UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Computer Science, Electrical Engineering and Mathematics

Department of Computer Science

Master Thesis

Easy-to-use on-the-fly binary program acceleration on many-cores

Author:

Marvin Damschen

First reviewer:

Jun.-Prof. Dr. Christian Plessl

Second reviewer:

Prof. Dr. Friedhelm Meyer auf der Heide

August 28, 2014

Abstract

This thesis introduces Binary Acceleration At Runtime (BAAR), an easy-to-use on-the-fly binary acceleration mechanism which aims to tackle the problem of enabling existent software to automatically utilize accelerators at runtime. It is able to achieve a speedup of up to 4 without any hints and 5.77 with hints, when comparing execution of code compiled with the Intel C++ Compiler at O2 on an Intel Xeon E5-2670 to execution using BAAR utilizing an Intel Xeon Phi 5110P accelerator card.

BAAR bases on the LLVM Compiler Infrastructure and has a client-server architecture. The client runs the program to be accelerated in an environment which allows program analysis and profiling. Program parts which are identified as suitable for the available accelerator are exported and sent to the server. The server optimizes these program parts for the accelerator and provides RPC execution for the client. The client transforms its program to utilize accelerated execution on the server for offloaded program parts.

This thesis makes three main contributions: Firstly, a server backend for BAAR to utilize an Intel Xeon Phi accelerator card for executing remote calls is presented. It requires LLVM IR to be transformed into Xeon Phi binaries. Secondly, a mechanism to automatically identify function calls suitable for remote execution on the accelerator is designed and implemented. Finally, BAAR is extended to perform automatic parallelization and vectorization on offloaded code using LLVM Passes.

The implementation of BAAR is evaluated in detail. Its practicality for real-world examples is shown based on stencil codes. The insights gained during evaluation are used to describe future directions of research, e.g., offloading more fine-granular program parts than functions, a more sophisticated communication mechanism or introducing on stack replacement.

Acknowledgements

I would like to express my gratitude to my advisor Jun.-Prof. Dr. Christian Plessl for his support throughout the work on this project and his valuable professional advice. I also want to thank Tobias Kenter from the Custom Computing Group for his constructive criticism.

My special thanks goes to the Paderborn Center for Parallel Computing, PC^2, for providing access to all the technical resources I needed. My heartfelt thanks to Axel Keller for his swift and helpful technical support.

I express my deepest appreciation to Katharina Kalisch for her continuous support for all of my endeavors.

Declaration Of Authorship

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text and that this work has not been submitted for any other degree or professional qualification except as specified.

Paderborn, August 28, 2014

Marvin Damschen

Contents

List Of Figures

List Of Listings

List Of Tables

1. Preface

1.1. Motivation

Modern computer systems supply multiple diverse computing cores. Current graphic cards can be utilized for general purpose computations and several instruction set extensions to the x86 architecture, especially for single instruction multiple data (SIMD) operations, are available. These techniques offer a huge potential for optimizing applications. However, to exploit the full performance these heterogeneous architectures provide, programs have to be specifically optimized and frameworks like, e.g., OpenCL or vendor-supplied APIs have to be used. Furthermore, existent software has to be adapted to the new developments in computer technology. Without adaptation, it misses the possible performance increase new instruction set extensions and accelerators provide, but it is unreasonable to expect that any software, which could benefit from these new developments, will be adapted to utilize any available instruction set extension or accelerator. The rise of multi-core over single-core processors in the past decade has shown how cumbersome the optimization of software for new computer architectures can be for software developers.

Experienced software developers are able to exploit the full potential of their systems by manually optimizing their code. However, to manually optimize for a broad range of steadily advancing computer architectures is time and resource consuming, it can quickly become unmanageable. Therefore, automatic tools to optimize software for instruction set extensions and accelerators are needed. Compiler suites supply invaluable tools for developers which aid optimizing software for new technology. Though when dealing with closed-source legacy software, developers relying on a certain library and users relying on a certain program cannot fulfill their needs for optimized software by employing compiler suites. The possibilities to optimize compiled software are generally scarce. A widely used approach to cope with this problem is just-in-time compilation. Most Java Virtual Machines perform just-in-time compilation of Java bytecode and therefore feature adaptive optimization, but only in a limited extent as the target platform is the same as the platform running the virtual machine. Most notably, accelerators are not utilized by this approach. To exploit the full performance of modern computer systems with existing software, a more general approach than just-in-time compilation is needed.

1.2. Related Work

Optimizing existent software for new technology has previously been studied. Several projects try to tackle this problem in various ways.

The *Intel SPMD Program Compiler (ispc)* project [22] follows a static approach. It is a compiler for a C-based programming language with *single program multiple data (SPMD)* extensions. ispc is designed to enable programmers to write programs which exploit vector units and multiple cores of current CPUs in a familiar language, without the need of directly writing intrinsics. In contrast to our approach, ispc requires a programmer to rework available source to use the ispc language. Furthermore, once the source is compiled into a binary there is no way of adapting the binary to changing environments.

KernelGen [19] is a project which follows a more dynamic, but also more specialized approach than ispc. KernelGen consists of a compiler pipeline, as well as a runtime environment. It supports standard C and Fortran code as input. The code is compiled into an object file which contains CPU code and GPU kernels for compute intensive parts in a preliminary representation. At runtime, the GPU kernels are automatically parallelized and JIT-compiled with the help of runtime information. Similar to our approach, KernelGen tries to tackle the problem of accelerating legacy programs without manually adapting them. However in contrast to our approach, KernelGen requires the source code to be available and it is specialized on NVIDIA GPUs.

The *sambamba* project [24] aims to parallelize and dynamically adapt programs to the available resources at runtime using adaptive dispatch between sequential and parallel function definitions. C/C++ code is statically analyzed during compilation and linked in intermediate form to a runtime environment resulting in a *fat binary*. The runtime environment can access the information gathered at compile time, as well as instrument the code dynamically. The information is used to dynamically adapt the program using just-in-time compilation to execute most efficiently in the current environment. sambamba creates binaries which can dynamically adapt, but does not incorporate accelerators. Similar to classic just-in-time compilation, the adaptation is limited to the target the program is currently running on. Our approach aims to support any resource available, including accelerators.

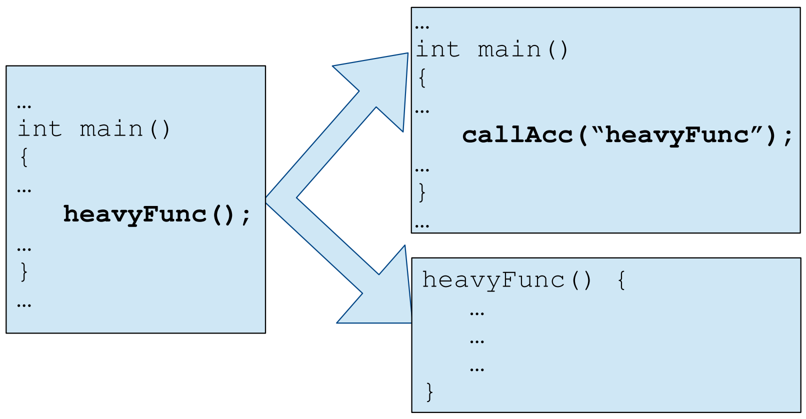

Figure 1.1.: The program is analyzed and profiled during execution to identify parts suitable for offloading. Once enough information is gathered, suitable parts are exported and the program is adapted to call these parts remotely.

1.3. Our Approach

Our approach is to profile and analyze running programs during execution in ways similar to just-in-time compilation techniques. Though instead of directly optimizing heavy parts of the program for the same processing unit the program is currently running on, we extract them into a module which can be processed further to optimized code for a suitable target. Our goal is to split the program into a *remote part* – the optimized code in the module – and a *local part* – an altered version of the program which calls the remote part instead of the original unoptimized code – as depicted in Figure 1.1. The optimization of the module as well as the alteration of the original program happen during execution, transparent to the user. The target on which the remote part is executed can be an accelerator in the same system the program is running on, it can also be a remote server communicating with our local system. Furthermore, the architecture of the target does not have to be known in detail to our local system. The optimization of the module takes place on the target system (*server*) and it provides previously defined interfaces to call procedures from our local system (*client*) once it is done. Communication between server and client can be handled over shared memory if running on the same system, or network otherwise. A sample setup is depicted in Figure 1.2.

This approach differs from the approaches presented in Section 1.2, because neither does it require the source code to be available nor is the acceleration target fixed in any way. Our approach is open to any combination of host architecture, target architecture and communication mechanism and is therefore not bound to any specific use case. It can be used to optimize programs to utilize a graphic card in the same system, it can also be used to offload heavy computations

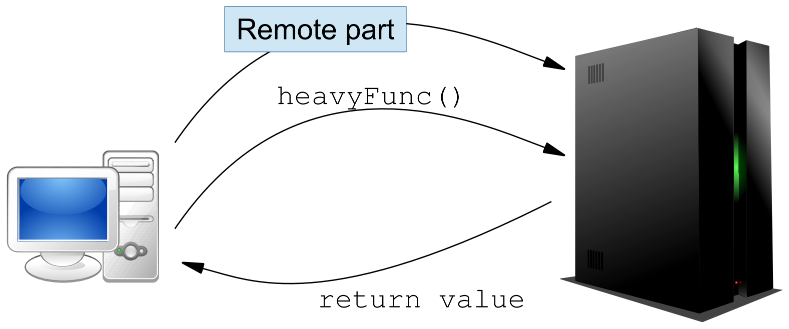

Figure 1.2.: The exported module (remote part of the program) is sent to the server, where it is transformed and optimized. Afterwards, the server is responsible for handling calls of the adapted original program (local part) to the remote part.

from a desktop computer to a super computer. However, our approach requires programs in a format which is suitable for analysis, profiling and adaptation in terms of optimization and transformation. A suitable format for our needs is the *LLVM immediate representation* (LLVM IR). *The LLVM compiler infrastructure* (LLVM) [17] is an extensive set of modular and reusable compiler and toolchain technologies. LLVM IR is a code representation used for compiler transformations and analysis within LLVM. It is safe to assume programs in the form of LLVM IR, even when source code is unavailable, as the feasibility of transforming binaries into LLVM IR has been shown [9]. LLVM IR gives us the possibility to make use of powerful LLVM tools to analyze and transform our program. Furthermore, we can export the parts of our program to be optimized in form of LLVM IR and send them to the server to be analyzed and transformed further. Additionally, we can make use of the LLVM backends to transform our code in LLVM IR into binaries for several target architectures. The LLVM compiler infrastructure is a fundamental element to realize our approach.

Finding a suitable representation for program code throughout our approach is just one of the several problems which have to be tackled for a successful realization. To be able to accelerate programs, they have to be analyzed first. Therefore, profiling metrics and approaches have to be explored to enable the implementation of sophisticated means to analyze programs during execution. To know which parts of the program consume much time and resources is not enough however, a model of the expected speedup in respect to general characteristics of the target architecture when offloading parts to a server is also necessary. Without it, we would possibly offload code which is not suitable for the target architecture the server provides. Furthermore, a suitable protocol for client-server communication has to be defined to ensure correct execution while minimizing delays when offloading and optimizing code; implementations of communication mechanisms should always exploit the advantages of their medium. Once the server receives the module, containing the exported program parts to be accelerated, it has to process the contained LLVM IR to make it most suitable for the target architecture by employing LLVM tools and possibly target-specific compilers, e.g., this can mean that the code has to be distributed to several threads and operations have to be transformed to utilize instruction set extensions. At the end of this process, a target-specific binary has to be employed by the server to execute incoming calls from the client.

1.4. Goals Of This Thesis And Document Structure

The main goals of this thesis are to realize our approach of an easy-to-use on-the-fly acceleration framework for binary programs and to study its prospects as well as its limitations. The existing implementations of acceleration client and server will be thoroughly introduced in the following chapter and be used as a basis to extract and offload code in the form of LLVM IR. This basis is incrementally extended in the remainder of this thesis.

Chapter 3 deals with the design and implementation of a mechanism to transform LLVM IR code into binaries for the Intel Xeon Phi processor family [15]. The Intel Xeon Phi will be the target architecture for the acceleration server throughout this thesis.

On the basis of the Intel Xeon Phi as an acceleration target, the possibilities of profiling and analyzing running programs are explored in Chapter 4. The acceleration client is extended to automatically identify function calls suitable for accelerated remote execution.

The Intel Xeon Phi requires highly parallelized and vectorized code to exploit its full potential. Therefore, parallelization and vectorization are crucial for the offloaded code to perform well. Chapter 5 discusses the problem of applying these optimizations automatically. Furthermore, the acceleration server is extended to perform automatic parallelization and vectorization on the code it receives from the client.

After acceleration client and server were extended to automatically identify, offload and optimize function calls to be accelerated on the Intel Xeon Phi, a proof-of-concept version of our approach of an easy-to-use on-the-fly acceleration framework for binary programs is ready to be evaluated. Chapter 6 discusses the evaluation of the implementation, its performance and the function call identification mechanism. In this chapter the practicality of our approach is demonstrated and valuable insights gained during evaluation are discussed. Additionally, opportunities for further developing our approach are pointed out.

Chapter 7 concludes this thesis. It discusses the insights gained in previous chapters and explores future directions of our approach of an easy-to-use on-the-fly acceleration framework for binary programs.

The following chapter extensively introduces the fundamentals needed for the topics discussed in the remainder of this thesis.

2. Introduction

2.1. Automatic Parallelization

To exploit the full performance of modern multiprocessor systems, sequential code needs to be parallelized. Manually parallelizing sequential programs is time consuming and error-prone however. *Automatic parallelization* are optimizations performed by a compiler which aim to relive programmers from this task. As most of the execution time of a program is generally spent in loops, automatic parallelization mainly focuses on these control structures. Automatic parallelization faces very difficult challenges, e.g., the number of iterations of a loop may be unknown at compile time, pointers as well as indirect addressing complicates dependence analysis and resource accesses need to be coordinated. The most commonly applied automatic parallelization techniques are multi-threading and vectorization, which will also be the central optimizations performed to accelerate programs on a many core processor in this thesis.

2.1.1. Multi-Threading

Automatic multi-threading is an optimization that tries to split up the iterations of a loop to execute them in separate threads. These threads can then be distributed among multiple processors at runtime, allowing parallel execution of formerly sequentially executed iterations.

It may not always be safe to execute a loop in multiple threads as the iterations may depend on each other. Additionally, the computations performed by the loop may not be heavy enough so that the overhead imposed by maintaining multiple threads may exceed the time saved. Therefore, automatic multi-threading requires a comprehensive analysis of the program to be parallelized.

Automatic multi-threading is commonly referred to as automatic parallelization or simply parallelization when used in context. Automatic parallelization techniques other than multi-threading are treated as special cases and denoted explicitly. In the following, the same terminology will be used. Automatic multi-threading will simply be referred to as parallelization throughout this

```
for (int i = 0; i < 1024; i++)
   A[i] = B[i] + C[i];
```

Listing 2.1: Simple scalar loop

```
for (int i = 0; i < 1024; i=i+4)
   A[i:i+3] = B[i:i+3] + C[i:i+3];
```

Listing 2.2: Vectorized loop

Figure 2.1.: Simple vectorization example

thesis.

2.1.2. Vectorization

Automatic vectorization tries to optimize programs to make use of the single instruction multiple data (SIMD) extensions available in modern processors wherever possible. For this purpose, scalar operations working on a single pair of operands are transformed into vector operations working on multiple pairs of operands at the same time.

Figure 2.1 gives an idea of the concept of vectorization. Listing 2.1 shows a `for`-loop, which iterates over all elements of arrays `B` and `C` to pairwise add them. The result is stored in array `A`. This scalar code operates on one pair of elements at a time. Listing 2.2 shows a vectorized version of the same loop. This version operates on four elements at a time. Consequently, `i` is increased by four in every iteration. Assuming the hardware provides a SIMD unit able to process four elements at a time, the performance of the loop is drastically improved.

Similar to parallelization, automatic vectorization may only be applied when no dependencies exist which could alter the result. To guarantee that vectorization can safely be applied, thorough program analysis is crucial.

In the remainder of this thesis, automatic vectorization will simply be referred to as vectorization.

2.2. The LLVM Compiler Infrastructure

This section introduces the compiler infrastructure LLVM. It is a fundamental part of the realization of our approach of on-the-fly binary program acceleration.

2.2.1. Overview

LLVM [11, 17] is an umbrella project consisting of numerous low-level toolchain tools and libraries. LLVM was formerly an acronym for Low Level Virtual Machine, today it is the brand of

```c
int main(int argc, char *argv[]) {







  printf("Hello World!\n");


  return 0;
}
```

Listing 2.3: Hello World in C

```llvm
@.str = private unnamed_addr constant [14 x i8]
    c"Hello World!\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  store i32 0, i32* %1
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = call i32 (i8*, ...)* @printf(i8*
      getelementptr inbounds ([14 x i8]* @.str,
      i32 0, i32 0))
  ret i32 0
}
```

Listing 2.4: Hello World in LLVM IR

Figure 2.2.: Hello World in C and LLVM IR code side-by-side (both shortened for brevity)

the umbrella project. One of its well-known sub-projects is the Clang C, C++ and Objective-C compiler, which generates binaries performing similar to binaries generated by GCC while providing a very modular and accessible design. However, LLVM is far more than just the host of Clang. An LLVM-based compiler follows the three phase approach of classical compiler design, consisting of frontend (programming language support), optimization and backend (instruction set support) [8]. From this point of view, Clang is just a frontend and it shares the optimization and backend phases with several other frontends. This three phase design is implemented very cleanly in LLVM and is not only used for static compilers like Clang, but also for interpreters and just-in-time compilers implemented within LLVM. This results in a very modular framework and high code reuse between different compilers as well as the other tools. A crucial part of LLVM which enables this clean implementation of the three phase design and high reusability of any of LLVM's modules is the LLVM Immediate Representation.

2.2.2. LLVM Immediate Representation

The *LLVM Immediate Representation* (LLVM IR) can be seen as the most important part of the LLVM design, it is a key factor which differentiates LLVM from other compiler implementations. LLVM IR is the interface to any mid-level analysis or transformation. Therefore, any frontend has to emit and any backend has to accept code in form of LLVM IR. LLVM IR is a low-level language, representing code in a RISC-like instruction set [21] with crucial higher-level information added. This allows the representation of arbitrary programs and permits extensive optimizations, supporting sophisticated analyses and transformations.

LLVM IR uses a load-store architecture, values are transferred only via `load` and `store` instructions using typed pointers between memory and registers. The LLVM instruction set overloads opcodes and tries to avoid multiple opcodes for the same operations. Therefore, the instruction set consists of just over 30 opcodes. These capture the key operations of ordinary processors. However, LLVM does not introduce machine specific constraints. It provides an infinite set of typed virtual registers and there are no pipelines or low-level calling conventions. Type information, as a high-level feature in LLVM IR, is stored language independently and allows high-level transformations on this low-level code. Figure 2.2 shows the "Hello World" program in C and LLVM IR code side-by-side to give an idea of how LLVM IR code looks like. LLVM IR is a more low-level language than C, initialization of the stack frame is explicitly visible in the code.

The primary code representation of LLVM IR is *Static Single Assignment* (SSA) form, each virtual register is written exactly once and every use of a register has a preceding definition. To keep the code compact, memory locations in LLVM are not in SSA form. The SSA form simplifies data flow optimizations by providing a compact definition-use graph. LLVM IR explicitly defines the control flow graph of any function. Functions consist of basic blocks, and every basic block is a sequence of instructions. The last instruction of each basic block is a single terminating instruction, which explicitly defines succeeding basic blocks.

LLVM IR defines equivalent representations in textual, binary, and compiler internal in-memory form. Conversion between these representations is possible without any information loss. The extensive transformations and optimizations available on LLVM IR, as well as its forms of representation make it a well fitting code representation for the aims of this thesis.

2.2.3. Pass And Pass Manager

Another central aspect of the LLVM design is the concept of a *Pass* [7]. Passes are units where all the compiler logic happens in LLVM. Any analysis or transformation of LLVM IR code happens inside a Pass. Dead code elimination, e.g., is implemented in a transformation pass called `dce`. Analysis passes gather information about the code they are running on and provide this information to other passes.

Passes are managed and run by an infrastructure which is accessible through the *Pass Manager* class. Passes are added to the Pass Manager, the execution of the Passes can then be initiated on an LLVM IR Module or function using it. The Pass Manager optimizes Pass execution by avoiding recomputation of analysis results as much as possible and scheduling the Passes to run efficiently by pipelining Passes together.

Our approach uses Passes to analyze the original program and to transform it into the local part. Furthermore, program optimization in the server relies heavily on LLVM Passes.

2.2.4. Execution Engine

The LLVM *Execution Engine* [4] is a library for running LLVM IR code. It provides a portable interpretation-only execution, as well as extensive just-in-time compilation features for supported targets. It uses the same code generation backends as the LLVM compiler, but can optionally use some faster components.

The Execution Engine supports recompilation of program parts during execution, which allows changing the program at runtime. It is an important part of the client-side of our approach and will enable us to build an environment to profile, analyze and alter the program during execution.

2.2.5. Polly

Polly [13] is an LLVM subproject which integrates a state-of-the-art *polyhedral optimization* infrastructure into LLVM. Polyhedral optimizations are optimizations which are based on the *polyhedral model* [12], a formalism mostly used for the automatic parallelization of suitable programs. Polly is build around CLooG and Isl[1], well-known and advanced libraries used in the polyhedral optimization community. Polly accepts LLVM IR as input and is therefore programming language independent. Through the use of LLVM IR, language constructs like C++ iterators, pointer arithmetic and `goto` based loops are supported.

Polly operates on *Static Control Parts* (SCoP), maximal sets of consecutive statements where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters [10]. SCoPs are detected by semantically analyzing *regions*, subgraphs of the control flow graph which are connected to the remaining graph by an entry and an exit edge only. With the help of LLVM's powerful program analysis Passes, Polly is able to successfully recover loop bounds, conditions and array subscripts from the LLVM IR. No syntactic restrictions are imposed on the program source code. Because Polly detects SCoPs by checking low-level code for SCoP semantics, Polly's optimizations are not limited to for-loops as many polyhedral optimization tools are. Arbitrary control flow structures can be valid SCoPs. This is the case when they

[1]`http://polyhedral.info`

can be rewritten as a well-structured set of for-loops and if-conditions with affine expressions in lower and upper bounds and in the operands of the comparisons. Any memory access that behaves like an array access with affine subscripts is allowed.

Polly will be used to parallelize the code offloaded to the accelerator in this thesis. It can detect parallel loops exposed during polyhedral optimization and automatically generate thread-level parallel code using GNU OpenMP[2] runtime calls.

2.3. The BAAR On-The-Fly Acceleration Environment

BAAR was initiated during my time as a student assistant in the Computer Engineering Group at the University of Paderborn. This section thoroughly describes the status quo of the on-the-fly acceleration infrastructure BAAR in its state before the extensions described in the following chapters were made. BAAR stands for "Binary Acceleration At Runtime" and is our design of an easy-to-use on-the-fly binary program accelerator.

2.3.1. BAAR Client

Overview

The client program is responsible for providing an environment to execute, observe and analyze programs present in the form of LLVM IR. The client is run from the console. On startup, the form of communication with the server – either shared memory or sockets – and the program to run has to be specified. The program is parsed into an in-memory representation and used to initialize an instance of the LLVM Execution Engine. The Execution Engine is then used to run the program in parallel to the main thread of the client to be able to alter the running program concurrently.

While the program is running, the variadic function `callAcc` is declared in it. `callAcc` is a wrapper function for an accelerated function call. Its arguments are an encoded form of the function signature of the to be called function and an indefinite number of additional arguments, completing the function call in conformance to the signature. `callAcc` is defined in the client runtime source file, externally accessible and outside of any class as shown in Listing 2.5. The `callAcc` definition in the client runtime just delegates the function call to an instance of a subclass of the `AbstractClient` class which handles the function call in a remote procedure call fashion. Which subclass the instance belongs to is determined by the command line arguments at

[2]`https://gcc.gnu.org/projects/gomp/`

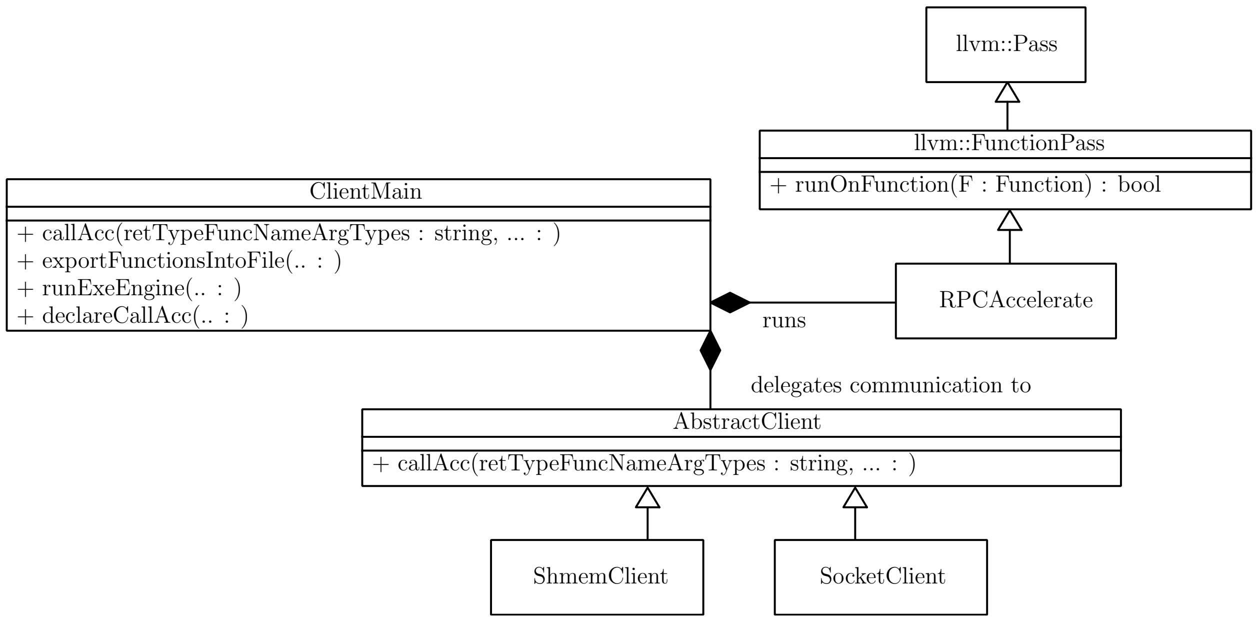
llvm::Pass
llvm::FunctionPass
+ runOnFunction(F : Function) : bool
ClientMain
+ callAcc(retTypeFuncNameArgTypes : string, ... :)
+ exportFunctionsIntoFile(.. :)
+ runExeEngine(.. :)
+ declareCallAcc(.. :)
runs
RPCAccelerate
delegates communication to
AbstractClient
+ callAcc(retTypeFuncNameArgTypes : string, ... :)
ShmemClient
SocketClient

Figure 2.3.: BAAR Client implementation overview

startup. Currently, it is either an instance of a class implementing communication over sockets (`SocketClient`) or an instance of a class implementing communication over shared memory (`ShmemClient`).

After the declaration of `callAcc`, compute intensive functions which should be executed by the BAAR Server are exported into an LLVM IR module, the remote part (see Section 1.3). Which functions to export is currently hardcoded, it is a goal of this thesis to choose functions by profiling and analyzing the running program. Once all chosen functions are exported, the connection to the server is established over the communication mechanism specified at client start and the remote part is transferred.

```
extern "C" void callAcc(const char *retTypeFuncNameArgTypes, ...) {
   // obtain list of arguments following retTypeFuncNameArgTypes
    va_list args;
    va_start(args, retTypeFuncNameArgTypes);

    AccClient->callAcc(retTypeFuncNameArgTypes, args);

    va_end(args);
}
```

Listing 2.5: Definition of variadic function `callAcc` in client runtime

Each function exported has to be altered for the original program to form the local part (see Section 1.3). This procedure is implemented in an LLVM Pass, which is explained in detail in the following section. Note that the user program is running concurrently in the Execution Engine while all of these steps happen. Figure 2.3 gives an overview of the modules the client implementation consists of.

`RPCAccelerate` Pass

The implemented LLVM Pass, named `RPCAccelerate`, is run on every function of the original program which were previously exported. This process transforms the program into the local part. The aim is to delegate every call to a function exported to the remote part to the server. It would be costly to wrap every call to such a function in the original program with the `callAcc` function, therefore the function definition itself is altered.

First, the function body is cleared completely. Afterwards, the Pass encodes the function signature into a string to pass as the first argument to the `callAcc` function. To support any possible return type without having to overload `callAcc` for every type, a variable of the functions return type is defined in the body of the function. It is passed to `callAcc` as a second argument and functions as a container for the return value. Now the call to `callAcc` is inserted, having the function signature as the first argument, the return value container as the second argument, and all the arguments of the function being processed as the rest of the arguments. Finally, a return is inserted into the function, either returning the value of the return value container or `void`. An exemplary output of the `RPCAccelerate` Pass when run on a function with the C signature `unsigned testfunc(short arrArg[][10], unsigned int arrArgSize)` is shown in Listing 2.6.

Communication

`callAcc` delegates the communication to an instance of a subclass of the `AbstractClient` class. Currently, there are two subclasses available `ShmemClient`, for handling communication over shared memory, and `SocketClient`, for handling communication over TCP/IP sockets. Both implementations are responsible for connecting to the server and sending the remote part. Additionally, they have to handle function calls by marshalling function name and arguments, as well as unmarshalling the result and arguments altered during the call.

Marshalling and unmarshalling of values is performed with type information available in the first argument of the `callAcc` call, the encoded function signature. Currently, primitive data types as well as arrays of any dimension with elements of a primitive data type are supported. Pointers

```
@.str3 = private unnamed_addr constant [31 x i8]
    c"10;32:testfunc:14;10;16:10;32\0A\00", align 1   // encoded
    signature

; Function Attrs: nounwind uwtable
define i32 @testfunc([10 x i16]* %arrArg, i32 %arrArgSize) #0 {
    %functionReturnValuePtr = alloca i32      // return value container

    call void (i8*, ...)* @callAcc(i8* getelementptr inbounds ([30 x
        i8]* @.str3, i32 0, i32 0), i32* %functionReturnValuePtr, [10 x
        i16]* %arrArg, i32 %arrArgSize)              // call client
        runtime callAcc

    %functionReturnValue = load i32* %functionReturnValuePtr
    ret i32 %functionReturnValue              // return value from
        container
}
```

Listing 2.6: Function declaration of an offloaded function with the C signature `unsigned testfunc(short arrArg[][10], unsigned int arrArgSize)` after `RPCAccelerate` was run on it (comments added)

```
for (const auto& currArg : argTypes) { // iterate over type information
   switch (currArg) {   // marshall argument according to type
      case llvm::Type::FloatTyID:
      // floats are promoted to doubles by variadic function callAcc
      case llvm::Type::DoubleTyID:
         sprintf(msg, ":%la", va_arg(args, double));
         // add marshalled argument to message, move to next
         break;
      case llvm::Type::IntegerTyID:
         switch (*intBWiterator++) {
         // integers are marshalled according to their bitwidths
            default: // other cases omitted
               sprintf(msg, ":%x", va_arg(args, int));
               // add marshalled argument to message, move to next
               break;
         }
         break;
      default:
         error("ERROR, LLVM TypeID " + currArg + " in function \"" +
            funcName + "\" is not supported");
   }
}
```

Listing 2.7: Excerpt from `SocketClient` class source code showing argument marshalling in a heavily shortened form (with added comments)

are always assumed to point to the first element of an array and are therefore treated as arrays. Arrays are transferred to and from the server as a whole. There is currently no mechanism to detect which elements of an array are actually read or altered. Furthermore, the argument following an array argument has to be an integer indicating the total number of elements in the array. A heavily shortened excerpt from the `SocketClient` class source code showing how some of the primitive data types are marshalled into a string to be send over a socket is displayed in Listing 2.7. When communicating over shared memory, the values are written to the shared memory area directly, without changing the representation.

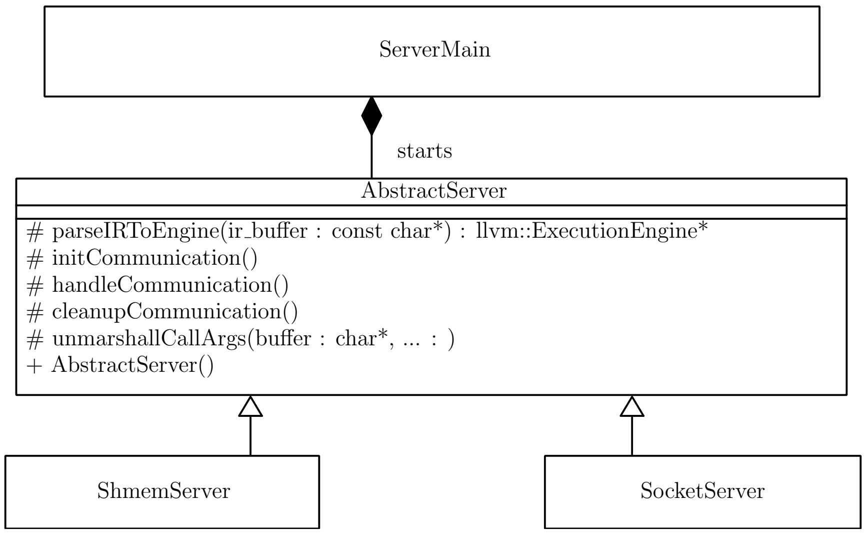

Figure 2.4.: BAAR Server implementation overview

2.3.2. BAAR Server

Overview

The server program is responsible for optimizing and executing code for the client. It is run from console, command line arguments specify which type of communication mechanism to use. Currently, communication over sockets and over shared memory is available. At startup, the server initializes the communication to enable the client to connect. Upon client connection, the client sends its remote part. The server accepts the remote part, verifies it to be correct LLVM IR code and initializes its backend with it. So far, the only backend available is running the code in an LLVM Execution Engine directly, without any prior optimization. It is a goal of this thesis to design and implement a modular way to perform optimizations and add further backends to the server. Once the backend is initialized, the server accepts remote procedure calls from the client and performs them. After any performed call, the result and potentially altered arguments are sent back to the client. Upon client disconnect, the communication is shut down cleanly.

Communication

The communication mechanisms are implemented in subclasses of the `AbstractServer` class, as depicted in Figure 2.4. If communication over sockets is chosen at startup, an instance of the `SocketServer` is initialized to handle the communication. Upon initialization, it creates a TCP/IP socket and the server starts listening on it. For every incoming client connection, a new thread is forked from the main thread to handle the communication with the client.

In the case of communicating over shared memory, an instance of the `ShmemServer` class is instantiated. It allocates a shared memory region upon initialization. Furthermore, two semaphores are allocated. One semaphore is used to signal events for the other communication party to wait on, the second is used to force the order of the protocol during communication. Currently, the server supports one client only in shared memory mode.

Upon client disconnect, the corresponding socket is closed and the shared memory as well as semaphores are freed, respectively.

While accepting and performing remote procedure calls from the client, values are marshalled and unmarshalled with type information from the remote part the client sent. Similar to the client, the server supports any combination of primitive data types as well as arrays of any dimension with elements of a primitive data type as arguments. Furthermore, any value return type is supported. Arrays are always transferred as a whole. Listing 2.7 displays a heavily shortened excerpt from the `SocketClient` class source code, showing how some of the primitive data types are marshalled into a string. Marshalling of arguments on the server side is a very similar procedure.

The BAAR Client and Server will be heavily extended in the following chapters.

2.4. Intel Xeon Phi

The Intel Xeon Phi coprocessor is the first product based on the Intel Many Integrated Core (Intel MIC) architecture [15]. It is a symmetric multiprocessor on-a-chip with up to 61 cores and a clock rate of up to 1.24 GHz per core. After its release in 2013, it quickly gained prominent use in the world's fastest supercomputers [6]. The Intel Xeon Phi Codename Knights Corner is available in the form of a PCI Express extension card, connecting the coprocessor chip to an Intel Xeon-based host system.

The x86-based coprocessor cores provide a subset of the Intel 64 instruction set [14]. There is no support for MMX, AVX or SSE instructions, but the newly introduced Intel Initial Many Core Instructions (IMCI) for 512-bit wide vector computations are supported. The vector processing unit executing IMCI instructions aims to support very high computational throughput for single and double precision calculations. The instruction pipeline follows a superscalar in-order design. Each core supplies four independent thread contexts and can execute two instructions per clock cycle.

To fully exploit the performance of the Intel Xeon Phi coprocessor, programs have to be vectorized and parallelized. The 512-bit wide vector units of the coprocessor support the execution of operators on many pairs of operands at once. Furthermore, the many cores on the coprocessor with the ability to execute two instructions per clock cycle offer massive performance for parallel programs. To utilize one core for over 90%, at least two threads have to be executed on it. Depending on the specific Xeon Phi product available, there are between 57 and 61 cores available. This means, to fully exploit the coprocessor performance, over 100 threads executing vectorized code have to be run. Fortunately, the Intel Xeon Phi supports well known software frameworks like OpenMP, MPI and OpenCL which aid parallelization.

Extensive information on the Intel Xeon Phi coprocessor its architecture and programming interface can be found in [15]. The coprocessor will be the target accelerator for executing the offloaded code throughout this thesis.

3. Implementation Of The Xeon Phi Backend

3.1. Introduction

The Intel Xeon Phi was chosen as the first accelerator to function as a target for the implementation of our approach to on-the-fly binary program acceleration. This choice was made, because its x86 based architecture allows porting of existent software just by recompiling it in many cases and its support of well known software frameworks like OpenMP makes us confident that its highly parallel architecture can effectively be utilized in our context. Therefore, code generation, optimization and estimation of speedups will focus on the Intel Xeon Phi throughout this thesis.

With the use of the LLVM Execution Engine, running programs in form of LLVM IR is very easy on targets for which a backend is available in LLVM. For the Intel Xeon Phi this is not the case at this writing. Furthermore, the Intel Compilers are currently the only compilers which can generate binaries for the Xeon Phi which utilize the powerful vector instructions introduced with the IMCI instruction set. However, the Intel Compilers do not support LLVM IR in any way. Therefore, to be able to offload code in the form of LLVM IR onto the Xeon Phi, a way of generating Xeon Phi binaries from LLVM IR code has to be designed and implemented. Intel announced to release a Xeon Phi backend to LLVM with the next generation of Xeon Phi accelerators, which are expected to arrive in the second half of 2015. This means that it would be wise to allow an easy transition to Intel's backend in the implementation of our approach.

A notable feature of the Intel Xeon Phi coprocessor is that it runs a dedicated Linux operating system on the card itself. This gives us the choice of putting the acceleration server on the host CPU or on the accelerator itself. The following Section 3.2 discusses how we placed the server on the accelerator itself and why.

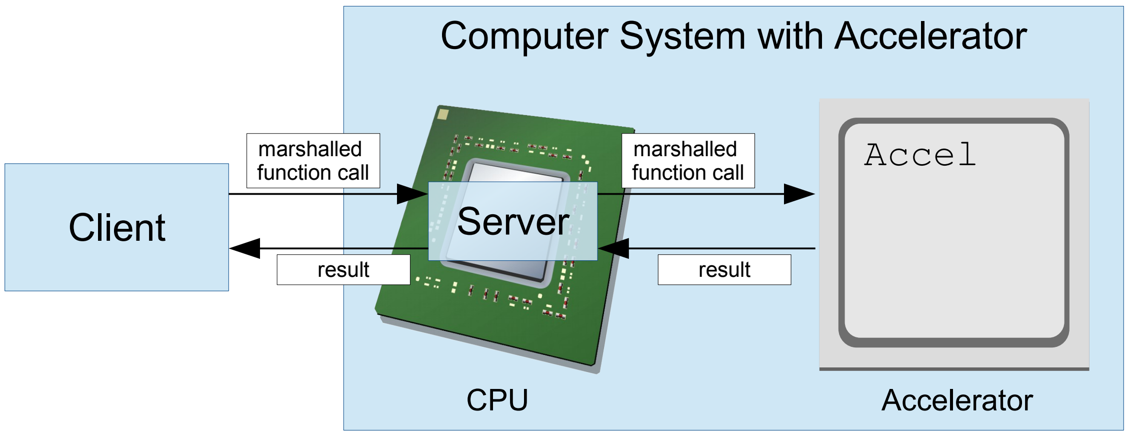

Figure 3.1.: Overhead when placing BAAR Server on CPU

3.2. Running The Server On The Xeon Phi

For implementing support for the Intel Xeon Phi as a target for the BAAR Server, the first choice to make is where to place the server. For many accelerators, the only feasible way would be to run the server on the host CPU, which typically does not share memory with the accelerator. In this case, the server would accept the LLVM IR module, perform optimizations and then generate a binary for the accelerator. Afterwards, the server would execute the binary on the accelerator and function as a proxy to run remote procedure calls from the client on the accelerator. This approach introduces a lot of data transfers. The program is transferred from the client to the server and from the server to the accelerator. More critically, any argument to a call has to take the detour over the server to get to the accelerator performing it and any result or altered argument has to take the detour back to the client. This introduces overhead, which could be avoided if the client could talk directly to the accelerator. This problem is depicted in Figure 3.1.

The Intel Xeon Phi coprocessor runs a dedicated Linux operating system, which can communicate to other systems directly, without taking a detour over the host CPU. Most importantly for our approach, it supports TCP/IP communication over sockets and can be contacted by other systems in the same network. Furthermore, the Intel Xeon Phi architecture supports a subset of the well known Intel 64 instruction set, which enables straight forward porting of existent software. These features enable us to place the BAAR Server directly onto the coprocessor and let it communicate with BAAR Client instances, possibly running on different systems in the same network, over TCP/IP sockets. Arguments sent from the client to the server would then directly be available for executing the remote procedure call without any further copying. This saves a lot of overhead, but introduces the problem of porting the BAAR Server and all of its dependencies to the Xeon Phi. As the Intel Compilers are only available on the host CPU, the whole software has to be cross-compiled. Because placing the server on the Xeon Phi results in a very clean software architecture and saves a lot of overhead, we choose this approach and tackle the problem of cross-compiling the BAAR Server and its dependencies, most notably LLVM, for the Intel Xeon Phi.

3.3. Preparations For Xeon Phi Support

So far, the only way for the BAAR Server to execute LLVM IR code was to instantiate an LLVM Execution Engine with the code and pass function calls to it. As the Xeon Phi is not supported as an LLVM code generation target and the Intel Compilers are currently the only way to get efficient binaries for it, we have to add an additional way for the server to execute LLVM IR. Several ways of execution may be added in the future. Therefore, a modular design is needed, allowing to choose which way to use at server startup.

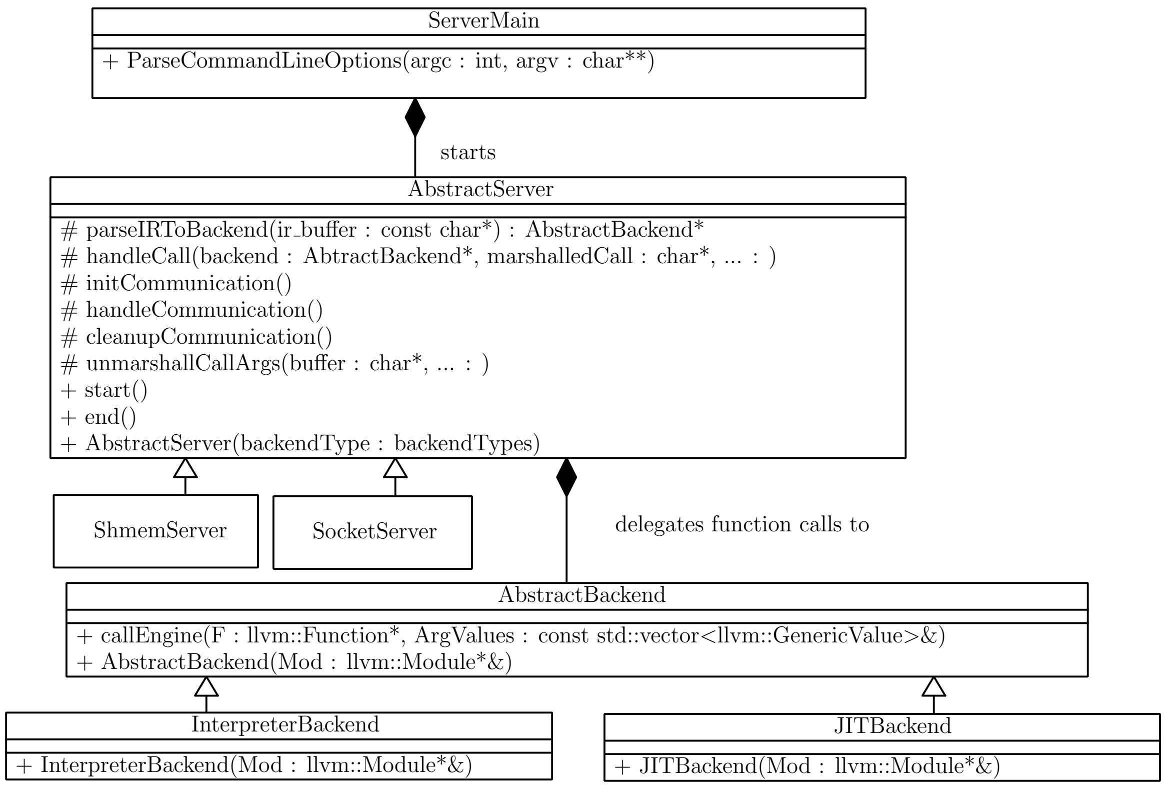
ServerMain
+ ParseCommandLineOptions(argc : int, argv : char**)
starts
AbstractServer
# parseIRToBackend(ir_buffer : const char*) : AbstractBackend*
# handleCall(backend : AbtractBackend*, marshalledCall : char*, ... :)
# initCommunication()
# handleCommunication()
# cleanupCommunication()
# unmarshallCallArgs(buffer : char*, ... :)
+ start()
+ end()
+ AbstractServer(backendType : backendTypes)
ShmemServer
SocketServer
delegates function calls to
AbstractBackend
+ callEngine(F : llvm::Function*, ArgValues : const std::vector<llvm::GenericValue>&)
+ AbstractBackend(Mod : llvm::Module*&)
InterpreterBackend
+ InterpreterBackend(Mod : llvm::Module*&)
JITBackend
+ JITBackend(Mod : llvm::Module*&)

Figure 3.2.: BAAR Server with server backends implementation overview

In the following, the "ways of execution" will be called *server backends*. Which server backend to use should be selectable via command line options; furthermore, any combination of communication mechanism and server backend should be supported. To cleanly implement this, the command line interface of the BAAR Server is rewritten using the LLVM Command Line Library[1], and additionally the server does not directly instantiate an LLVM Execution Engine anymore. An abstract class `AbstractBackend` is introduced which is the parent class of any server backend. It is initialized with the remote part in LLVM IR and offers interfaces to parse function names to receive function objects, as well as an abstract interface to perform function calls. The abstract interface and the initialization of the class is implemented in child classes, one child class for every implemented server backend. This design is depicted in Figure 3.2.

The `JITBackend` is one implementation of the `AbstractBackend` class; on instantiation, it just instantiates an Execution Engine with the LLVM IR code received from the BAAR Client. It implements the previous behavior of the server, therefore any function call is directly passed to the Execution Engine. As the just-in-time capabilities of the Execution Engine are not available for the Xeon Phi, a second server backend using the Execution Engine is implemented. The `InterpreterBackend` class implements the `AbstractBackend` class similar to the `JITBackend` class. In contrast to the `JITBackend` class however, the `InterpreterBackend` class instantiates the Execution Engine explicitly without just-in-time capabilities. This leads to very poor performance and lacks the ability to call native functions[2]. The `InterpreterBackend` is helpful for testing the offload mechanism on new targets where native code generation is not yet available, as is the case for the Intel Xeon Phi. Once the system as depicted in Figure 3.2 is cross-compiled to the Xeon Phi, initial tests can be run on it. Note that the `JITBackend` can be cross-compiled for the Xeon Phi, but fails to run LLVM IR due to the lack of machine code generation.

Several preparations have to be made to cross-compile BAAR to the Xeon Phi. First, LLVM has to be cross-compiled as the BAAR heavily depends on its libraries. It is recommended to use cross-platform make[3] (CMake) to do so. Intel provides CMake toolchain files to ease cross-compilation. The whole process is described in Appendix B.1.2. Once LLVM is available in cross-compiled form, BAAR itself can be cross-compiled. So far, BAAR was built in the LLVM source tree using the LLVM makefile system[4]. To support cross-compilation, the old makefiles are discarded and CMake files are implemented. Additional to cross-compilation, using CMake to build BAAR has several advantages: the project can be moved out of the LLVM source tree and be built with a precompiled LLVM distribution, required and optional dependencies can be checked before compilation, the architecture for which the project is build can be detected simply and specific operations can be performed. Being able to detect the architecture is especially important for having a single source base from which builds for several architectures should be

[1]`http://llvm.org/docs/CommandLine.html`

[2]Calling native functions is possible when LLVM is built with libffi which is introduced in Section 3.4.4

[3]`http://www.cmake.org/`

[4]`http://llvm.org/docs/MakefileGuide.html`

compiled. Cross-compiling the offload mechanism is explained in detail in Appendix B.2.2.

Once the whole project is cross-compiled for the Xeon Phi, the server can be run on it with the `InterpreterBackend`. To test the basic setup, LLVM and the BAAR also have to be built for the host machine, a standard x86 desktop computer in our case. The required steps are thoroughly explained in Appendix B.1.1 and Appendix B.2.1, respectively. Afterwards, simple calculations can already be offloaded from the BAAR Client on the host to the BAAR Server on the Xeon Phi. They must not reference any native functions and perform very poorly, but the general setup already works.

3.4. Native Xeon Phi Support

3.4.1. Introduction

As it is now possible to run the BAAR Server on the Intel Xeon Phi in interpretation mode, the next step is to extend it to generate native code for this target. Several problems have to be solved for this. As currently the only way to get well-performing code for the Intel Xeon Phi is to use the Intel Compilers, the offloaded LLVM IR code has to be transformed into a representation the Intel Compilers accept. Additionally, the Intel Compilers are not available on the Xeon Phi itself. Thus, the compiler on the system hosting the Xeon Phi accelerator card has to be utilized. The BAAR Server on the Xeon Phi has to tightly cooperate with the host system to obtain native code for the accelerator. Another problem is to combine the type information from the LLVM IR code with the native functions available as `void` pointers to be able to perform arbitrary function calls.

3.4.2. Generating C/C++ Code From LLVM IR

The Intel Compilers support compilation of C/C++ and Fortran code, therefore the offloaded code has to be transformed from LLVM IR representation to one of these languages. LLVM itself provided a Pass which transformed LLVM IR code to C code until the release of version 3.1 in May 2012 [3]. Unfortunately, it was removed due to lack of maintenance. As the Intel ispc project (see Section 1.2) is an LLVM based compiler which also targets the Intel Xeon Phi architecture, this project copes with the same problem our project faces: generated LLVM IR code has to be transformed to be compiled with the Intel Compilers. For this, the ispc team chose to revive and adapt the LLVM C backend for internal use. Fortunately, ispc is an open source project. This enables us to integrate the LLVM C backend in its ispc adapted and updated form

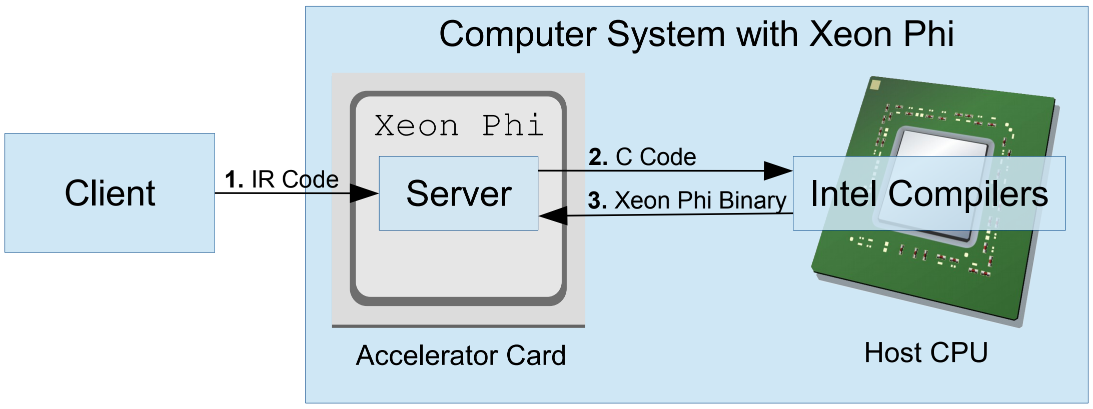

Figure 3.3.: High level view on the `ExtCompilerBackend` initialization process

into our project. Only minor modifications have to be made to resolve dependencies to the rest of the ispc project and integrate it cleanly. The result of these modifications is an independent software module which provides an interface to generate a `*.cpp` file from LLVM IR code.

3.4.3. Initialization Of The `ExtCompilerBackend`

The next step towards a native Xeon Phi server backend is to write a subclass for the `AbstractBackend` class which implements the initialization and the abstract function call interface using the modified ispc C backend. This subclass is called `ExtCompilerBackend`, as it will support more targets than our setup of the Xeon Phi eventually. The initialization is implemented in the constructor of the `ExtCompilerBackend` class. It gets LLVM IR code as an argument and directly transforms it into C/C++ code, which is written into a `*.cpp` file. For compiling the `*.cpp` file into native Xeon Phi code, the compiler of the system hosting the Xeon Phi accelerator card must be utilized. This involves the following steps:

1. Copy `*.cpp` file from Xeon Phi to host system
2. Initialize build environment and cross-compile code into shared object file
3. Copy shared object file from host system to Xeon Phi

A high level view on the initialization process is depicted in Figure 3.3. On a lower level, a shell script to perform these steps using Unix tools is generated at runtime in the `buildShellScript` member function of `ExtCompilerBackend`. `buildShellScript` is the only Xeon Phi specific part of the `ExtCompilerBackend` class. To generalize this server backend to be able to call external compilers in different setups, `buildShellScript` is extended to call the standard system compiler locally in the case of the server not running on the Xeon Phi. On which system the

```
std::string ExtCompilerBackend::buildShellScript(const std::string &export_name) { // export_name is the name of the exported C++ code
#ifdef _K1OM_ // when running on Xeon Phi, generate the following script
    return std::string("#!/bin/sh\n"
                       // get remote host
                       "remote_host=${SSH_CONNECTION%%' '*} \n"
                       // copy C++ code to host of Xeon Phi
                       "scp "+export_name+".cpp $remote_host:. \n"
                       // initialize build environment (setup specific)
                       "ssh $remote_host 'module add intel/compiler gcc/4.8.1 cmake/2.8.10.2 && "
                       // execute Intel Compilers remotely
                       "icc -w -mmic -o "+export_name+".so -shared -fPIC "+export_name+".cpp && "
                       // copy Xeon Phi binary from host to accelerator
                       "scp "+export_name+".so '\"$(hostname):.\"' \n'");
#else // when not running on Xeon Phi, generate the following script
    return std::string("#!/bin/sh\n"
                       // just run local standard C++ compiler
                       "c++ -w -o "+export_name+".so -shared -fPIC "+export_name+".cpp \n");
#endif
}
```

Listing 3.1: Definition of `ExtCompilerBackend::buildShellScript(..)`, generating the shell script to call an external compiler (comments added)

server is running on is determined by a preprocessor macro which the CMake scripts define when configuring the build. The function with this generalized behavior is shown in Listing 3.1.

The shell script generated by `buildShellScript` is executed by the constructor using a system call, immediately after generating the `*.cpp` file. Details on this process and how to start the acceleration server on the Xeon Phi can be found in Appendix C. Once the native code is available on the Xeon Phi in the form of a shared object file, the `ExtCompilerBackend` constructor loads this file dynamically into memory. It will provide the functions to call for executing remote procedure calls for the client.

This concludes the initialization of the `ExtCompilerBackend` class. The steps taken to obtain native Xeon Phi code from LLVM IR may seem cumbersome, they provide efficient code and fulfill our demands however. Once LLVM gains support for Xeon Phi code generation, this process can simply be replaced by running the BAAR Server with the already existent `JITBackend`.

3.4.4. Performing Function Calls With The `ExtCompilerBackend`

After the initialization of the `ExtCompilerBackend`, the server is ready to accept calls from the client. On arrival of a call, arguments are parsed into LLVM data types. The whole type information is available in LLVM's form of representation in the LLVM IR code the client sent for initializing the acceleration. This means that for executing the call, the gap from the call information – available in LLVM data types – to the native function – available as a symbol in the shared object – has to be bridged. As the function signature is only known at runtime and the `ExtCompilerBackend` is written in C++, a statically typed language, a *foreign function interface* is needed. A foreign function interface allows to call functions in a language which were written in another, it abstracts from *calling conventions*. A calling convention is a set of assumptions made by a programming language's compiler on where to find arguments on function entry and where to put the result. [23] In our case, we call a native function available as a `void` pointer in C++ with type information available in LLVM IR. For this purpose we will use libffi[5], a widely used foreign function interface library. LLVM can also optionally be built with libffi, allowing the Execution Engine to call native functions when interpreting LLVM IR without the availability of just-in-time compilation. Therefore, the Execution Engine provides functions to transform values and types in LLVM IR into libffi representations. These functions can perfectly be utilized for the `ExtCompilerBackend`. The whole process of executing a call works as follows in the `ExtCompilerBackend`:

1. Load function as a symbol from the shared object, results in an untyped (`void`) pointer to the function
2. Convert type information and values in LLVM IR into libffi representation
3. Call the function with libffi
4. Convert the result in libffi representation into LLVM IR and return it

With this mechanism in place, the BAAR Server provides native code generation for the Xeon Phi from LLVM IR code. As BAAR now supports offloading LLVM IR code onto a real accelerator, the next step is to automate the decision which parts of the code to offload. The following

[5]`http://sourceware.org/libffi/`

chapter deals with this problem.

4. Automatic Identification Of Suitable Function Calls

So far, the functions to be offloaded to the BAAR Server for acceleration were hardcoded. This chapter deals with the problem of extending the client to automatically identify functions suitable for the accelerator. The following two sections deal with detecting compute intensive functions and estimating their suitability for the accelerator. Once these steps are finished, it is safe to assume that the functions can be optimized to run efficiently on it. However, some remote calls to these functions on the server may still cause slowdowns when more time is spent to transfer arguments and results between client and server than is saved by being able to perform the computations more efficiently. Therefore, the third section investigates in trading off heaviness of computations against the size of arguments and results.

4.1. Gathering Candidates By Profiling

As a first step towards identifying function calls suitable to be offloaded to the Intel Xeon Phi, a set of candidate functions is chosen. This set should only contain promising candidates to reduce processing power needed for the following heavier analysis. However, any function which would be identified as suitable for the accelerator in the following analysis steps should also be contained in the candidate set. As functions offloaded to the accelerator are parallelized with polyhedral optimization, candidate functions should contain loops with high iteration counts. Loops with high iteration counts result in very frequently executed basic blocks. Therefore, the LLVM Pass `BlockFrequencyInfo` is run on every function defined in the running program module. The information gathered by this analysis Pass is used to calculate the expected number of times any basic block is executed per entry to the currently analyzed function. If the maximum number of times a basic block is executed reaches a defined threshold, the function containing this basic block is added to the list of candidates for acceleration. The threshold can be defined with a command line argument on BAAR Client startup, its default value is set to

100 so that functions containing loops are quickly added to the candidates set. How this process is implemented is shown in Listing 4.1.

Once the maximum basic block frequency of any defined function in the program module is calculated, the gathering of candidates for acceleration finishes. The candidates are more thoroughly analyzed as described in the following section.

4.2. Scoring Functions For Suitability

The set of candidate functions for acceleration, as described in the previous section, is further analyzed to determine their suitability for the accelerator more precisely. The idea is to determine a score for every function in the set. This score should give an abstract value of suitability. It is not necessary to estimate the speedup, the only information we need is whether a function is expected to run faster when offloaded. By default, a function with a score greater than zero is offloaded to the accelerator and further optimized by the BAAR Server. This threshold can be configured with a command line parameter. The score is additionally used to make runtime decisions based on its value and actual parameters to determine whether to execute a call remotely or locally. This procedure is explained in the following section.

For scoring the candidate functions, a new Pass called `AccScore` is introduced. It is run separately on every candidate function. At first, it utilizes Polly's `SCoPDetection` Pass to check if the function contains at least one SCoP (see Section 2.2.5). SCoPs are the entities Polly's polyhedral optimizations are performed on. As the code has to be parallelized to run efficiently on the Intel Xeon Phi and we rely on polyhedral optimization for this, static control parts are crucial for getting speedups when offloading functions. Therefore, a function without any static control part is assigned a score of zero and not further analyzed or offloaded.

A function containing at least one static control part is further analyzed. The `AccScore` iterates over every detected static control part and in every static control part over every contained loop nest. For every loop nest, the total numbers of floating point and integer operations are determined separately. These two counts can be weighted by command line arguments, the standard weight is one for both counts. The weighted counts are added and the result multiplied to the innermost basic block frequency of the current loop nest to give a weighted estimate of total floating point and integer operations performed when the current loop is executed. To determine the innermost basic block frequency, the LLVM Passes `LoopInfo` and `ScalarEvolution` are utilized. The weighted estimate of total floating point and integer operations defines the loop nest's score. The static control part's score is the sum of loop nest scores, and the function's score

```
// instantiate Pass Manager for the running program module
llvm::FunctionPassManager BlockFreqAnalysisPM(ProgramMod);
// add prerequisites for BlockFrequencyInfo Pass to Pass Manager
BlockFreqAnalysisPM.add(llvm::createLoopSimplifyPass());
// instantiate BlockFrequencyInfo Pass and add to Pass Manager
llvm::BlockFrequencyInfo* BFIPass = new llvm::BlockFrequencyInfo();
BlockFreqAnalysisPM.add(BFIPass);

// instantiate list for candidates
std::list<llvm::Function*> accelerationCandidates;
// iterate over any function in the running program
for (llvm::Module::iterator I = ProgramMod->begin(); I !=
    ProgramMod->end(); I++) {
    if (I->getName().compare("main") == 0) // do not try to accelerate
        main
        continue;
    // run Passes on current function
    BlockFreqAnalysisPM.run(*I);

    // get maximum absolute basic block frequency of function's basic
        blocks
    unsigned maxBBFreq = 0;
    for (llvm::Function::iterator J = I->begin(); J != I->end(); J++) {
        auto currFreq = BFIPass->getBlockFreq(J).getFrequency() /
            BFIPass->getBlockFreq(J).getEntryFrequency();
        if (maxBBFreq < currFreq)
            maxBBFreq = currFreq;
    }

// if function contains a basic block executed more frequently than
    threshold, add function to candidates
if (maxBBFreq > BBFreqThreshold)
    accelerationCandidates.push_back(&(*I));
}
```

Listing 4.1: Excerpt of client runtime code showing how candidate functions are gathered (shortened, added comments)

is the sum of static control part scores. The implementation of the function scoring is shown in Listing 4.2, Equation 4.1 gives a more concise representation of how the score is calculated.

$$\begin{aligned}\text{score}_{\text{loop}} &= (c_{\text{IOPs}} \cdot \text{IOPs}_{\text{loop}} + c_{\text{FLOPs}} \cdot \text{FLOPs}_{\text{loop}}) \cdot \text{innerBBFreq}_{\text{loop}} \\ \text{score}_{\text{function}} &= \sum_{\text{SCoP} \in \text{function}} \sum_{\text{loop} \in \text{SCoP}} \text{score}_{\text{loop}}\end{aligned} \tag{4.1}$$

The score is immediately used to drop candidates which cannot be parallelized (score equals zero) or which contain loops with too few compute intensive operations (low score, e.g., initializations of arrays). The scores of the remaining candidate functions are stored. These functions will be offloaded. The score will further be used to insert runtime decisions into these functions to determine whether to offload or to locally execute a certain call. The insertion of the runtime decisions is explained in the following section.

4.3. Runtime Decisions Based On Score And Argument Size

Despite the fact that the chosen functions from the candidate set are considered as suitable for the accelerator, it can still make sense to perform certain calls to these functions locally. This is the case, when the time taken to transfer arguments to and results from the accelerator is bigger than the time saved by running optimized code on it. Therefore, it makes sense to be able to make the decision whether to execute a call to an offloaded function locally or remotely at runtime, depending on the size and type of the arguments.

So far, the `RPCAccelerate` Pass run by the BAAR Client just clears the definition of offloaded functions and inserts an appropriate call to the accelerator. This means that any call to an offloaded function is executed remotely. Thus, the `RPCAccelerate` Pass has to be extended to insert a runtime decision based on score and actual arguments. Conceptually, the resulting function body of a function called `func` after the extended `RPCAccelerate` Pass was run on it should conceptually look like shown in Listing 4.3.

This means, the previously determined score of `func` is compared to the weighted size of the total bytes transferred in a remote call between client and server. If the score is bigger than the weighted number of bytes, the call is executed remotely. If this is not the case, the old function body is executed locally. Instead of comparing the score to the weighted total bytes transferred, this can also be seen as comparing the fraction of the score over the weighted total bytes transferred to a constant c as shown in Equation 4.3. From this point of view, the weight c gives a threshold of how many score points per bytes transferred a function call has to have

```
// get information gathered by Polly's SCoPDetection Pass
polly::ScopDetection &SCoPDetect = getAnalysis<polly::ScopDetection>();
if (SCoPDetect.begin() == SCoPDetect.end())
   score = 0; // set function score to zero if no SCoP could be detected
else {
   // get information gathered by LLVM's LoopInfo Pass
   LoopInfo &LoopInf = getAnalysis<LoopInfo>();
   // iterate over detected SCoPs
   for (auto Region = SCoPDetect.begin(); Region != SCoPDetect.end();
      Region++) {
      unsigned long regionScore = 0;
      // iterate over loop nests in current SCoP
      for (LoopInfo::iterator Loop = LoopInf.begin(); Loop !=
         LoopInf.end(); Loop++) {
         // calculate execution count of loop nest's innermost basic
            block
         unsigned long LoopInnermostTotalTripCount =
            getInnermostTotalTripCount(**Loop);

         unsigned totalLoopFLOPs = 0;
         unsigned totalLoopIOPs = 0;
         // count IOPs and FLOPs for current loop nest
         for (const auto& Block : (*Loop)->getBlocks()) {
            // determine OP counts of current basic block (Visitor
               pattern)
            visit(Block);
            totalLoopFLOPs += lastBB_FLOPs;
            totalLoopIOPs += lastBB_IOPs;
         }
         // calculate current loop nest's score
         unsigned long loopScore = LoopInnermostTotalTripCount *
            (IOPsWeight*totalLoopIOPs + FLOPsWeight*totalLoopFLOPs);
         // accumulate to get region (SCoP) score
         regionScore += loopScore;
      }
      // accumulate to get function score
      score += regionScore;
   }
}
```

Listing 4.2: Excerpt of `AccScore` Pass implementation (shortened, added comments)

```
bytesTransferred = argumentSize + resultSize
if (funcScore > c*bytesTransferred)
   callAcc(func, arguments);
else
   oldfunc(arguments);
```

Listing 4.3: Conceptual scheme of a function body after RPCAccelerate was run on it

to be offloaded.

$$\text{funcScore} > c \cdot \text{bytesTransferred} \tag{4.2}$$

$$\Leftrightarrow \frac{\text{funcScore}}{\text{bytesTransferred}} > c \tag{4.3}$$

To achieve this behavior, the `RPCAccelerate` Pass is extended to insert two new basic blocks in front of the old function body instead of clearing the function. The first basic block, now the first of the function, is filled with operations to calculate the total argument size in byte. The total size of primitive arguments can be calculated directly by the `RPCAccelerate` Pass. The size of arrays has to be calculated at runtime. For this, the existing limitation that the argument after an array is an integer indicating the total number of elements is exploited. For every array, instructions are inserted to multiply the argument following the array in the argument list to the size of the element type. Further instructions are inserted to add the array sizes and the size of the primitive arguments. The result of these instructions is the total size of arguments in bytes. To finish the size calculation, instructions to add the result type size and the total array size a second time is inserted to obtain the total number of bytes which have to be transferred between BAAR Client and Server during a remote call. The total array size is added twice, as the communication mechanism currently always transfers arrays to and from the server completely. Note that a more intelligent communication mechanism would only transfer changed array entries back to the client after the call. In this case, the total number of bytes which have to be transferred between client and server could be reduced significantly in many cases. After the size calculation, the result is multiplied by the weight c, which can be set by a command line argument. By default, the weight is set to zero to force any function call to be executed remotely. After the argument size calculation, the first basic block ends with a conditional branch. In the case of the weighted size being smaller than the function score, the following – still empty – basic block is executed. In the opposite case, the function is executed locally by branching to the old function body.

The second basic block of the function is the true branch in the scheme in Listing 4.3. It is filled with the exact same instructions, with which the initial `RPCAccelerate` Pass implementation filled the cleared out function body. These instructions call the `callAcc` function to execute a remote call of the current function. Additionally, the return of the remotely calculated result is ensured. This basic block is executed in the case that the weighted total bytes to be transferred are smaller than the function score. An exemplary output of the extended `RPCAccelerate` Pass is shown in Listing 4.4. This concludes the extension of the `RPCAccelerate` Pass. The extensions to the BAAR Client described in this section enable it to make the decision whether to execute the function call locally or remotely at runtime.

This chapter described the automatic detection of suitable function calls for remote execution on the accelerator in three sections: Section 4.1 introduced the gathering of candidate functions which are further analyzed and scored as discussed in Section 4.2. Finally, Section 4.3 described how the score is used to insert operations which allow the BAAR Client to make the decision whether to execute remotely or not per call. So far, remote execution on the BAAR Server is unlikely to achieve any speedups as the server does not optimize the offloaded code at all. The following chapter tackles this problem.

```
// encoded function signature
@.str1 = private unnamed_addr constant [23 x i8]
    c"0:testfunc:14;3:10;32\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @testfunc(double* %fin, i32 %elements) #0 {
  // calculate total array size (only one array argument)
  %arrayNumElementsArgExt = sext i32 %elements to i64
  // double 8 byte wide, multiply element count by 16 to count array
      twice
  %arraySizeInByte = mul i64 16, %arrayNumElementsArgExt
  %totalArraySizeInByte = add i64 0, %arraySizeInByte
  // add return type size (0 byte), total array size and total
      non-array argument sizes (4 byte) to get total argument size
  %totalArgSizeInBytes = add i64 %totalArraySizeInByte, 4
  // compare score total argument size
  %1 = icmp uge i64 2107023936, %totalArgSizeInBytes
  // branch to remote call if score >= total argument size, to original
      function body otherwise
  br i1 %1, label %3, label %4

// remote call
; <label>:3                                       ; preds = %0
  call void (i8*, ...)* @callAcc(i8* getelementptr inbounds ([22 x i8]*
      @.str1, i32 0, i32 0), double* %fin, i32 %elements)
  ret void

// original function body
; <label>:4                                       ; preds = %0
// ...
```

Listing 4.4: Excerpt of an offloaded function's declaration having the C signature `void testfunc(double* fin, int elements)` after the extended `RPCAccelerate` Pass was run on it (shortened, comments added)

5. Optimizing The Remote Part On Server Side

The general idea for the optimization of offloaded code in our approach is that the client only knows the characteristics of the accelerator the server provides. With these characteristics, e.g., vector width, number of hardware threads or communication speed, the client chooses parts of the running program to offload. The client itself does not optimize the code in any way however, this is the server's task. The server knows the accelerator in detail, it knows its architecture, its current load and possibly communicates with vendor specific libraries to utilize it. The additional information the server has, is important for optimization. In contrast to the client, the server can perform time consuming optimizations without impairing the overall progress. Simultaneously, the client runs the original program and still makes progress. Once the server is done, the client transforms the original program into the local part and makes use of the accelerated remote execution.

So far, the BAAR Server did not perform any transformations on the offloaded IR code. Furthermore, no mechanism to notify the client once the remote execution can be utilized was available. The following sections explain the implementation of these parts.

5.1. Server Side Optimization

Once the offloaded LLVM IR code is received on the server side, the BAAR Server uses LLVM libraries to parse the textual representation into an in-memory form. The result is an LLVM IR *Module*, it is verified and can be transformed and analyzed with Passes afterwards. For this purpose, a new function `optimizeModule(..)` is defined in the `AbstractServer` class (See Figure 3.2) to be available for any communication and backend combination. `optimizeModule(..)` is immediately called after verifying the Module. This function will eventually take care of the whole optimization of the Module. After the return of `optimizeModule(..)`, the server backend is initialized with the optimized Module in the same fashion as implemented previously. To enable the BAAR Client to run the original program code while the server optimizes the module and use the remote part once the server is done, the server has to notify the client when it is ready. This is implemented by letting the client simply wait for a message from the server in a separate thread. More specifically, implementations of the abstract `initializeAccelerationWithIR(..)` function declared in the `AbstractClient` class is required to return only when the server signals to be ready or an error occurrs from now on. Respective implementations for different communications mechanism have to ensure this behavior and handle errors in a reasonable way. The currently available implementations for communication over shared memory and sockets are extended to fulfill the new requirements.

For our chosen accelerator, the Intel Xeon Phi, it is crucial to have many threads (optimally more than a hundred) running vectorized code. Therefore, the offloaded code has to be parallelized and vectorized in the `optimizeModule(..)` function by the server. The implementation of these optimizations and the specifics to enable good results on the Intel Xeon Phi are explained in the following sections.

5.2. Parallelization

Parallelization is performed before vectorization to first use exploitable parallelism to distribute calculation among as many threads as possible. The Intel Xeon Phi provides up to 61 cores, each able to run four threads in parallel. Therefore, distributing the calculation among many threads is crucial. Polly can detect SCoPs in the offloaded LLVM IR code, models them in polyhedral representation and generates code with more parallelism exposed. Using Polly's OpenMP code generation, this parallelism is exploited on the thread level. Polly's OpenMP code generation transforms loop bodies in detected SCoPs into subfunctions and inserts OpenMP library calls into the function to distribute execution of iterations among a number of threads. The number of threads used by the OpenMP library can be defined by the `OMP_NUM_TREADS` environment variable, which we will set to 240 for the Xeon Phi with 61 cores. This ensures enough resources to be available for the operating system. The code generated by Polly's OpenMP code generation has to be linked to an OpenMP library implementation to be able to be run. Polly inserts OpenMP library calls for the GNU implementation of the OpenMP library, GOMP[1]. So far, GOMP is not available for the Xeon Phi. Fortunately, the Intel OpenMP library implementation running on the Xeon Phi is binary compatible to GOMP. Therefore, IR code parallelized with Polly's OpenMP backend can be run by the server on the Xeon Phi without any modifications.

Parallelization on the server side is enabled by implementing the so far empty `optimizeModule(..)`

[1]`https://gcc.gnu.org/projects/gomp/`

function in the `AbstractServer` class. Polly's code generation passes require some analyses and simplifications to be performed previously. Thus, the following Passes [5] are run before Polly:

- **Alias Analysis**, a class of Passes which analyze pointers. An alias analysis Pass tries to determine if two pointers may point to the same object in memory. As Polly introduces additional parallelism, this is a crucial analysis. The specific LLVM alias analysis Passes run by the server are the `TypeBasedAliasAnalysisPass` and the `BasicAliasAnalysisPass`
- **Promote Memory To Register**, a Pass that promotes memory references to register references. This is especially important in unoptimized code, where simple variable uses are often present as a sequence of loads and stores. As memory accesses complicate parallelization considerably and may even cause it to fail, this is an important Pass
- **Loop Simplify**, a Pass that transforms natural loops into a simpler form. It simplifies following analyses and transformations and makes them more effective. This Pass is a requirement for Polly's code generation Passes
- **Induction Variable Simplification**, a Pass that analyzes and transforms induction variables and derived computations into simpler forms, making subsequent analyses and transformations simpler. An additional requirement for Polly's code generation Passes

All of these Passes are added to a Pass Mangager instantiated in the `optimizeModule(..)` function. Afterwards, Polly's code generation Pass can be run. Polly provides a command line argument to enable the code generation to emit OpenMP parallelized code. After Polly's Passes, additional Passes to simplify the control flow graph and to try to combine instructions are run on the Module. Finally, the resulting Module is verified and the optimization finishes. After the optimization, the backend can be initialized as described in Section 3.3. For testing purposes, the whole parallelization process can be disable with a command line argument when starting the server. The resulting `optimizeModule(..)` function is shown in Listing 5.1.

If Polly is available and its requirements are known, running its Passes and using it for parallelization on suitable code is as easy as described in this section. Getting Polly to run on the Xeon Phi takes some effort however. Cross-building LLVM with Polly is described extensively in Section B.1.2

5.3. Vectorization

After parallelization, vectorization is performed on the offloaded LLVM IR code. LLVM supplies two powerful vectorization passes: The *Loop Vectorizer* and the *SLP Vectorizer*. [1] The Loop

```
void AbstractServer::optimizeModule(llvm::Module *Mod) {
    llvm::PassManager AutoParVectPasses;

    if (!DisablePolly) {
      // polly preparation passes
      AutoParVectPasses.add(llvm::createTypeBasedAliasAnalysisPass());
      AutoParVectPasses.add(llvm::createBasicAliasAnalysisPass());
      AutoParVectPasses.add(llvm::createPromoteMemoryToRegisterPass());
      AutoParVectPasses.add(llvm::createLoopSimplifyPass());
      AutoParVectPasses.add(polly::createIndVarSimplifyPass());
      // polly passes
      AutoParVectPasses.add(polly::createCodeGenerationPass());
      // cleanup passes
      AutoParVectPasses.add(llvm::createGlobalOptimizerPass());
      AutoParVectPasses.add(llvm::createCFGSimplificationPass());
      AutoParVectPasses.add(llvm::createInstructionCombiningPass());
    }

   // run parallelization and verify resulting IR code
    AutoParVectPasses.run(*Mod);
    llvm::verifyModule(*Mod, llvm::PrintMessageAction);
}
```

Listing 5.1: `optimizeModule(..)` defined in `AbstractServer` showing the Passes needed for paralellization

Vectorizer provides features which allow complex loop vectorization. It supports numerous powerful features like vectorizing loops with trip counts only known at runtime, inserting runtime checks to determine if pointers point to overlapping regions, vectorizing accumulation variables (e.g., sums), flattening `if` statements, vectorizing mixed types and more. The superword-level parallelism vectorizer, short SLP Vectorizer, focuses on combining similar independent instructions into vector instructions. It works on memory accesses, arithmetic operations, comparison operations and PHI-nodes. Both vectorizers can be forced to emit vector instructions of a certain vector width with a command line argument. This is important for targets which are not supported for code generation in LLVM, as is the case for the Intel Xeon Phi.

To enable vectorization on the server side, the `optimizeModule(..)` function in the `AbstractServer` class is extended. For the vectorization passes to run effectively, some preparations have to be made. Most importantly, the vectorization passes rely on target-specific information about available vector instructions, e.g, vector width and cost, which have to be gathered. For this, the LLVM class `TargetMachine` has to be instantiated for the current target. The `TargetMachine` class provides interfaces to all target-specific information, giving a complete machine description. It also provides an interface to conveniently add all necessary target-specific analysis passes to a Pass Manager. Getting an instance of the `TargetMachine` for the desired target requires a sequence of steps. First, the target triple, giving a description of the desired target has to be made accessible. It can be created from a string representation or, as in our case, be obtained from the LLVM IR Module. The LLVM `TargetRegistry` class provides a library of available targets, it is used to look up the respective `Target` instance for our target triple. The `Target` instance can then be used to create a target machine instance, information like special sub-target features and additional options can be provided in this process.

The very first Passes the `optimizeModule(..)` function adds to the Pass Manager now are the target and data layout specific analysis passes obtained from the `Module`, `TargetTriple` and `TargetMachine` class instances. Obtaining all of these class instances and Passes fails when LLVM does not support the desired target. This is the case for our desired target, the Intel Xeon Phi. To overcome this, the 64bit x86 target is used to provide the information we need for vectorization. For this, the x86 target has to be compiled without just-in-time compilation support for the Xeon Phi, as cross-compilation fails otherwise. This process is described in Appendix B.1.2. Building the x86 target with just-in-time compilation support fails for the Xeon Phi, because it makes use of assembler instructions the Xeon Phi does not support. To get 512bit vector instructions suitable for the Xeon Phi eventually, command line arguments to set the vector width provided by the vectorization passes have to be used when starting the BAAR Server.

Similar to Polly's code generation Passes, the vectorization Passes require some analyses and simplifications to be performed previously. In fact, the vectorization Passes need all the preparation Passes listed in Section 5.2 but the Induction Variable Simplification Pass to be run as a prerequisite.

Once these steps are finished, all the preparations for the Loop Vectorizer and SLP Vectorizer Passes are made and the Passes can be run on the LLVM IR Module. Similar to the parallelization with Polly, Passes to simplify the control flow graph and to try to combine instructions are run as a cleanup procedure for the Module afterwards. It is also possible to disable the whole vectorization process with a command line argument. The resulting implementation of `optimizeModule(..)` is shown in Listing 5.2.

It is important to note that the vector instructions inserted by the vectorizers have to be supported by the LLVM backend for the desired target. For our target, the Intel Xeon Phi, we take a detour over the C backend which's output is compiled with the Intel Compilers. Therefore, the C backend has to support vector instructions. Fortunately, the ispc team extended the C backend to support vector instructions in a flexible way. For any vector instruction, a function call for the specific instruction is emitted. The definitions of these functions are implemented in target-specific header files. Optimally, a function definition of this kind consists of one intrinsic for the desired target only. If the target architecture does not support an LLVM vector instruction natively, the respective function definition may consist of several lines of code. There are also header files for generic x86 targets, which implement the LLVM vector instructions with loops due to the lack of a vector unit. The Intel Xeon Phi is also supported by the header files provided by ispc. This gives us a way to get native Xeon Phi code not only from scalar, but also from vectorized LLVM IR code without any extra effort over the implementation described in Section 3.4.

This finishes the description of the optimizations performed on the server side of BAAR. The previous chapters extended BAAR to be able to automatically identify, offload and optimize functions to run them natively and efficiently on the Intel Xeon Phi. The following chapter deals with the evaluation of these extensions and BAAR in general.

```
void AbstractServer::optimizeModule(llvm::Module *Mod) {
    llvm::PassManager AutoParVectPasses;
    // prepare target-specific analyses
    TargetLibraryInfo *TLI = new
       llvm::TargetLibraryInfo(Triple(Mod->getTargetTriple()));
    AutoParVectPasses.add(TLI);
    AutoParVectPasses.add(new llvm::DataLayout(Mod->getDataLayout()));

   // obtain TargetMachine instance
    llvm::Triple ModuleTriple(Mod->getTargetTriple());
    llvm::TargetMachine *Machine = 0;
    if (ModuleTriple.getArch())
      Machine = GetTargetMachine(Triple(ModuleTriple));
    llvm::OwningPtr<llvm::TargetMachine> TM(Machine);

    if (TM.get()) // add target-specific analysis Passes
      TM->addAnalysisPasses(AutoParVectPasses);
    else
        std::cout << "INFO: " << "No TargetMachine detected\n";

        // [...] Polly Passes omitted here, see previous section

    if (!DisableVectorization) {
        // vectorization preparation passes
        AutoParVectPasses.add(llvm::createTypeBasedAliasAnalysisPass());
        AutoParVectPasses.add(llvm::createBasicAliasAnalysisPass());
        AutoParVectPasses.add(llvm::createPromoteMemoryToRegisterPass());
        AutoParVectPasses.add(llvm::createLoopSimplifyPass());
        // vectorization passes
        AutoParVectPasses.add(llvm::createLoopVectorizePass());
        AutoParVectPasses.add(llvm::createSLPVectorizerPass());
        // cleanup passes
        AutoParVectPasses.add(llvm::createCFGSimplificationPass());
        AutoParVectPasses.add(llvm::createInstructionCombiningPass());
    }
    AutoParVectPasses.run(*Mod); // run optimizations
    llvm::verifyModule(*Mod, llvm::PrintMessageAction); // verify result
}
```

Listing 5.2: Final `optimizeModule` implementation in `AbstractServer` (commented, shortened)

6. Evaluation

With the extensions made as described in the previous chapters, BAAR was developed to be a prototype of an easy-to-use on-the-fly binary program accelerator targeting the Intel Xeon Phi many-core coprocessor. This chapter discusses the very first evaluation of BAAR. Section 6.1 discusses BAAR's abilities and limitations to score and offload certain functions. Section 6.2 introduces a class of well-fitting computations and thoroughly evaluates the performance of BAAR when executing an example of this class. Section 6.3 evaluates BAAR's mechanism to identify function calls to execute remotely.

6.1. Abilities And Limitations Identified

BAAR automatically identifies functions containing SCoPs with heavy computations. These functions are then automatically parallelized, vectorized and compiled for the Intel Xeon Phi. Functions containing SCoPs can therefore utilize the computational power of the Intel Xeon Phi through BAAR, without any assistance by the user. As BAAR relies on Polly for parallelization, SCoPs are a requirement for functions accelerated with it, however. Additionally, as BAAR is still in a proof-of-concept state, it introduces its own limitations, e.g., global values are only copied once to the accelerator at the current state and are not kept in sync with the local part running on the client. This section explains important limitations which were identified during evaluation.

6.1.1. Alias Analysis

As already mentioned in Section 5.2, it is crucial to know whether two pointers involved in a calculation may point to the same value when introducing additional parallelism. Only if all pairs of pointers involved in a calculation point to separate memory regions, it can be guaranteed that the calculation produces the same result after parallelization and vectorization were applied. Therefore, the alias analysis gives conservative results. If it cannot be proven that two pointers do not alias under any circumstance, it is assumed that they do alias. This causes parallelization

```
#define N 100
void matmul(double A[N][N], int elementsA, double B[N][N], int
    elementsB, double C[N][N], int elementsC) {
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)  {
            C[i][j] = 0;
            for(int k = 0; k < N; k++)
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
}
```

Listing 6.1: Matrix multiplication example which fails alias analysis

and vectorization to fail in many cases which may seem clearly parallelizable to the programmer.

An example which seems easily parallelizable but fails LLVM's Basic Alias Analysis Pass is shown in Listing 6.1. It shows a simple multiplication of two matrices, the result is stored in a third matrix. Although matrix `A` and `B` are obviously only read and the calculation could correctly be parallelized to run on a separate thread for any tuple (i, j), Polly is not able to generate parallelized code for this example. For BAAR this means that instead of running the example on up to $N \cdot N = 10000$ threads on the Intel Xeon Phi, offloading fails and the calculation is executed on a single thread on the client's CPU.

A common workaround to simplify alias analysis is to define the variables in question as global variables, LLVM's Basic Alias Analysis Pass identifies global variables as not aliasing directly. However, neither is the frequent use of global variables good programming practice nor will this workaround produce correct results when used with BAAR in its current state. Another workaround is to just instruct Polly to ignore possible aliasing during optimization with the `-polly-ignore-aliasing` command line argument. This is not a general solution for BAAR, as it also ignores aliasing in cases where two pointers do in fact alias. In these cases, the calculations may produce incorrect results. However, the command line argument allows BAAR to correctly identify, score and offload the function shown in Listing 6.1. Unfortunately, this simple example seems to trigger a bug[1] [2] in the current version of Polly's OpenMP code generation. As interesting as the example in Listing 6.1 may be for evaluating the performance of BAAR, the bug currently prevents any parallelization attempt using Polly. Performance evaluation is performed on an example of stencil computation in Section 6.2, which works without

[1] `http://llvm.org/bugs/show_bug.cgi?id=17207`
[2] `http://llvm.org/bugs/show_bug.cgi?id=20010`

```
void seidel(double A[N][N], int elementsA) {
    for (int t = 0; t < STEPS; t++)
        for (int i = 1; i < N-1; i++)
            for (int j = 1; j < N-1; j++)
                A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                        + A[i][j-1] + A[i][j] + A[i][j+1]
                        + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
}
```

Listing 6.2: Example containing a SCoP with data dependencies

any workarounds.

In conclusion, BAAR relies on sophisticated alias analysis. As the very vivid LLVM project provides the alias analysis Passes for our project, BAAR will directly profit from any developments made in LLVM.

6.1.2. Data Dependencies

Parallelization as well as vectorization add new levels of parallelism and involve changes in the order of operations within the optimized loops. Therefore, these optimizations require the result of the calculation to remain the same when the order of operations changes. Listing 6.2 shows an example where this is not the case. E.g., for calculating `A[1][1]`, `A[1][0]` is needed which has to be calculated already. More generally, `A[i][j]` *depends* on `A[i][j-1]`, which defines a fixed order of operations. This kind of data dependency is called *read-after-write* and prevents the loop nest to be safely parallelized or vectorized.

When running the example shown in Listing 6.2 with BAAR, the `AccScore` Pass correctly decides to analyze this function more thoroughly as Polly's `SCoPDetection` Pass properly detects the SCoP. However at its current state, the `AccScore` Pass does not include the analysis of data dependencies in the function scoring process. Thus, the function is falsely treated as if it could be parallelized, gets a high score and is chosen to be offloaded. Succeeding function calls are executed remotely and as the code cannot be parallelized nor vectorized, the same single threaded code is run on the server. In our case, the server is running on an Intel Xeon Phi which is optimized for highly parallel code. Therefore, offloading this function causes considerable slowdowns.

To prevent BAAR to offload SCoPs which cannot be parallelized, the `AccScore` Pass has to be extended to include data dependency analysis. A possible way to implement this is to use Polly's `Dependences` Pass, which itself utilizes Isl's data dependency analysis. It is run on SCoPs and provides an integer indicating which kinds of data dependencies were detected. This information could easily be integrated in BAAR's `AccScore` Pass.

This section discussed limitations identified during evaluation which are important to resolve for making BAAR more generally usable. The following section deals with evaluating the performance of functions which were automatically offloaded and optimized by BAAR.

6.2. Performance

To evaluate and analyze execution time and possible speedups, *stencil codes* [20] are chosen to be run with BAAR. Stencil codes are a class of computations on arrays. Elements are updated by operating on the element itself and elements in its environment in a fixed pattern, the *stencil*. Stencil codes are run on arrays iteratively. Listing 6.2 shows an example of a stencil code with data dependencies. In contrast to this example, stencil codes often offer a great opportunity to exploit data parallelism and the number of operations per array element can nicely be tuned by changing the number of iterations and the size of the array. Therefore, stencil codes are promising candidates to achieve good performance with BAAR while allowing the analysis of the execution. As a representative for the stencil codes, the two dimensional Jacobi stencil [18] is chosen from the PolyBench/C[3] benchmark collection to conduct a detailed evaluation of BAAR.

The Jacobi method is a popular algorithm for solving Laplace's differential equation on a square domain, which is important for many fields of science, e.g., fluid dynamics or in the study of heat conduction. The main computation of the stencil code is exactly the same as in the PolyBench/C counterpart, the tunability parameters and memory allocations have been simplified to ensure SCoP detection by Polly and compatibility with BAAR. The function is shown in Listing 6.3, a program simply initializing an array to pass as the function argument and performing two calls to `jacobi_2d` is run with the BAAR Client. To ensure that both calls are offloaded, the Execution Engine executing the program is not run in parallel but after the server signals it is ready to accept calls. The whole procedure is run in several variations in respect to the size of the problem (N, STEPS) and optimizations performed on the server side. For all variations, the whole procedure of starting the program with the client, offloading the function, initializing the server and performing the calls is executed ten times. For any execution elapsed times are measured in μs, average values are rounded.

[3] `http://www.cs.ucla.edu/~pouchet/software/polybench/`

```
void jacobi_2d(double A[N][N], int elementsA) {
   double (*B)[N][N] = (double(*)[N][N])malloc(sizeof(double)*N*N);

   for (int t = 0; t < STEPS; t++) {
      for (int i = 1; i < N - 1; i++)
         for (int j = 1; j < N - 1; j++)
            (*B)[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] +
               A[1+i][j] + A[i-1][j]);
      for (int i = 1; i < N-1; i++)
         for (int j = 1; j < N-1; j++)
            A[i][j] = (*B)[i][j];
   }

    free((void*)B);
}
```

Listing 6.3: Two dimensional Jacobi stencil code used for performance evaluation

Elapsed times are measured by BAAR Client and Server using `std::chrono::high_resolution_clock`, the clock with the smallest tick period provided by the C++11 standard library [16]. The server sends measured times taken to execute client requests to the client. The client gathers all times measured by client or server and stores them in a file for easy post-processing. The measured times are the following:

- **Analysis**, the time taken by the client for estimating basic block frequencies, building the candidate set and scoring the functions
- **Acceleration Initialization**, this time is also measured on the client-side. It includes the time taken to declare `callAcc` in the program module, export the chosen functions to an LLVM IR Module, sending the Module to the server and waiting for the server to signal readiness to execute remote calls
- **Optimization**, the time taken by the server to optimize the LLVM IR code the client sent during the initialization process, measured on the server side
- **Backend Initialization**, this also happens during acceleration initialization on the server side. It is the time taken to initialize the chosen server backend. This measure is especially interesting when using the `ExtCompiler` server backend
- **Alteration**, the time taken to run the `RPCAccelerate` Pass on the offloaded functions and recompile them using the Execution Engine

- **Execution Accelerator**, measured time the native function call takes on the accelerator
- **Execution CallAcc**, time taken to execute a remote call on the client-side. This is essentially the time taken on the accelerator plus time taken for argument marshalling and unmarshalling as well as sending and receiving receiving the data

Additional to the time measuring, the BAAR Client is extended by command line arguments to disable parallel execution of the program to be accelerated and to run the program several times for automated evaluation. Not running the program to be accelerated in parallel, but only after acceleration, ensures that the program is readily accelerated when times are measured and avoids potential timing issues during automated measuring runs.

During evaluation, the client is run on the computer system hosting the Xeon Phi accelerator card. Potentially, it could be run on any system which is able to initiate a TCP/IP connection to the Xeon Phi. The host system features an Intel Xeon E5-2670 CPU with 16 cores at 2.6 GHz and 64 GB of RAM. The operating system is Scientific Linux 6.4 using the Intel Many Core Platform Stack in version 2.1.6720 to operate the Intel Xeon Phi. The server is run on the Xeon Phi itself, as explained in Section 3.4. The specific Intel Xeon Phi card used for evaluation is the Intel Xeon Phi 5110P with 60 cores at 1.053 GHz and 8 GB of RAM.

To be able to rate the quality of the execution times measured when running the program with BAAR, they are put in relation with execution times measured when running the program on the system's CPU hosting the Xeon Phi accelerator card. For this, the program is extended to measure the time taken to call the `jacobi_2d` function shown in Listing 6.3 using C++11's `std::chrono::high_resolution_clock`, the same system clock used to measure elapsed times with BAAR. The program is compiled using the Intel C++ Compiler in version 14.0.0 with GCC 4.8.0 compatibility on optimization level O2 (optimization for speed) and no additional compiler flags. The resulting highly optimized binary is executed on an Intel Xeon E5-2670, a high-end workstation processor and by far the most commonly used processor in systems listed in the TOP500 Supercomputer list [6] as of June 2014.

The performance of function execution and achievable speedups is evaluated in Section 6.2.2. This evaluation is conducted on the `jacobi_2d` example in detail, and additionally on a finite-difference time-domain stencil to show that BAAR is not limited to a single example.

6.2.1. Program Analysis And Acceleration

The times taken for program analysis are measured for the `jacobi_2d` function shown in Listing 6.3 for different values of N and STEPS, however only the measurements for N = 1000 and

STEPS = 10 are discussed in the following, as changing these constants in the program results in the same program analysis and optimization, as well as the same initialization of the backend on the server side.

Analysis

Procedure / Value	Analysis	Alteration
Average	2054 μs	11555 μs
Minimum	1990 μs	5985 μs
Maximum	2163 μs	12997 μs

Table 6.1.: Time taken for program Module analysis and alteration

Table 6.1 lists average, minimum and maximum values for the time taken for analyzing the program module and altering it on the client-side. The values are taken from a set of ten complete time measurements of the whole process of offloading and running the program with BAAR. Analyzing the program module includes identifying the `jacobi_2d` function as a candidate and scoring it as discussed in Chapter 4. Alteration includes running the `RPCAccelerate` Pass to transform the original program into the local part with `callAcc` calls and runtime decisions, as well as recompiling functions altered by `RPCAccelerate`. Analysis and alteration are executed by the client, which is running the original program in parallel. Therefore, the time spent in these processes should be as low as possible to avoid delaying its computations. Analysis took 2054 μs and alteration 11555 μs on average. To conclude the analysis of time the client has to invest to be able to call functions remotely, the initialization process of the acceleration still has to be taken into account.

Acceleration Initialization

Table 6.2 lists the time taken for initializing the acceleration after the program has been analyzed. It includes optimization and initializing the `ExtCompilerBackend` which both happens solely on the server side. The remaining time is spent in communication between client and server. On average, the total procedure of initializing the acceleration measured from client-side took 3347507 μs. Subtracting the average measured times for optimization and server backend initialization, this results in 12524 μs on average the client has to spend communicating with the server. Overall, the client has to invest roughly 27 ms on average to be able to call `jacobi_2d`

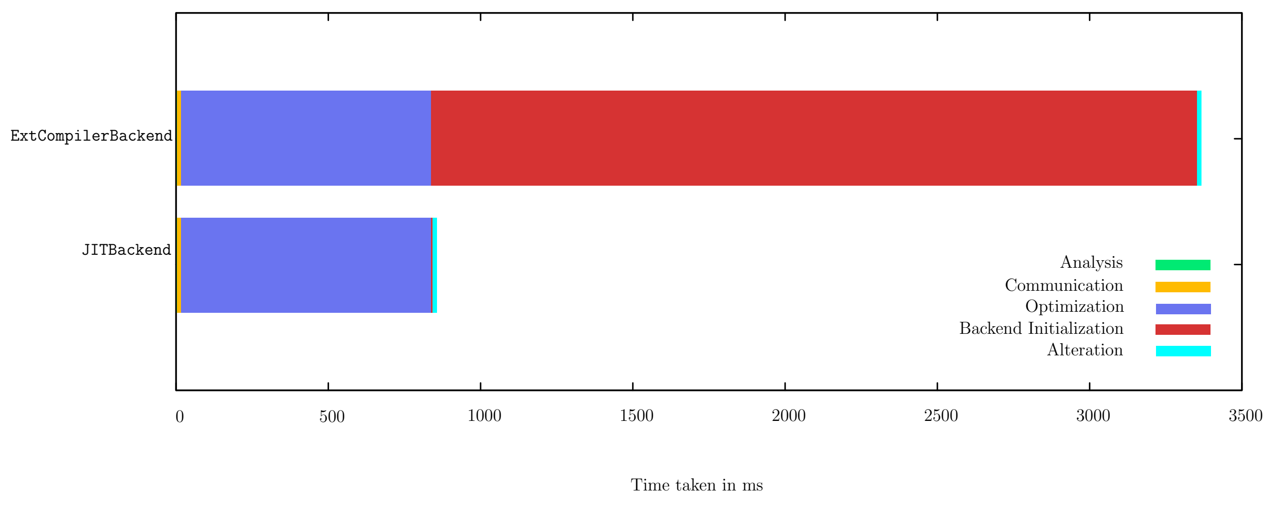

Figure 6.1.: Distribution of time taken for acceleration initialization

remotely on the server running on the Xeon Phi. In general, the aim is to accelerate heavy computations taking from seconds to several days. Furthermore, these processes are run only once by the client. Thus, requiring the client to invest some ten ms for a considerable speedup for all of the following calls to accelerated functions is reasonable.

Procedure / Value	Total	Optimization	Backend Init.
Average	3347507 μs	823365 μs	2511618 μs
Minimum	3103219 μs	818483 μs	2271782 μs
Maximum	3786403 μs	840576 μs	2949265 μs

Table 6.2.: Time taken initializing the acceleration with all optimizations

As Figure 6.1 illustrates, the majority of the time of initializing the acceleration on the server side is spend in initializing the `ExtCompilerBackend`, taking 2511618 μs on average and almost exactly 75% of the total time. This long period of time is spend in server backend initialization due to the unavailability of a native Xeon Phi LLVM backend. Therefore, getting a Xeon Phi binary for the offloaded LLVM IR code is a time consuming process as explained in Chapter 3. It requires transforming the LLVM IR code into C code and communicating with the host CPU to compile the C code into a native Xeon Phi binary. When the Xeon Phi LLVM backend becomes available, the time spend on initializing the server backend will drastically be improved without requiring changes in BAAR. Once this is the case, instead of using the `ExtCompilerBackend`, the `JITBackend` can be used. The `JITBackend` simply instantiates an Execution Engine. On an

Intel Core i7-3517U x86 laptop processor initializing the `JITBackend` with the exact same code used in the previous measures takes 3025 μs [4], 0.12% of the `ExtCompilerBackend` on the Xeon Phi, on average. Assuming this value for backend initialization in our setup, the time taken for optimization almost solely determines the time taken for initializing the acceleration as depicted in Figure 6.1. Once the native backend is available, it can be expected that the distribution of time taken for the acceleration initialization will look very similar to this.

Optimization

For analyzing the optimization phase of the acceleration initialization further, Table 6.3 lists the time taken for variations of the optimization phase to compare. ”None” means only the target-specific analysis passes are enabled and run on the offloaded IR code. ”Polly” adds preparation, optimization and cleanup Passes needed for Polly’s OpenMP code generation. ”Polly + Vectorization” adds vectorization to ”Polly”, i.e., full optimization. The measures show that with analysis Passes only, the optimization phase already takes 70.31% of the time full optimization takes on `jacobi_2d`. The measured times taken for the optimization phase includes initialization of the Pass Manager and target-specific Passes. Possibly, the optimization phase could be sped up when the initialization of the Pass Manager and target-specific Passes would directly be performed when the server is started. However, this could potentially limit the flexibility of BAAR in the future. The current concept allows a server to provide multiple accelerators, a client can choose a specific accelerator by specifying the respective target in the LLVM IR code sent to the server. Therefore, the target on the server side is determined by reading it from the LLVM IR code when initializing the acceleration. If a server with multiple accelerators is even a practical use case for BAAR and whether limiting the flexibility could improve the performance are interesting questions for future research.

Value \ Optimization	None	Polly	Polly + Vectorization
Average	578905 μs	785964 μs	823365 μs
Minimum	577123 μs	781985 μs	818483 μs
Maximum	585322 μs	793622 μs	840576 μs

Table 6.3.: Time taken for optimization

”Polly” already takes 95.46% of the time taken for full optimization on average. Polly transforms

[4] ten measurements, min: 3003 μs, max: 3064 μs

the LLVM IR into a polyhedral representation, utilizes isl and CLooG to perform polyhedral optimizations and transforms the result back into LLVM IR. Therefore, this optimization is more time consuming than LLVM Passes optimizing LLVM IR directly. When compared to Polly, the LLVM vectorization Passes are much more lightweight, adding only 37411 μs on average to the time taken for optimization with Polly only, resulting in 823365 μs in total for full optimization. Whether the time taken for optimization is acceptable depends on the use case. On the one hand the client is not impaired by the optimizations run on the server and a long time taken for optimization is reasonable for a great speedup of long running calculations. On the other hand however, the optimizations should finish quickly so that the client profits from the acceleration as soon as possible, certainly before the program finishes. Future research could explore the possibility to cache the resulting code when the client disconnects to have it readily available when it reconnects in another program run. This could be implemented by calculating a hash value from the LLVM IR code the client sends to the server and mapping the code resulting from optimization to this value. When a new client connects, the server would have to check whether an optimized code was already mapped to the hash value calculated from the LLVM IR code the client sends. This extension has the potential to effectively speed up the acceleration initialization for clients which want to accelerate code known to the server.

6.2.2. Function Execution

When analyzing the times taken for executing a `jacobi_2d` function call, the second call to the function is considered. The first call with the `ExtCompilerBackend` takes roughly 300 ms more than the second call[5]. The main reason is that the function has to be loaded from the shared object file on the hard drive into main memory on the first call. As the server already has the offloaded code in memory when using the `JITBackend` and native code is generated during acceleration initialization, the first and second call performances are still subject to caching but the calls perform more similarly.

The following section deals with the raw execution time of a `jacobi_2d` call, i.e., the time taken on the respective processing unit to execute the function call natively. Especially times taken for marshalling and unmarshalling of arguments are not considered. Analyzing the raw execution times gives insights on how well the optimizations improve the code and how `jacobi_2d` scales on the Xeon Phi. Afterwards, the total time taken for a remote call using `callAcc` is considered to see how much overhead BAAR adds, where it could be improved and if the mechanism can be used in practice at all.

[5]E.g., for STEPS=10 and N=1000 with full optimization the first call takes 360 ms on average over ten measures (min: 349 ms, max: 393 ms), the second 19 ms (min: 17 ms, max: 21 ms)

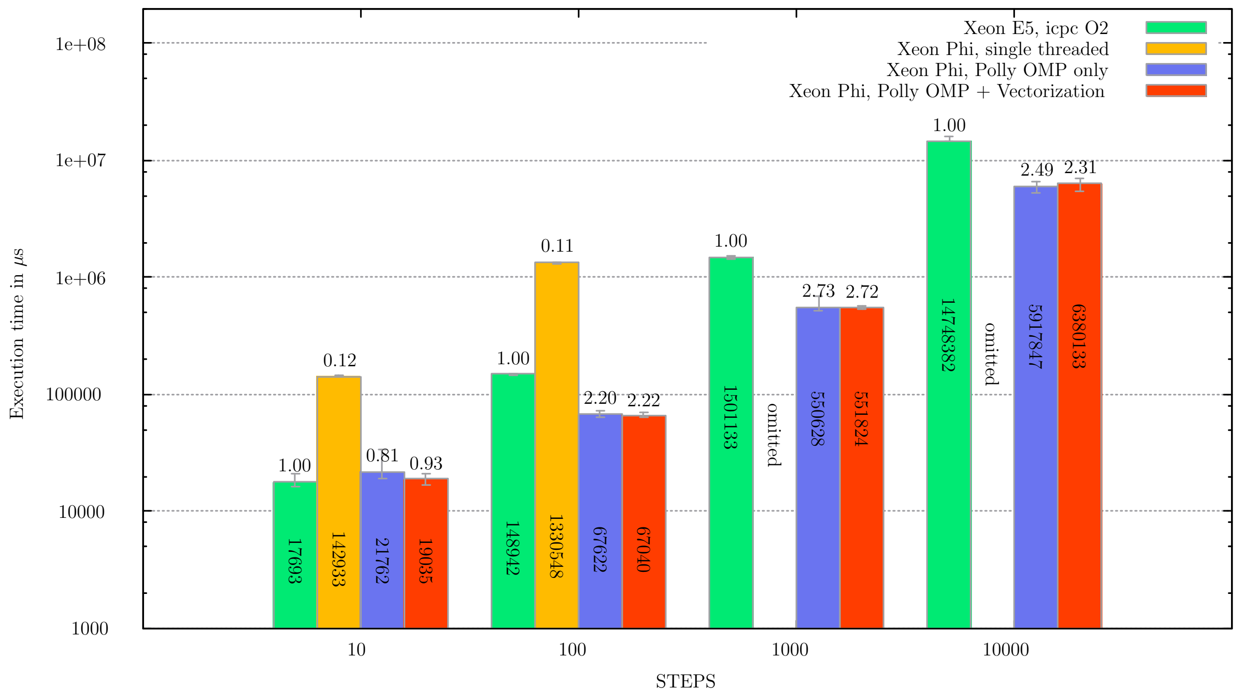

Figure 6.2.: Raw execution times for `jacobi_2d` with N=1000 (logarithmic scale)

Raw Execution Time

Figure 6.2 shows raw execution times for `jacobi_2d` with $N = 1000$, $\text{STEPS} \in \{10, 100, 1000, 10000\}$ on the Xeon E5 and Xeon Phi. As N is fixed, only the number of operations on the data changes, not the data size. With $\text{STEPS} = 10$, utilizing the Xeon Phi does not result in any speedup, even with full optimization `jacobi_2d` takes a few milliseconds longer on the Xeon Phi than the heavily optimized code on the Xeon E5. As the Xeon Phi's architecture is optimized for highly parallel code, it is no surprise that running unoptimized single-threaded code on the Xeon Phi results in a massive slowdown when compared to the optimized code on the Xeon E5. The Xeon Phi takes roughly 8 times longer, with $\text{STEPS} = 100$ almost 9 times. The fully optimized code however utilizes all the cores the Xeon Phi has by running 240 threads using OpenMP. Maintaining and synchronizing this many threads creates some overhead though. Therefore, with $\text{STEPS} = 10$ there are too few operations to exploit the advantage of the increased parallelism on the Xeon Phi. When increasing STEPS, the overhead for maintaining the threads becomes more and more negligible. With $\text{STEPS} = 100$, this already results in a speedup of 2.22 when comparing the fully optimized code on the Xeon Phi to the optimized code on the Xeon E5. Note that this is a comparison of heavily optimized code generated by the Intel Compilers running on the high end CPU Intel Xeon E5 with automatically parallelized and vectorized code by BAAR utilizing LLVM libraries and a detour over C code on the Intel Xeon Phi. With $\text{STEPS} = 1000$

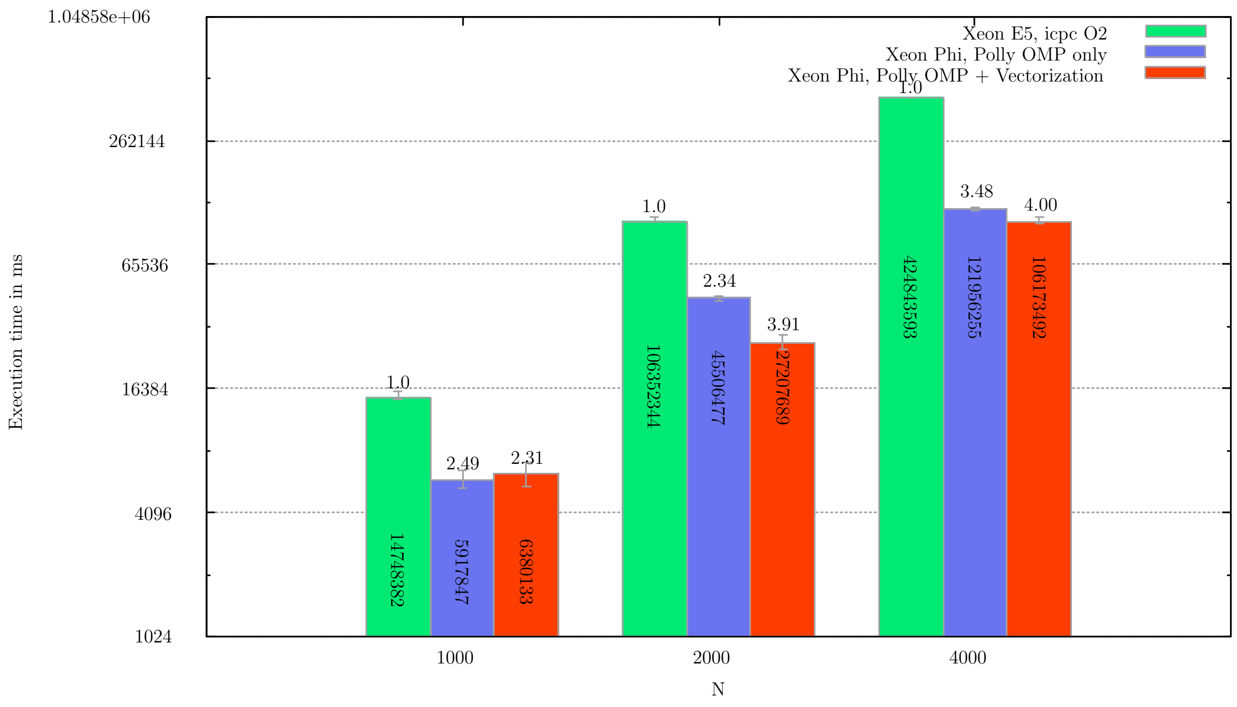

Figure 6.3.: Raw execution times for `jacobi_2d` with STEPS = 10000 (logarithmic scale)

and STEPS = 10000, the speedup ranges between 2.31 and 2.72 on average. The execution without vectorization is always on par with the fully optimized code, for N = 1000 this example does not profit from vectorization. Also, as the speedup of 2.72 was measured with STEPS = 1000, it seems we cannot achieve a higher speedup without altering N. Next, we will increase N to see how this example scales with an increased data size and to try to achieve an even higher speedup.

Figure 6.3 shows raw execution times for `jacobi_2d` with STEPS = 10000, $N \in \{1000, 2000, 4000\}$ on the Xeon E5 and Xeon Phi. With N = 2000, the speedup of the parallelized and vectorized code on the Xeon Phi over the heavily optimized code on the Xeon E5 is 3.91. It rises to 4 in these measurements when N is doubled to 4000. This means the raw execution time is over 5 minutes less on the Xeon Phi (1 min 46 s) compared to the Xeon E5 (7 min 5 s). The measurements also show that the vectorization profits from data sizes bigger than N = 1000. Setting N = 2000, the speedup with vectorization and parallelization compared to parallelization only is 1.67. However, with N = 4000 the speedup drops to 1.15. The exact reasons for this have to be clarified in future research. It can be an issue of data alignment, which was not taken in consideration at the current state of BAAR. The C backend taken from Intel's ispc supports several parameters. Potentially, the parameters chosen for BAAR are not optimal.

Overall, the speedup of up to 4 without any hints, when comparing parallelized and vectorized execution of this example on the Xeon Phi to heavily optimized code on the Xeon E5 is a promising result for the practicality of BAAR. Especially when considering that the Xeon E5 is a high end processor, which potential clients may not have available. However, so far only the raw execution time was considered. To evaluate BAAR, the overhead imposed by marshalling and unmarshalling the call during `callAcc` has to be taken into account.

`callAcc`

Table 6.4 is based on the same set of measurements as Figure 6.3 but considers the full `callAcc` call for calling `jacobi_2d` remotely instead of just the raw execution time. It lists the average, minimum and maximum time taken for the `callAcc` call, the speedup compared to heavily optimized execution on the Xeon E5, as well as the amount of time spent on the actual function call on the Xeon Phi (raw execution time). The data shows that from the previously calculated speedup of 4, when just comparing raw execution times with N = 4000 and STEPS = 10000, only a speedup of 2.17 remains when considering the full `callAcc` call. When increasing STEPS for a fixed N, the share of the communication on `callAcc` decreases so that the speedup of the `callAcc` call is closer to the speedup of the raw execution time. It is evident however, that the simple communication mechanism currently implemented in BAAR heavily impairs the execution of remote calls.

Value \ N	1000	2000	4000
Average	12406 ms	49945 ms	195859 ms
Minimum	11562 ms	47984 ms	193814 ms
Maximum	13165 ms	52912 ms	201101 ms
Speedup	1.19	2.13	2.17
Spent in raw execution (avg.)	51.43 % (6380 ms)	54.48 % (27208 ms)	54.21 % (106173 ms)

Table 6.4.: Time taken for `callAcc` calling `jacobi_2d` with STEPS = 10000 and full optimization

Almost 46 % of the time is spent in communicating with the client, which mostly consists of marshalling and unmarshalling the arguments. As the task manager shows, during `callAcc` when not executing the function call, a single core on either side is working at full capacity. This is due to the simple design of the communication mechanism over sockets (`SocketClient` and SocketServer, respectively). When marshalling or unhmarshalling an array, every single element is sequentially transformed into its string representation and concatenated to a string representing the whole array. The resulting string is concatenated to a string which eventually contains all arguments in their respective string representations. For BAAR in its current prototype form, it had the advantages of being straight forward to implement and being transparent to the endianness of the accelerator. However, it is currently the limiting factor for achieving better speedups. It leaves much room for improvement, e.g., the elements of an array could perfectly be marshalled in parallel, marshalling of arguments on the client-side could be interleaved with unmarshalling on the server side and vice versa. Using sockets as a means of communication is appealing because they are universally usable. For BAAR however, more suitable alternatives exist. A promising alternative for future research is the *Message Passing Interface* (MPI), a standard for exchanging messages in distributed systems. It supports much better performing mechanisms to transfer values than transforming them into a string representation. Furthermore, it transparently supports shared memory systems. This could additionally make it an alternative to the shared memory communication mechanism implemented in ShmemClient and ShmemServer, which is not evaluated in this thesis.

Finite-Difference Time-Domain

Additional to the Jacobi stencil, function execution is evaluated with the finite-difference time-domain (FDTD) stencil [25]. FDTD is a widely used modeling technique in computational electrodynamics. This stencil code is also part of PolyBench/C, a simplified version without tunabililty parameters is shown in Listing 6.4.

While jacobi_2d was accelerated by BAAR without any hints, BAAR Client and Server have to be started with the --polly-ignore-aliasing command line argument to enable Polly to detect the SCoPs in this code. Otherwise, parallelization fails because the alias analysis cannot prove that there is no aliasing (see Section 6.1.1).

Figure 6.4 shows average execution times for fdtd_2d with STEPS = 10000 and N = 2000 over ten runs and achieved speedups compared to execution of code compiled with the Intel C++ Compiler at O2 on an Intel Xeon E5-2670. On the Xeon E5, the execution took 217337 ms on average[6]. Considering only the raw execution times taken by executing the code with BAAR, the speedups are 2.36 with parallelization only as well as 5.77 with parallelization and optimization. When comparing these results to the measurements taken for the jacobi_2d example, the speedup with parallelization only is very similar. Enabling vectorization in addition

[6]min: 216034 ms, max: 217582 ms

```
void fdtd_2d(double ex[N][N], int elements_ex, double ey[N][N], int
    elements_ey, double hz[N][N], int elements_hz, double _fict_[STEPS],
    int elements__fict_) {
    for(int t = 0; t < STEPS; t++) {
        for (int j = 0; j < N; j++)
            ey[0][j] = _fict_[t];
        for (int i = 1; i < N; i++)
            for (int j = 0; j < N; j++)
                ey[i][j] = ey[i][j] - 0.5*(hz[i][j]-hz[i-1][j]);
        for (int i = 0; i < N; i++)
            for (int j = 1; j < N; j++)
                ex[i][j] = ex[i][j] - 0.5*(hz[i][j]-hz[i][j-1]);
        for (int i = 0; i < N - 1; i++)
            for (int j = 0; j < N - 1; j++)
                hz[i][j] = hz[i][j] - 0.7 *  (ex[i][j+1] - ex[i][j] +
                    ey[i+1][j] - ey[i][j]);
    }
}
```

Listing 6.4: FDTD stencil code for a square matrix

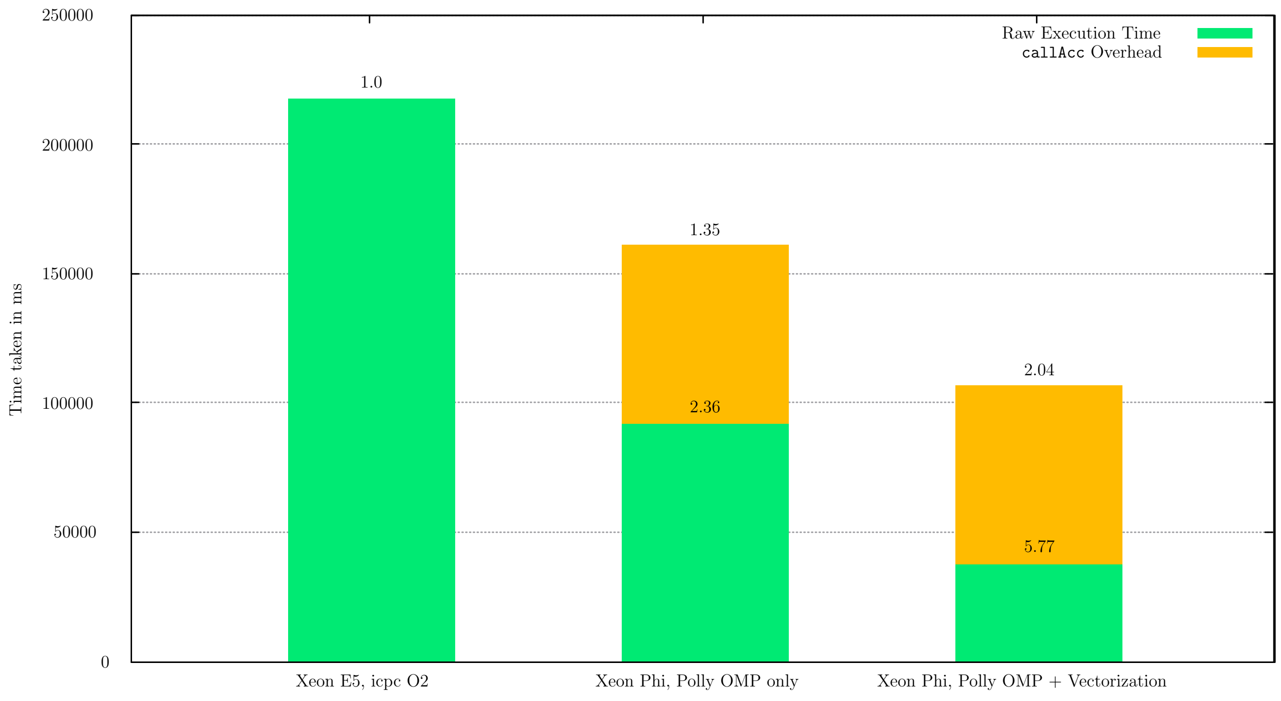

Figure 6.4.: Execution times for `fdtd_2d` with STEPS = 10000, N = 2000

to parallelization gives the execution on BAAR another speedup of 2.44, resulting in a speedup of 5.77 in total. With the `jacobi_2d` example, the highest speedup achieved was 4, the effect of vectorization was much smaller. This stresses that future research should try to improve the parameters used for vectorization in BAAR and investigate in requirements for more reliable results.

When considering the full time taken for executing `callAcc` on `fdtd_2d`, BAAR takes 160742 ms[7] on average with parallelization only and 106564 ms[8] with parallelization and vectorization. From the previously achieved speedups of 2.36 and 5.77 only 1.35 and 2.04 remain, respectively. This underlines that the simple communication mechanism impairs the execution and should urgently be improved.

This section concludes the evaluation of BAAR's performance. The following section evaluates the quality of the runtime decision whether to offload a certain function call or not.

6.3. Runtime Decision

Section 6.2.2 discussed the performance of BAAR based on the `jacobi_2d` function shown in Listing 6.3, as well as the `fdtd_2d` function shown in Listing 6.4. The evaluation showed that marshalling and unmarshalling of arguments is currently the major limiting factor for achieving higher speedups by inappropriately utilizing one CPU core at full capacity. However, even with a better performing communication mechanism in place, the overhead imposed by `callAcc` will always be system dependent. Therefore, the runtime decision whether to offload a certain call or to run it locally, can be influenced by command line parameters. As discussed in Section 4.3, the runtime decision is equivalent to:

$$\frac{\text{function score}}{\text{bytes to transfer}} > c, \text{ where } c \text{ is set by a command line argument (0 if unset)}$$

This section aims to gain insights on whether the runtime decision in its current form is well designed, how to choose the command line parameter c as well as how the decision can be improved.

Table 6.5 lists calculated values for $\frac{\text{function score}}{\text{bytes to transfer}}$ when running `jacobi_2d` with BAAR for several values of N and STEPS. The weights for floating point and integer operations is unset and therefore defaults to one. Thus, the function score approximates the total number of operations executed when calling `jacobi_2d`. Consequently, when increasing STEPS for a fixed

[7]min: 159758 ms, max: 162014 ms

[8]min: 105639 ms, max: 107707 ms

N by multiplying it by ten, $\frac{\text{function score}}{\text{bytes to transfer}}$ is also multiplied by ten. Altering STEPS does not change the bytes transferred, but only the number of operations. For a fixed STEPS, the number of operations is directly proportional to N^2, as is the size of the array. As the array and its number of elements are the only two arguments to `jacobi_2d`, the bytes to transfer is mostly determined by the size of the array. Therefore, when multiplying N by two for a fixed STEPS, $\frac{\text{function score}}{\text{bytes to transfer}}$ should only marginally change. This is correctly mirrored by the values calculated by BAAR shown in Table 6.5.

STEPS \ N	1000	2000	4000
10	29.88011	29.94002	29.97001
100	298.8011	299.4002	299.7001
1000	2988.011	2994.002	2997.001
10000	29880.11	29940.02	29970.01

Table 6.5.: $\frac{\text{function score}}{\text{bytes to transfer}}$ for `jacobi_2d` and several values of N and STEPS

To be able to choose a reasonable value for c, the values of $\frac{\text{function score}}{\text{bytes to transfer}}$ have to be matched with the actual speedups achieved by BAAR compared to the execution on the Intel Xeon E5. Table 6.6 lists the speedups. As for STEPS $\in \{10, 100, 1000\}$ and N $\in \{1000, 2000, 4000\}$ the speedups are less than one because a huge amount of time is spent on marshalling and unmarshalling, c has to be greater than 2997 ($\frac{\text{function score}}{\text{bytes to transfer}}$ for N = 4000, STEPS = 1000). For STEPS = 10000 and N $\in \{1000, 2000, 4000\}$, the speedups are all greater than one. Thus, it is reasonable to choose $2997 < c \leq 29880$ for this example and setup.

STEPS \ N	1000	2000	4000
10	< 0.001	0.005	0.005
100	0.024	0.046	0.047
1000	0.227	0.420	0.424
10000	1.189	2.125	2.173

Table 6.6.: Average speedup for `jacobi_2d` and several values of N and STEPS when comparing offloaded to local execution

To obtain a more generally applicable or a more precise value for c, more measurements with examples other than `jacobi_2d` would have to be performed. The result would still be specific to the simple socket communication mechanism, client CPU and accelerator. Thus, future research is needed to introduce a mechanism to automatically determine a fitting value for c for a specific setup. A first approach could be to have two small benchmarks. One for evaluating the communication performance by simply sending a small array there and back between server and client, marshalling and unmarshalling it as needed. The second benchmark would run a simple kernel on the client CPU and in optimized form on the accelerator. With the function score and bytes to transfer of these benchmarks, a system specific value for c could automatically be determined. Additionally, it would be interesting to introduce self-adaptation. As a first step a simple approach could be used to gain valuable insights. E.g., when offloading a function `f` with $\frac{\text{function score}}{\text{bytes to transfer}} = c_1 > c$, `f` could be run on the client CPU and accelerator in parallel. Should the accelerator return the result before the call on the client CPU finishes, cancel it and proceed steadily. Should the client CPU finish the calculation first however, set $c = c_1$ to require future calls to have a higher $\frac{\text{function score}}{\text{bytes to transfer}}$ value.

7. Conclusion And Future Directions

This thesis introduced BAAR, our approach of tackling the problem of enabling existent software to automatically utilize accelerators. Important extensions to the mechanism were made to achieve a proof-of-concept implementation. BAAR is capable of automatically detecting functions suitable for acceleration, offloading them as well as automatically parallelizing and vectorizing them. Whether to utilize the Intel Xeon Phi is decided at runtime per function call. This whole process is transparent to the user. The evaluation could show BAAR's practicality by achieving a speedup of up to 4 without any hints, when comparing execution of a real-life example compiled with the Intel C++ Compiler at O2 on an Intel Xeon E5-2670 to execution using BAAR utilizing an Intel Xeon Phi 5110P accelerator card. With hints, even a speedup of 5.77 could be achieved. This points out the performance possible when alias analysis is improved.

In its proof-of-concept state, BAAR can already achieve great results for idempotent functions containing SCoPs with a high compute intensity, as was shown on the basis of stencil codes. The evaluation has shown that BAAR has a high potential to be more generally applicable and achieve better performances.

To make BAAR more widely applicable, several problems require further research:

- It would be very interesting to evaluate the performance of BAAR with different combinations of client system and server system. E.g., the BAAR Client is easily portable to the ARM architecture with only minor alterations in the build scripts as the ARM architecture is already supported by LLVM. This leads to an interesting setup consisting of several BAAR Clients running on low-power ARM-based systems sharing an Intel Xeon Phi accelerator for their calculations provided by the BAAR Server. The server already supports multiple clients when communicating over sockets, so the current state of BAAR already supports this setup
- The server should be extended to provide several targets to the clients, not only one target throughout its whole lifetime. This would lead to many attractive directions for further research. It would increase the potential for increased performance to have several targets with different characteristics the client can choose from. Additionally, it would be

interesting to investigate in criteria other than performance. Maybe the BAAR Server can provide a target to the client which can perform a certain calculation much more power efficient, during calculation the client can standby and save power. It is also thinkable to switch the target when requirements change

- So far, the functions supported by BAAR have to be idempotent. Global variables are only copied once to the server upon acceleration initialization and pointers are always interpreted as arrays, a mechanism to keep global variables in sync between client and server as well as an extension to support arbitrary values pointed to are needed. Additionally, system calls are not supported, yet. This problem could be tackled by two approaches. Firstly, system calls could be wrapped into a call which sends the system calls to the client and the response back. Secondly and maybe more interestingly, the granularity at which program parts are offloaded could become finer. At this point, functions are offloaded, but it is thinkable to offload at the granularity of SCoPs. These are less likely to contain system calls and additionally this approach could lead to a performance increase because parts of the function which cannot profit from execution on the accelerator are executed on the client system. However, it has to be further researched whether this approach leads to an increased amount of data which has to be transferred between client and server
- The constant c which is compared to $\frac{\text{function score}}{\text{bytes to transfer}}$ of a certain function call to make the decision whether it should be run remotely or locally, is currently set by the user at server start or defaults to zero. Section 6.3 already discussed how c can automatically be determined. This would increase the usability of BAAR. Additionally, making c variable and self-adapting has been discussed, which can potentially increase the performance of BAAR and should therefore be considered for future research
- Currently, only the size of the arguments of a certain call are used as runtime information. Especially alias analysis could profit from utilizing runtime information and increase the applicability of BAAR

Several future directions of research could improve the performance of BAAR drastically:

- The evaluation has shown that the communication mechanism is currently the weak point of BAAR when trying to achieve great speedups. Section 6.2.2 already discussed that a communication mechanism between client and server which utilizes MPI could drastically improve the performance of an offloaded call. Especially argument marshalling would profit from a more sophisticated mechanism than currently implemented
- When having arrays as arguments, they are always transferred to and from the server as a whole for every call. Ideally, a more intelligent mechanism would only copy elements to

the server which are used by the code executed on it and copy back elements which were altered. Additionally, it could make sense to cache arrays for subsequent calls

- At this point, a function which is called before the acceleration is initialized is executed locally by the client, even when the initialization finishes while the call is still executing. This situation could be improved by introducing *on stack replacement*. With on stack replacement, the running function call would be suspended once the acceleration finishes, the remote execution of this call for the current state of execution would be setup and the call would be continued by running it remotely on the accelerator. Especially long running function calls could greatly profit from this extension to BAAR
- On a more technical level, BAAR would greatly profit from a native LLVM backend for the Intel Xeon Phi. It would enable us to drop the detour over C code and improve the quality of vectorization. Such a backend is scheduled to arrive with Intel's next generation Xeon Phi products code named "Knights Landing" in the second half of 2015[1]. BAAR would directly profit from a native Xeon Phi backend by enabling the already implemented `JITBackend` to be used

In sum, this thesis successfully introduced and implemented a new approach for easy-to-use on-the-fly binary program acceleration on many-cores. Valuable insights were gained during evaluation and numerous future directions of research were discussed.

[1] `http://lists.cs.uiuc.edu/pipermail/llvmdev/2013-July/063697.html`

Appendix A. Contents Of The Attached Data Medium

The attached data medium includes the complete commented source code of BAAR in its states before and after the extensions of this thesis were implemented. Every example used during evaluation and time measurements made are also incorporated. Furthermore, snapshots of all referenced websites are enclosed. In addition, a digital copy of this thesis is included.

Appendix B. Building The BAAR Environment

This chapter thoroughly explains how to build BAAR with LLVM and Polly for an x86 system and how to cross-build it for the Intel Xeon Phi. This process may seem tedious, but please note that most of the dependencies are needed for Polly which provides the automatic parallelization for BAAR. If you do not need automatic parallelization, consider to remove Polly as a dependency. This would greatly simplify the build process, as only LLVM and BAAR itself need to be build then, which are mostly self-contained.

B.1. Building LLVM Including Polly With CMake

This section briefly describes the steps needed to build LLVM with Polly as a requirement for BAAR. Details on the LLVM-specific CMake variables and more information can be found in the LLVM documentation [2]. Note that a host build of LLVM has to be available for the host cross-building LLVM for the Intel Xeon Phi.

B.1.1. Host Build

Building LLVM with Polly on an x86 machine is straight forward. Create a directory for the source code, it will be called `$LLVM_SRC` in the following. Afterwards, run the following commands to obtain the correct version of the source code:

```
$ git clone http://llvm.org/git/llvm.git -b release_34 $LLVM_SRC
$ git clone http://llvm.org/git/polly.git -b release_34
    $LLVM_SRC/tools/polly
```

To build Polly, libgmp, CLooG and isl are required. libgmp should be available through the operating system package management system or already installed. Polly requires a fixed version of CLooG and isl, which can be obtained by a script supplied with it. To obtain the source and build it, create two directories outside the `$LLVM_SRC` directory. These will be called `$CLOOG_SRC` and `$CLOOG_X86`. Afterwards, run the following commands:

```
$ $LLVM_SRC/tools/polly/utils/checkout_cloog.sh $CLOOG_SRC
$ cd $CLOOG_SRC
$ ./configure --prefix=$CLOOG_X86
$ make && make install
$ cp -r $CLOOG_SRC/include/cloog $CLOOG_X86/include
$ cp -r $CLOOG_SRC/isl/include/isl $CLOOG_X86/isl/include
```

If you get an error mentioning undefined `ACLOCAL_PATH` or `AC_PROG_LIBTOOL` variables, it means your system does not have libtool installed. Either install it with your operating system package manager or compile it yourself and rerun the script. If you still get these errors, run the following commands with `$LIBTOOL_INSTALL_DIR` pointing to your libtool installation:

```
$ export $LIBTOOL_INSTALL_DIR/share/aclocal
$ cd $CLOOG_SRC
$ libtoolize
$ cd isl
$ libtoolize
$ cd ..
$ ./autogen
```

Afterwards, rerun the commands listed previously.

When the build completes, the requirements to build LLVM with Polly are met. Create another directory, outside of any other directory created so far, which we will call `$LLVM_X86`. Afterwards, run the following commands for building:

```
$ cd $LLVM_X86
$ cmake -DCLOOG_INCLUDE_DIR=$CLOOG_X86/include
    -DCLOOG_LIBRARY=$CLOOG_X86/libcloog-isl.la
    -DISL_INCLUDE_DIR=$CLOOG_X86/isl/include
    -DISL_LIBRARY=$CLOOG_X86/isl/libisl.la $LLVM_SRC
$ make -j4
```

Once these commands finish, the main LLVM tools and libraries as well as Polly for the host are available in `$LLVM_X86`.

B.1.2. Cross-Building LLVM Including Polly For The Intel Xeon Phi

Cross-building LLVM with Polly for the Intel Xeon Phi roughly follows the same procedure as described for x86 based systems in Section B.1.1. However, libgmp may not be available on the Xeon Phi and therefore also has to be cross-built in this process. Furthermore, some additional preparations have to be taken care of to enable cross-building Polly's other requirements.

Setup Environment For Cross-Building

Before compiling anything, the environment for cross-building has to be set up. At first, the compiler has to be set to the Intel Compilers and the correct flags have to be set with the following commands:

```
export CC=icc
export CXX=icpc
export CFLAGS="-mmic"
export CXXFLAGS=$CFLAGS
```

Additionally, CMake needs a text file which gives information about the compiler toolchain for cross-compilation. Create a text file, it will be called `$TOOLCHAIN_FILE` in the following. The file has to include the following lines:

```
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR k1om)
SET(CMAKE_SYSTEM_VERSION 1)

SET(CMAKE_C_COMPILER   icc)
SET(CMAKE_CXX_COMPILER icpc)
SET(_CMAKE_TOOLCHAIN_PREFIX  x86_64-k1om-linux-)

SET(CMAKE_FIND_ROOT_PATH /usr/linux-k1om-4.7)
```

After these steps, the environment for cross-compiling for the Xeon Phi is ready.

libgmp

libgmp is a requirement for CLooG, which itself is needed for Polly. It may already be available on the Intel Xeon Phi, if it is not it has to be cross-built as well. For this, create a build directory for it, which will be called `$GMP_MIC` in the following, and perform the following commands to obtain the libgmp source code and cross-build it:

```
$ cd $GMP_MIC
$ wget https://gmplib.org/download/gmp/gmp-6.0.0a.tar.bz2
$ tar xfj gmp-6.0.0a.tar.bz2
$ ./gmp-6.0.0/configure --host=x86_64-k1om-linux --disable-assembly
   --prefix=$GMP_MIC
$ make -j4 && make install
```

CLooG and isl

Assuming the CLooG source is available in `$CLOOG_SRC`, create a directory for the Xeon Phi build outside of it, it will be called `$CLOOG_MIC` in the following. Execute the following commands to cross-build CLooG:

```
$ cd $CLOOG_MIC
$ $CLOOG_SRC/configure --prefix=$CLOOG_MIC --host=x86_64-k1om-linux
   --with-gmp-builddir=$GMP_MIC
$ make -j4 && make install -i
```

Again, if you get an error mentioning undefined `ACLOCAL_PATH` or `AC_PROG_LIBTOOL` variables, it means your system does not have libtool installed, see Section B.1.1 for more information.

LLVM with Polly

To build LLVM with Polly, the previous steps (including Section B.1.1) all have to have completed successfully. If this is the case, create a directory outside any other directory previously create, it will be called `$LLVM_MIC` in the following. Afterwards, perform the following commands:

```
$ sed -i'' '28,32 s/^/\/\//' $LLVM_SRC/lib/Target/X86/X86JITInfo.cpp
$ cd $LLVM_MIC
$ cmake -DCMAKE_TOOLCHAIN_FILE=$TOOLCHAIN_FILE
    -DCMAKE_CROSSCOMPILING=True -DTARGET_TRIPLE=k1om-unknown-linux-gnu
    -DLLVM_DEFAULT_TARGET_TRIPLE=k1om-unknown-linux-gnu
    -DLLVM_TARGETS_TO_BUILD="X86"
    -DLLVM_TABLEGEN=$LLVM_X86/bin/llvm-tblgen
    -DGMP_LIBRARY=$GMP_MIC/lib/libgmp.so
    -DGMP_INCLUDE_DIR=$GMP_MIC/include
    -DCLOOG_INCLUDE_DIR=$CLOOG_MIC/include
    -DCLOOG_LIBRARY=$CLOOG_MIC/lib/libcloog-isl.so
    -DISL_INCLUDE_DIR=$CLOOG_MIC/include
    -DISL_LIBRARY=$CLOOG_MIC/lib/libisl.so $LLVM_SRC
$ make -j4
$ sed -i'' '28,32 s/^\/\/*//' lib/Target/X86/X86JITInfo.cpp
```

The X86 target is utilized to get information required for vectorization (see Section 5.3). Note that the LLVM X86 target does not compile for the Intel Xeon Phi with JIT enabled, because JIT uses assembler instructions not supported by the Intel Xeon Phi. The LLVM build scripts do not recognize the Intel Xeon Phi and default to X86 as target platform, which enables JIT. Therefore, sed is used to comment the lines enabling JIT before the cross-build and uncomment them after the build finished.

Once these commands finish, the main LLVM tools and libraries as well as Polly for the Intel Xeon Phi are available in `$LLVM_MIC`.

B.2. Building BAAR

B.2.1. Host Build

When LLVM and Polly are built as a requirement for BAAR, the mechanism itself can be built. How to build the requirements and variables defined in this process are described in Section B.1.1. For the next steps create two directories for the source code and the binaries, these will be called `$BAAR_SRC` and `$BAAR_X86` in the following. Unpack the BAAR source code from the data medium to `$BAAR_SRC`. Afterwards, run the following commands to build it:

```
$ ln -s $LLVM_SRC/tools/polly/include/polly/ $LLVM_SRC/include/.
$ ln -s $LLVM_X86/tools/polly/include/polly/ $LLVM_X86/include/.
$ ln -s $LLVM_X86/lib/LLVMPolly.so $LLVM_X86/lib/libLLVMPolly.so
$ cd $BAAR_X86
$ cmake -DLLVM_SRC_DIR=$LLVM_SRC -DLLVM_BIN_DIR=$LLVM_X86 $BAAR_SRC
$ make
```

The first three commands are used to create symlinks to ensure CMake finds Polly's include files and library. Once the build finishes, the BAAR client and server are available in `$BAAR_X86/out/bin/` as x86 binaries. If the BAAR Server fails to build stating that `ffi.h` could not be found, install libffi with your operating system's package manager. The client does not require libffi to build successfully, so you can ignore this error if the server runs on another target in your use case.

B.2.2. Cross-Building BAAR For The Intel Xeon Phi

Cross-building the offload mechanism is very similar to building it for the host. Assuming all the previous steps were completed successfully, just create a directory outside any other previously created directory. The directory will be the output directory for the build and called `$BAAR_MIC` in the following. Then run the following commands:

```
$ ln -s $LLVM_MIC/tools/polly/include/polly/ $LLVM_MIC/include/.
$ ln -s $LLVM_MIC/lib/LLVMPolly.so $LLVM_MIC/lib/libLLVMPolly.so
$ cd $BAAR_MIC
$ cmake -DCMAKE_TOOLCHAIN_FILE=$TOOLCHAIN_FILE
   -DCMAKE_CROSSCOMPILING=True -DLLVM_SRC_DIR=$LLVM_SRC
   -DLLVM_BIN_DIR=$LLVM_MIC $BAAR_SRC
$ make
```

Once these steps finish successfully, the whole environment is completely built. The following chapter explains how BAAR is started to be able to automatically offload code to an Intel Xeon Phi.

Appendix C. Starting The BAAR Environment

This chapter is specific to the Paderborn Center for Parallel Computing's (PC^2) HPC cluster OCuLUS[1] and explains how to start BAAR to be able to reproduce the evaluation of Chapter 6.

First, copy `baar_server` from `$BAAR_MIC/out/bin/`, `baar_client` from `$BAAR_X86/out/bin/` and `knc-i1x8.h` from `$BAAR_SRC/server/pass/` into your home directory.

Now we can start the BAAR Server on the Intel Xeon Phi. The `ExtCompilerBackend` used to generate Xeon Phi binaries from LLVM IR utilizes the compiler of the system hosting the Intel Xeon Phi over a secure shell connection. Therefore, we need to allocate the node hosting the Xeon Phi accelerator card interactively and additionally the Xeon Phi accelerator card itself. Afterwards, we establish a secure shell connection from the host node to the Xeon Phi and start the server on it. This done by the following commands:

```
ccsalloc --res=rset=arch=MIC:hostname=phi001-mic0+hostname=phi001 -I
ssh phi001-mic0
./baar_server -backend=extcompiler -force-vector-width=8
   -enable-polly-openmp
```

In another terminal window, we will compile a test program and start the client with it on the previously allocated node. It should connect to the server running on the Xeon Phi using sockets:

[1]`http://pc2.uni-paderborn.de/`

```
ssh phi001
module add intel/compiler gcc/4.8.1 cmake/2.8.10.2
clang -S -emit-llvm test.c -o test.ll
./baar_client -host=phi001-mic0 ./test.ll
```

The `-help` command line argument lists the commands available for BAAR Client and Server. `-polly-ignore-aliasing` is a hidden command line argument for Polly, it instructs Polly to parallelize code in cases where it could not be proven that pointers do not overlap. Starting BAAR Client and Server with this argument extends the possible test cases.

Bibliography

[1] Auto-Vectorization in LLVM. Website: `http://llvm.org/docs/Vectorizers.html`. [Online; accessed 26-August-2014].

[2] Building LLVM with CMake. Website: `http://llvm.org/docs/CMake.html`. [Online; accessed 26-August-2014].

[3] LLVM 3.1 Release Notes. Website: `http://llvm.org/releases/3.1/docs/ReleaseNotes.html#whatsnew`. [Online; accessed 17-August-2014].

[4] LLVM Execution Engine Class Reference. Website: `http://llvm.org/docs/doxygen/html/classllvm_1_1ExecutionEngine.html`. [Online; accessed 26-August-2014].

[5] LLVM's Analysis and Transform Passes. Website: `http://llvm.org/docs/Passes.html`. [Online; accessed 26-August-2014].

[6] The TOP500 Supercomputer. Website: `http://www.top500.org/`. [Online; accessed 16-August-2014].

[7] Writing an LLVM Pass. Website: `http://llvm.org/docs/WritingAnLLVMPass.html`. [Online; accessed 26-August-2014].

[8] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[9] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM.

[10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications.* lulu.com, June 2011.

[12] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *In IEEE PACT*, pages 106–111. IEEE Computer Society Press, 1998.

[13] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, December 2009. Order Number 253669-033US.

[15] J. Jeffers, J.R. Jeffers, and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Elsevier Science & Technology Books, 2013.

[16] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley Professional, 2nd edition, 2012.

[17] Chris Lattner and Vikram Adve. 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, 2004. IEEE Computer Society.

[18] B.P. Lester. *The Art of Parallel Programming.* Prentice Hall, 1993.

[19] Dmitry Mikushin, Nikolay Likhogrud, Eddy Zheng Zhang, and Christopher Bergström. KernelGen the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs. Technical Report 2013/02, University of Lugano, July 2013.

[20] David A. Padua. *Encyclopedia of Parallel Computing.* Springer, 2011.

[21] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[22] Matt Pharr and William R Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[23] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[24] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. Sambamba: A runtime system for online adaptive parallelization. In *Compiler Construction*, pages 240–243. Springer, 2012.

[25] Dennis M. Sullivan. *Electromagnetic Simulation Using the FDTD Method.* Wiley-IEEE Press, 2013.