  $bb \notin loopHeader$  then
      condJump  $\leftarrow$  FINDFIRSTTERMINATOR( $bb$ )
      INSERTPROFILECOUNTER( $bb$ , counterNo++)
      INSERTPROFILECOUNTER(getSuccessor( $bb$ , 0), counterNo++)
    end if
  end if
end for
```

Algorithm 8 Algorithm to read profile counters

```
counterNo  $\leftarrow$  0
for  $bb \in blocks(kernel)$  do
  if  $numSuccessors(bb) = 2$  then
    if  $bb \notin loopHeader$  then
      condJump  $\leftarrow$  FINDFIRSTTERMINATOR( $bb$ )
      for  $w \leftarrow 0; w < Warps, w++$  do
        for  $l \leftarrow 0; l < Lanes, l++$  do
          counterArr[counterNo][w][l]  $\leftarrow$  readProfileCounter()
          counterArr[counterNo + 1][w][l]  $\leftarrow$  readProfileCounter()
        end for
      end for
      counterNo  $\leftarrow$  counterNo + 2
    end if
  end if
end for
```

Once the profile counters are read, the profile guided analysis phase analyzes these profile counters to determine whether the conditional branches were unanimous or not during the instrumented run. The analysis treats conditional branches differently depending upon whether they are inside a loop or not.

- **Conditional branches not nested within a loop:** For conditional branches not nested within a loop, the profile counters can be used to

exactly determine if the branches were unanimous or not during the instrumented run. Using this information the HARP backend applies if-conversion for nonunanimous branches and split-join insertion for unanimous branches. For branches predicted unanimous by profile information, split-join insertion is necessary. This is because, the branch characteristics of the instrumented run may not be accurate for all runs of the kernel. When the branch characteristics do not match, at worst performance degradation is acceptable, but not incorrect results.

- ***Conditional branches within a loop:*** For conditional branches within loop structures, the profile counters do not exactly represent if the branches were unanimous or not. Loops are central to performance, and code generator should use split-join infrastructure whenever branches within a loop are unanimous for a majority number of iterations. Profile counters summarize the branch behavior across all iterations. From the profile information, we cannot tell if the conditional branches within loops were unanimous in the same iterations.

In order to determine the exact behavior of the branches in every iteration, tracing or sampled tracing can be employed. This is infeasible as it can lead to large and unbounded array of counters. Therefore, the HARP backend uses the following heuristics:

A conditional branch within a loop is predicted **Taken** if:

$$\text{profileCounter}(\text{taken}) > \text{loopHighUnanFrac} * \text{profileCounter}(\text{branch})$$

A conditional branch within a loop is predicted **Not Taken** if:

$$\text{profileCounter}(\text{taken}) < \text{loopLowUnanFrac} * \text{profileCounter}(\text{branch})$$

`loopHighUnanFrac` and `loopLowUnanFrac` are parameters in the decision framework set to 0.9 and 0.1 currently.

Based on per-lane prediction of branch direction, a conditional branch in a warp is predicted unanimous if:

$$\begin{aligned} \Sigma (\text{lanes predicted Taken}) &= \text{Lane size} \\ \vee \\ \Sigma (\text{lanes predicted Not Taken}) &= \text{Lane size} \end{aligned}$$

Finally based on per-warp prediction of unanimousness, a conditional branch is classified as unanimous if:

$$\Sigma (\text{warps predicted unanimous}) > \text{unanFrac} * \text{Warp size}$$

`unanFrac` is a parameter in the decision framework set to 0.9 currently.

Based on these heuristics, the decision framework classifies a conditional branch as unanimous. All other branches are classified as nonunanimous. Similar to the static analysis guided decision framework, split-join is inserted for unanimous branches and other branches are if-converted.

CHAPTER V

EXPERIMENTS AND RESULTS

The HARP compiler is evaluated using the LLVM Test Infrastructure as well as a microbenchmark suite called *harpbench*. Details of both these test frameworks are discussed below:

- **LLVM Test Infrastructure:** The code generation tests for the HARP compiler are integrated into the LLVM Test Infrastructure. Both regression-tests and whole program tests are included. The regression tests are compile-only tests, consisting of assertions to validate the generated assembly. The whole program tests are executed on the HARP emulator. The LLVM Test Infrastructure provides an automated testing framework, driven by a tool called *lit*. *lit* is a lightweight testing tool, which provides easy launch, runs tests in parallel and summarizes results.
- **harpbench:** *harpbench* is a microbenchmark suite written in the C language with HARP specific extensions. It consists of a representative workload set for in-memory SIMT processing. There are both SPMD (Single Program Multiple Data) style programs and simple single threaded programs in *harpbench*. The *harpbench* programs are tested on the HARP emulator.

Both these test frameworks are used to evaluate two aspects of the HARP compiler:

- Code generation robustness on different combinations of types and operations
- Comparison of the control divergence management techniques

Each evaluation criteria is discussed in the following sections.

5.1 Evaluation of code generation

In order to evaluate the robustness of the HARP compiler, new regression tests were developed and integrated within the LLVM Test Infrastructure. The tests are written to exercise different code paths in the HARP backend, for different kinds of operations and types. Code generation of the HARP compiler is also tested with `harpbench` microbenchmark which consists of whole-program tests that enable us to validate the correctness of generated code. Table 3 consists of details of these tests.

Table 3: List of tests to evaluate code generation of HARP compiler

Code generation tests	
LLVM LIT Infrastructure Regression and whole-program tests testing different components of the HARP backend	Arithmetic/Logic/Bitwise operations Load/Store/Pointer Arithmetic operations Floating point operations Function Call/Return/Indirect Function Call Calling Convention, argument passing Structures, Strings Control flow structures Small and Large immediates Global variables/Stack allocation
<code>harpbench</code> single-threaded programs	quick sort bubble sort shuffle
<code>harpbench</code> SPMD-style programs	vector sum prefix sum scan scan-par binary search hashtable bfs

The number of tests on which the robustness of the HARP compiler is tested is limited in comparison with the size of test frameworks for production quality compilers. Developing regression tests, and fixing bugs in the HARP compiler is an on-going and continuous effort.

5.2 Evaluation of decision framework for control divergence management

The different control divergence management techniques are evaluated with the `harpbench` microbenchmark on the HARP emulator. The experiments are run on a 16 warp, 16 lane configuration with the `harp32` configuration. `harpbench` consists of 7 data-parallel programs. The divergence ratio of conditional branches is diverse across the different `harpbench` programs. The programs *vecsum*, *psum* mainly consist of conditional branches which are functions of `lane Id` and `warp Id`. *binsearch*, *hashtable*, *scan*, *scan-par* consist of conditional branches which are dependent on lane indexed data, making them highly sensitive to input. *bfs* consists of a combination of both such branches. Figure 21 shows the ratio of divergent conditional branches in `harpbench` for the 16 warp, 16 lane configuration.

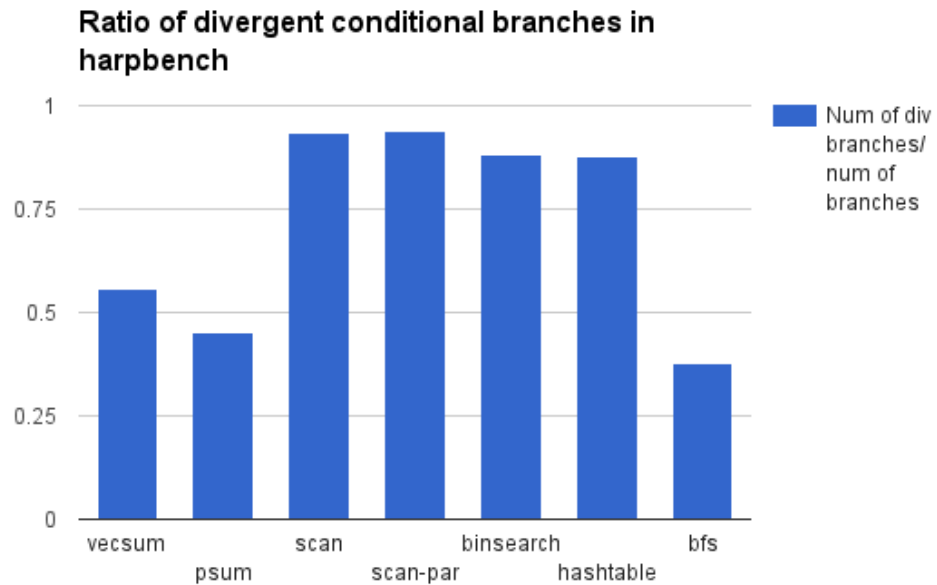


Figure 21: Divergence ratio of `harpbench` programs executed with a 16 warp, 16 lane configuration

The diverse divergence ratios of the `harpbench` programs enable us to clearly evaluate the different control divergence management techniques, each of which respond

differently to the divergence ratio. The results of running `harpbench` with split-join only, predication only, static analysis guided decision framework and profile guided decision framework are shown in Figure 22. The vertical axis represents the kernel dynamic instruction count (KDIC) improvement of the different control divergence management techniques over split-join control divergence management.

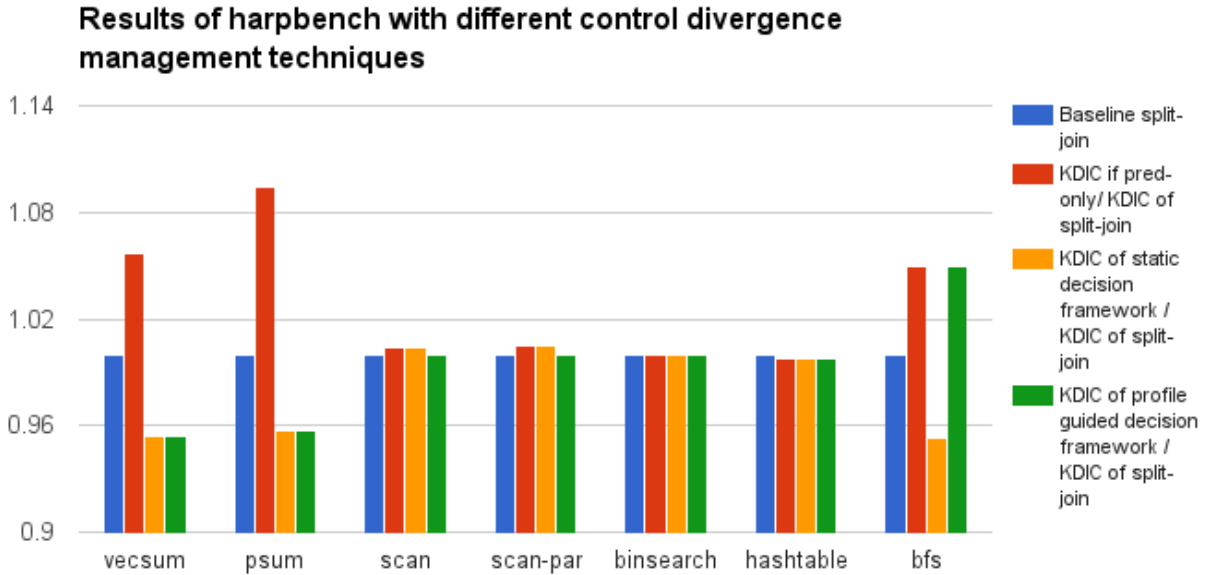


Figure 22: Kernel dynamic instruction count of different control divergence management techniques

The following can be inferred from the preceding results:

- Split-Join-only control divergence management has lower kernel dynamic instruction count than predication-only control divergence management for most programs. This is true even for high-divergence programs *scan*, *scan-par*, *binsearch*. *hashtable* has a slightly higher dynamic instruction count with split-join-only control divergence management.
- Predication-only control divergence management increases kernel dynamic instruction count for most programs. The effect is higher for low divergence

programs *vecsum*, *psum*, *bfs*.

- The kernel dynamic instruction count of static analysis and dynamic profile guided decision frameworks lie in between split-join only and predication-only control divergence management for all programs.
- Dynamic profile guided decision framework is only slightly superior or on-par with static analysis guided decision framework for most programs. And performs poorly on *bfs*.

From the results, it is clear that a split-join-only control divergence management is better than a predication-only control divergence management for these benchmarks. This is true even for high-divergence programs. Because, even if the divergence ratio for these programs are high, they are not 100%. Applying predication to branches that are unanimous at runtime can increase the dynamic instruction count significantly. However the effect is not proportional to the dynamic instruction count if we use split-join instructions for nonunanimous branches, since they merely increase up to 4 instructions per conditional branch. Further, dynamic instruction count is the only measure available on the HARP emulator, making it hard to evaluate clearly, the cost of split-join instructions.

The programs *vecsum*, *psum*, *bfs* consist of a mix of unanimous and nonunanimous conditional branches, analyzable statically. This explains, the large improvement in kernel dynamic instruction count with the static analysis guided decision framework.

Dynamic profile guided decision framework does not provide a high added benefit over the static analysis decision framework for **harpbench**. This is because, static analysis decision framework does not have a large number of false positives in **harpbench**. *scan*, *scan-par*, *binsearch*, *hashtable*, *bfs* have hard-to-analyze branches that are not classified by static analysis as nonunanimous and are left unpredicated. The dynamic profile guided decision framework classifies some of these branches as

nonunanimous correctly, *scan*, *scan-par* benefit slightly from this classification. However *bfs* sees increased kernel dynamic instruction count. This is because the dynamic profile guided decision framework ends up predicating a few long conditional branches.

In conclusion, we can say that there is no universally good control divergence management for all programs in **harpbench**. Programs tend to have both unanimous and nonunanimous conditional branches, and a different control divergence management technique is effective for each. Further, the increase in dynamic instruction count for *bfs* with dynamic profile guided decision framework signals the need for additional factors such as size of the conditional branches to be modelled within the decision framework.

CHAPTER VI

CONCLUSION

In this thesis, a compiler backend for a new exploratory in-memory SIMT architecture called HARP has been designed and developed. The HARP compiler is built on top of the open source LLVM compiler infrastructure. Target specific classes and functions that are called upon by the target independent code generation algorithms of LLVM have been implemented. The code generation of the HARP compiler is evaluated on a set of unit tests and whole-program tests integrated into the LLVM Test Infrastructure, as well as on *harpbench* microbenchmark, which consists of representative workload set for in-memory SIMT architectures.

The thesis also presents the compiler support required for control divergence management for the HARP architecture. New decision frameworks guided by static analysis and dynamic profile information to select between the control divergence management techniques supported by the HARP architecture are developed. Results of the different control divergence management techniques are evaluated on the *harpbench* microbenchmark.

CHAPTER VII

FUTURE WORK

Candidate improvements to current code generation and compiler support for control divergence management are described below:

- ***Tail-call optimization:*** Tail-call optimization enables forgoing allocation of new stack frames for tail-calls. The backend support for tail-call optimization should be implemented.
- ***Reduce the number of reserved registers:*** We reserve some general purpose and predicate registers during code generation to handle certain idiosyncrasies. Most of these scenarios can be worked around using `Register Scavenging`. This should be implemented to generate efficient code.
- ***Predicate aware SSA and predicate-aware code generation algorithms:*** The LLVM If-conversion pass is implemented on the Machine IR after register allocation. This is because, before register allocation, LLVM is in SSA form which is unsuitable for predication. Since, the If-Converter works on the IR with allocated physical registers, it results in several missed if-conversion opportunities. A predicate-aware SSA form such as [23], would enable if-conversion to be implemented before register allocation, which will reduce the missed opportunities. Along with a predicate aware IR, predicate-aware code generation algorithms like register allocation, liveness analysis should be developed in order to generate efficient code.
- ***Static analysis of complex expressions:*** Current static analysis predicts all conditional branches directly dependent on `lane Id` as nonunanimous. For

conditional branches with complex conditional expressions, this may not hold. Advanced static analysis should be implemented to classify such conditional branches.

- ***Additional factors affecting decision framework:*** Depending upon the workload, predicating long branches can have negative impacts on dynamic instruction count, cache behavior, etc. This factor should be further studied and incorporated in the decision framework.

The current HARP compiler is a first step. It is hoped that this lays the foundations of a robust optimizing compiler infrastructure for embedded SIMT processors.

REFERENCES

- [1] “clang, a c language family frontend for llvm.” <http://clang.llvm.org/>.
- [2] “Greedy register allocation in llvm 3.0.” <http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html>.
- [3] “Llvm-dev mainling list.” <http://lists.llvm.org/mailman/listinfo/llvm-dev>.
- [4] “Llvm language reference manual.” <http://llvm.org/docs/LangRef.html>.
- [5] “The llvm target-independent code generator.” <http://llvm.org/docs/CodeGenerator.html>.
- [6] “Tablegen.” <http://llvm.org/docs/TableGen>.
- [7] “Tutorial: Creating an llvm backend for the cpu0 architecture.” <http://jonathan2251.github.io/lbd/>.
- [8] “Writing an llvm backend.” <http://llvm.org/docs/WritingAnLLVMBackend.html>.
- [9] “clang cfe internals manual.” <http://clang.llvm.org/docs/InternalsManual.html>.
- [10] ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., and WARREN, J., “Conversion of control dependence to data dependence,” in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’83, (New York, NY, USA), pp. 177–189, ACM, 1983.
- [11] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., and ZADECK, F. K., “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, Oct. 1991.
- [12] ET AL., H. C., “Hybrid memory cube specification 1.0.”
- [13] FUNG, W. W. L., SHAM, I., YUAN, G., and AAMODT, T. M., “Dynamic warp formation: Efficient SIMD control flow on SIMD graphics hardware,” *ACM Trans. Archit. Code Optim.*, vol. 6, pp. 7:1–7:37, July 2009.
- [14] HAN, T. D. and ABDELRAHMAN, T. S., “Reducing branch divergence in GPU programs,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 3:1–3:8, ACM, 2011.

- [15] KERSEY, C., YALAMANCHILI, S., KIM, H., NIGANIA, N., and KIM, H., “Harmonica: An fpga-based data parallel soft core,” in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 171–171, IEEE, 2014.
- [16] KERSEY, C. D., YALAMANCHILI, S., and KIM, H., “Cymric: A framework for prototyping near-memory architectures,”
- [17] KERSEY, C. D., YALAMANCHILI, S., and KIM, H., “Simt-based logic layers for stacked dram architectures: A prototype,” in *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 29–30, ACM, 2015.
- [18] LATTNER, C. and ADVE, V., “Llvm: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO ’04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.
- [19] LEE, Y., GROVER, V., KRASHINSKY, R., STEPHENSON, M., KECKLER, S. W., and ASANOVIĆ, K., “Exploring the design space of spmd divergence management on data-parallel architectures,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 101–113, IEEE Computer Society, 2014.
- [20] MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., and BRINGMANN, R. A., “Effective compiler support for predicated execution using the hyperblock,” in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, (Los Alamitos, CA, USA), pp. 45–54, IEEE Computer Society Press, 1992.
- [21] NICKOLLS, J., BUCK, I., GARLAND, M., and SKADRON, K., “Scalable parallel programming with cuda,” *Queue*, vol. 6, pp. 40–53, Mar. 2008.
- [22] STONE, J. E., GOHARA, D., and SHI, G., “Opencl: A parallel programming standard for heterogeneous computing systems,” *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.
- [23] STOUTCHININ, A. and DE FERRIERE, F., “Efficient static single assignment form for predication,” in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, (Washington, DC, USA), pp. 172–181, IEEE Computer Society, 2001.
- [24] STRATTON, J. A., GROVER, V., MARATHE, J., AARTS, B., MURPHY, M., HU, Z., and HWU, W.-M. W., “Efficient compilation of fine-grained spmd-threaded programs for multicore cpus,” in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’10, (New York, NY, USA), pp. 111–119, ACM, 2010.

- [25] ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., and SHEN, X., “On-the-fly elimination of dynamic irregularities for gpu computing,” *SIGPLAN Not.*, vol. 46, pp. 369–380, Mar. 2011.