Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi

Marvin Damschen*
Karlsruhe Institute of Technology
marvin.damschen@kit.edu

Heinrich Riebler, Gavin Vaz and Christian Plessl
University of Paderborn
{heinrich.riebler | gavin.vaz | christian.plessl}@uni-paderborn.de

***Abstract*—In this paper, we study how binary applications can be transparently accelerated with novel heterogeneous computing resources without requiring any manual porting or developer-provided hints. Our work is based on Binary Acceleration At Runtime (BAAR), our previously introduced binary acceleration mechanism that uses the LLVM Compiler Infrastructure. BAAR is designed as a client-server architecture. The client runs the program to be accelerated in an environment, which allows program analysis and profiling and identifies and extracts suitable program parts to be offloaded. The server compiles and optimizes these offloaded program parts for the accelerator and offers access to these functions to the client with a remote procedure call (RPC) interface. Our previous work proved the feasibility of our approach, but also showed that communication time and overheads limit the granularity of functions that can be meaningfully offloaded. In this work, we motivate the importance of a lightweight, high-performance communication between server and client and present a communication mechanism based on the Message Passing Interface (MPI). We evaluate our approach by using an Intel Xeon Phi 5110P as the acceleration target and show that the communication overhead can be reduced from 40% to 10%, thus enabling even small hotspots to benefit from offloading to an accelerator.**

I. Introduction

Modern computer systems are increasingly heterogeneous, i.e., they augment conventional CPUs with different kinds of computing accelerators. The types of accelerators are manifold, e.g., many-core architectures – like the Intel Xeon Phi or GPUs – or reconfigurable architectures like FPGAs. For exploiting the performance and energy efficiency benefits, that are enabled by these accelerators, applications have to be adapted and optimized for the specific programming model and instruction set of the available accelerators. Performing this adaptation for an ever-evolving set of accelerators is tedious and error prone. Hence, it is desirable to automate the process of porting applications to new architectures. When the source code is available, developers can use compilers and libraries to port and optimized the software to the new accelerator technologies. However, when dealing with closed-source and legacy software, users and developers that rely on specific binary applications or libraries, have no tools available to convert their software to a form that profits from the benefits of heterogeneous computing. One technique that could play a role in this area is just-in-time compilation. Common just-in-time compilation systems focus on transforming code into more optimized versions running on the same architecture and do not consider offloading program parts to accelerators. Therefore, a more general approach than just-in-time compilation is needed to exploit the full performance of modern computer systems when running existing software.

This work is based on *Binary Acceleration At Runtime (BAAR)*, an approach and implementation to enable easy-to-use on-the-fly binary program acceleration [1]. BAAR has a client-server architecture. The BAAR Client provides an LLVM-based environment to execute binaries in the form of LLVM Immediate Representation (LLVM IR). During execution, a binary is profiled and analyzed in this environment. Compute-intensive parts are extracted into the *offloaded part* and sent to the server. While the original program continues to be executed on the client side, the BAAR Server optimizes the offloaded part for an available accelerator and provides interfaces to the client to execute these parts of the program as a remote procedure call on the accelerator. The decision whether to execute the code locally on the client side or remotely is decided at runtime depending on the actual arguments and data volume.

While our approach uses LLVM IR as the binary format, this does not limit the approach to applications available in LLVM IR as several projects have shown that it is feasible to transform different binary formats into LLVM IR [2]–[4]. BAAR has been designed to be modular and flexible. The client does not have to know which specific accelerator is available on the server, only the characteristics of the accelerator. It is also conceivable for the server to provide multiple targets. Client architecture, communication mechanism and acceleration target are independently exchangeable. This allows runtime adaptation in changing environments and optimization for different goals, e.g., improving performance of legacy code on a workstation or minimizing power consumption of low-power computing devices. The exchangeability of the communication mechanism allows the BAAR approach to be used in a wide range of granularity and use cases, e.g., a desktop computer offloading computations to a remote server in a data center, or a CPU calling an FPGA on the same system-on-chip. Preliminary evaluation on a prototype [1] have stressed the importance of a low-overhead communication protocol between BAAR Client and Server. This work briefly introduces the overall approach of BAAR. Afterwards, the main contribution of this work is presented: A low-overhead communication mechanism for on-the-fly program acceleration in a client-server architecture for a changing environment.

The remainder of this paper is structured as follows. In Section II we discuss related work, in particular runtime

* This work was conducted while Marvin Damschen was student at the University of Paderborn

978-3-9815370-4-8/DATE15/©2015 EDAA

systems which try to dynamically adapt programs at runtime. Afterwards, Section III gives an overview of the architecture and toolflow of our approach and Section IV describes the new communication system. In Section V we compare the performance and finally draw a conclusion in Section VII.

II. Related Work

Optimizing existing software for new technology is an active research topic, with several projects trying to tackle this problem in various ways. In offloading-based approaches, communication is a crucial design aspect.

The Intel SPMD Program Compiler (ispc) [5] follows a static approach. It compiles a C-based programming language with single program multiple data (SPMD) extensions, enabling programmers to exploit vector units and multiple CPU cores in a familiar language without the need to know intrinsics. For ispc, the program source code has to be available in a high-level language and needs to be manually adapted. Once compiled, there is no way of adapting the binary program to changing environments at runtime.

KernelGen [6] is a compiler pipeline and runtime environment for standard C and Fortran code. The code is compiled into binaries containing CPU code and GPU kernels. At runtime the GPU kernels are automatically parallelized and JIT-compiled using runtime information. KernelGen is specialized on NVIDIA GPUs, communication between accelerator and host is highly specific. Other setups, with different types of accelerators or remote execution, are by design not supported.

The Sambamba project [7] aims at dynamically adapting programs to available resources at runtime. C/C++ code is compiled into an intermediate form, containing function definitions in sequential and automatically parallelized versions. This intermediate code is linked with a runtime environment into a fat binary. The runtime environment just-in-time compiles the intermediate code and dynamically adapts the execution by gathering information and running either the sequential or parallel version of functions per call. Similar to classic just-in-time compilation, the adaptation is limited to the target machine the program is started on. Accelerators or remote execution are not supported.

Warp processing [8] introduces a new processing architecture that operates on a standard binary. At runtime, the Warp processor automatically detects the binary's compute-intensive parts and utilizes an FPGA to reimplement them as a custom hardware circuit, the previously detected software parts are replaced by calls to the new hardware implementation. While the goals of accelerating standard binaries is the same as in our work, we do not focus on a specific accelerator architecture. Ultimately, we aim for supporting several accelerators with different characteristics and always utilizing the most fitting one under changing demands.

III. Approach

In this section, we first describe the overall client-server architecture of our *Binary Acceleration At Runtime (BAAR)* system. After the general overview, we present the acceleration procedure to detect application hotspots, the code generation of optimized heterogeneous executables and our support of

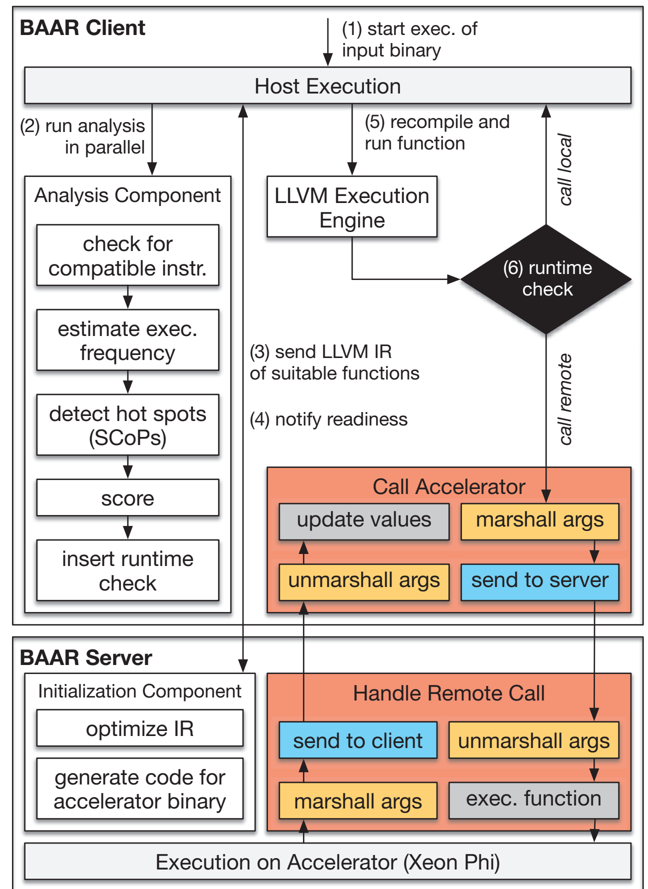


Fig. 1: Overall architecture and acceleration procedure execution of BAAR.

runtime checks to guide the migration. As a last point, we discuss the communication mechanism between the client and the server in more detail to motivate this work.

A. Overall Architecture

The conceptual architecture of our system and the sequence of execution is depicted in Figure 1. It consists of the client shown at the top, and the server at the bottom. The server is initialized in the *Initialization Component*. The client executes an application and performs the analysis, profiling and scoring of functions in the *Analysis Component*. When a function is suitable for the accelerator – provided by the server – and has at least one promising hotspot, it is exported. The server receives the exported functions (offloaded part), performs architecture dependent analyses, vectorization, parallelization and standard optimizations. Finally, it generates an executable binary for the accelerator. After this, the client is informed that the server is ready. The decision whether a specific hotspot will be executed remotely or locally is made at runtime for each specific call of the hotspot, depending on the expected benefit. Hence, the client inserts code for runtime decisions into the corresponding functions, recompiles them in the *LLVM Execution Engine* and dynamically decides whether to execute

the function on the local CPU or the remote accelerator card. In the latter case, it provides the required data to execute the function remotely and initiates the communication with the server.

B. Acceleration Procedure

The acceleration procedure can be divided into client and server tasks. The *BAAR Client* receives the application binary in LLVM IR and starts (Figure 1 (1)) the execution in the LLVM Execution Engine in a just-in-time fashion. The main task of the client is identifying suitable functions to offload to the server (Figure 1 (2)). Note, that this is done in parallel to the execution of the application. A suitable function for offloading has the following properties: 1) it is only composed of instructions which are supported by the server/accelerator 2) it has at least one basic block with a high estimated number of execution (frequency), and 3) it can profit from the advantages of executing on the server. Suitability can be ensured by code analysis with LLVM Passes [9]. By ensuring the last two properties we guarantee that the code can be executed in a sufficient number of threads to leverage the full performance of a many-core accelerator like the Intel Xeon Phi accelerator. Therefore, we compute the number of operations which can be parallelized during the optimization by detecting Static Control Parts (SCoPs), maximal sets of consecutive statements where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters [10], [11]. For every candidate function with at least one SCoP we further analyze loop nests of the SCoPs. Within the loop nest, each instruction type is determined and assigned with a configurable weight. All weights are aggregated and multiplied by the frequency of the innermost basic block of the loop. The result is the *score* of the loop nest. All loop nest scores of all SCoPs are summed up to build the function score, according to Equation 1. A higher score means a better suitability for the accelerator card. The remaining candidate list of suitable functions builds the remote part and is sent as LLVM IR to the server (Figure 1 (3)).

$$\begin{aligned} \text{score}_{\text{loop}} &= \text{innerBBFreq}_{\text{loop}} \cdot \sum_{\text{instr}\in\text{loop}} (\text{type}_{\text{instr}} \cdot \text{weight}_{\text{instr}}) \\ \text{score}_{\text{func}} &= \sum_{\text{SCoP}\in\text{func}} \sum_{\text{loop}\in\text{SCoP}} \text{score}_{\text{loop}} \end{aligned} \tag{1}$$

The main task of the *BAAR Server* is to optimize the received code and to provide an acceleration target to execute remote procedure calls from the client. The automatic optimization focuses on target-specific passes, parallelization and vectorization, to fully utilize the Xeon Phi accelerator card (Figure 1 *Initialization Component*). Parallelization is based on polyhedral optimization provided by the LLVM subproject Polly [10]. Polly offers an interface to polyhedral optimizations using CLooG and isl to the LLVM world. It enables us to transform sequential SCoPs into thread-parallel code utilizing OpenMP. For vectorization, we use the Loop Vectorizer and Superworld-Level Parallelism (SLP) Vectorizer provided by LLVM. Unfortunately Xeon Phi is not yet a supported target, and hence we use the x86 target to emit suitable vector instructions. After optimization, we need to transform the obtained LLVM IR into an executable for the accelerator card. As there is currently no Xeon Phi backend available for LLVM, we use an adapted C backend from ispc [5] and a detour over the Intel Compilers to generate the appropriate binary from LLVM IR for the accelerator.

Regardless of the actual score of a function, the payoff of offloading the computation to the server depends on the saved computation time and additional communication time, which is determined by communication latencies and the volume of data that needs to be transferred. Therefore, we insert a *runtime decision* to determine whether to use the local function or a remote procedure call to the server (Figure 1 *Analysis Component*). For every function we insert a new basic block at the beginning to determine the total number of bytes that needs to be transferred and then evaluate if the fraction $\frac{\text{score}_{\text{func}}}{\text{total bytes to transfer}}$ is greater than a configurable threshold. If this check is evaluated to true (Figure 1 (6)), the client dynamically starts the initialization of the remote call. A detailed discussion of our technique and the benefits or deferring offloading decisions to runtime is discussed in our previous work [12]. In the next section, we discuss the steps required for a remote procedure call in more detail and present our improvements.

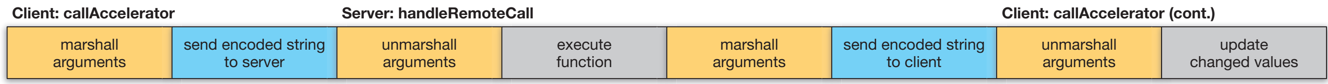
IV. Communication Mechanism

In our initial implementation of BAAR [1], which was the basis for this work, we executed the remote call by marshalling the function signature and arguments into a concatenated string, which is then sent to the server over a TCP/IP socket. The server unmarshalls the string, maps the information to corresponding data structures and executes the call on the accelerator card. After the execution is complete, the return value and possible modified arguments are marshalled and sent back to the client.

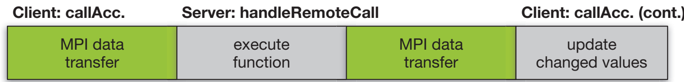
This communication mechanism is very simple and thus can lead to inefficiencies in many cases, but it handles architectural differences easily (e.g. memory layout) between the client and server. This operation is especially costly for transfers of arrays, because each element needs to be encoded into the string representation. To offload as many functions as possible to the accelerator, the costs for the remote procedure call and migration of data should be minimized. Otherwise a suitable function for which the time taken to transfer the arguments is longer than the time saved by using the accelerator.

Figure 2a illustrates a breakdown of the communication time in our baseline implementation of BAAR that used TCP/IP sockets. Once the socket connection between the client and server is setup, it is used to send the offloaded part as well as transfer data to perform remote procedure calls. Based on the sequence of messages, the communication overheads can be broken down into two parts: 1) one-time overheads 2) recurring overheads. The one-time overhead is incurred only once during the setup of remote (accelerated) function, whereas the recurring overhead is associated with every remote function call.

1) One-time Overheads: When the client decides to offload a function to the server, it first connects to the server and sends the offloaded part. This needs to be done only once (for the first



(a) with (un-)marshalling of arguments and TCP/IP sockets.



(b) with MPI data transfer.

Fig. 2: Required operation to perform a remote procedure call

function call) as the LLVM IR code for successive function calls is then already present on the server. The communication overhead associated with this operation is the time required for the client to connect to the server and the time required to transfer the LLVM IR. As sockets are used for the connection, these overheads can vary based on the performance of the underlying network, and are governed by network properties such as bandwidth, throughput and latency.

2) Recurring Overheads: In order to execute the function, in addition to the function definitions in the offloaded part, the server also requires the associated data that the function operates on (function arguments). In the case of primitive data types, the value can be easily transferred, however in the case of pointers (arrays), the entire underlying data structure needs to be transferred to the server and once the computation is complete, transferred back to the client. This needs to be done every time the function is called on the server.

A. Previous Communication Mechanism

Figure 2a represents the communication mechanism used in our previous work. When the client wants to perform a remote procedure call, the function arguments need to be transferred. Therefore we first marshal the function arguments by encoding the data into a delimited string. This string holds the signature of the function as well as the size of the data, followed by the actual data for each function argument. Once the string is received by the server, it first needs to unmarshal the data before it can execute the function call. After the function has completed its execution, the same procedure is used to send the updated content from the server back to the client. These operations need to be performed for every function call.

The time required for marshalling or unmarshalling the parameters depends on the size of the arrays and the number of arguments. The time required to transfer the data depends on the size of the message (delimited string) and the network parameters. The size of the message depends on the number of elements and also the data type, e.g., it takes fewer characters to represent an `int` as a string as compared to a `double`.

By looking at these steps, we can say that in order to reduce the communication overhead, we need to minimize the time taken to marshal/unmarshal the arguments as well as reduce the size of the message.

B. Improved Communication Mechanism

In our approach, we minimize the communication overhead by making use of the Message Passing Interface (MPI) to transfer the data. MPI is a library interface specification designed for the message-passing programming model in an environment of parallel processes. The main emphasis is on data movement from the address space of one process to another, abstracting different characteristics between the system architectures where the processes are running. It was proposed as a standard by a broadly committee and there are several free and vendor-specific implementations (e.g. Open MPI, Intel MPI) available. By using MPI, we have the following advantages.

1) Faster communication: In the previous approach we used sockets as the basis of the communication mechanism between the client and server. This limits the communication speed to that of the TCP/IP communication channel. However, MPI can transparently and automatically detect and select the best available communication channel, e.g., shared memory, InfiniBand, TCP/IP, or PCIe. In our evaluation, MPI enables a lightweight, high-performance communication between BAAR Client on the CPU and BAAR Server on the Xeon Phi accelerator card.

2) Marshaling and unmarshalling: In our previous approach, we had to first marshal and the unmarshal the arguments. This was necessary, as we had to transfer data across different architectures and it was important to guarantee that the right data was transferred in spite of differing byte orders for different architectures. MPI on the other hand, handles this conversion internally and supports a broad variety of data types. In addition, by not using a string representation for the arguments encoding, the required data to be transferred can be reduced.

Thus, by using MPI we can eliminate the need to explicitly marshal and unmarshal data at the client and server, as depicted Figure 2b. Instead, the MPI library implicitly uses its internal, highly optimized marshalling routines. Additionally, we can also reduce the one-time overhead by sending the LLVM IR over MPI, as it utilizes the best available communication channel.

V. Evaluation

In this section we evaluate the practicality of our approach. The improved prototype, denoted as BAAR$_{mpi}$, is used to

execute stencil codes with different input sizes N. We compare the results to our last prototype BAAR$_{tcp}$ and CPU execution only on the target platform described in the following section.

A. Setup

For evaluation purposes, we choose one specific target platform: An Intel Xeon E5-2670 machine equipped with an Intel Xeon Phi 5110P accelerator card, communicating over PCIe. The host uses two 8-core (16 threads) Sandy Bridge CPUs running at 2.6GHz and provides 64GB of main memory. The Xeon Phi card features 60 cores at 1.053GHz, 8GB of main memory and runs a dedicated Linux itself besides the operating system running on the Xeon E5 (Scientific Linux 6.4 with Intel Many Core Platform Stack 2.1.6720). The BAAR approach is more widely applicable, however. The current implementation already allows remote execution on an Intel Xeon Phi in a separate machine. Future work will generalize the implementation to target different types of accelerators, e.g., GPUs, FPGAs or CGRAs and investigate algorithms for adapting to a changing environment.

The CPU binaries are obtained from C files taken from the Polybench benchmark suite [13]. They are compiled with Intel Compiler version 14.0.0 using optimization level O2 (for speed) running natively on the Xeon E5. For BAAR$_{tcp}$, the client runs on the Intel Xeon E5 and the server on an Intel Xeon Phi accelerator card, which also functions as the target for automatic parallelization and vectorization and for executing the automatically offloaded function calls. For BAAR$_{mpi}$ the executables are deployed in the same way, but the execution is started on the Xeon E5 with `mpirun` to creates the required MPI environment.

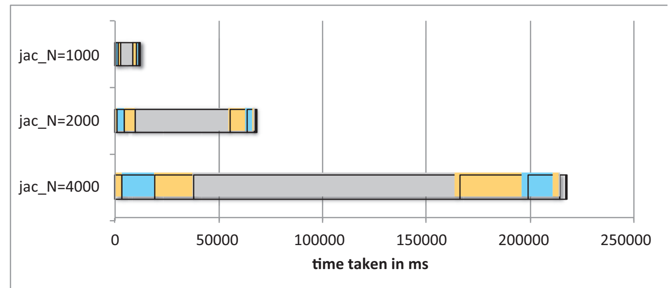
The executed stencil for our evaluation is the 2D Jacobi stencil (`jac`). The number of iterations S is set to 10.000 and the stencil is executed on an array of size N^2 with $N \in \{1000, 2000, 4000\}$.

B. Raw Communication Performance

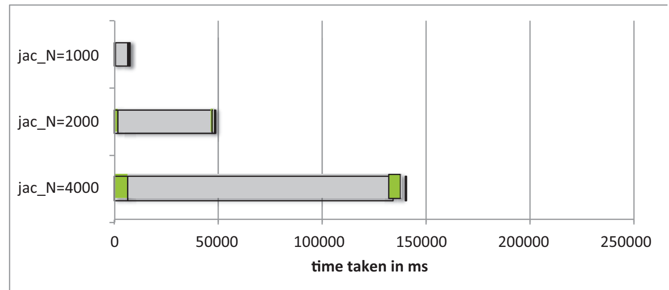
In this subsection we look at an in-depth analysis of how much time is spent in which phases of communication, when a remote function call is performed. The generally required steps are depicted in Figure 2 and in this evaluation we match these steps to actual time taken to perform each of the steps.

Figure 3a shows the amount of time spent per phase for the TCP/IP approach in BAAR$_{tcp}$. Overall, we spend around 42% of the total time for communication. The time required to transfer the encoded string from the client to the server and from the server back to the client is nearly the same (7.2% and 6.6% on average, respectively), as the whole array the stencil operated on has to be transferred back. However, marshalling and unmarshalling the same amount data at the server side takes 6-10× longer than that at the client side. This is because a single processor core of the Xeon Phi accelerator card is not as powerful as one of the client and impairs the sequential marshalling.

Figure 3b shows the amount of time spent for our new MPI approach BAAR$_{mpi}$. The improved approach, spends only around 8% of the time on communication in total. We can see that by using MPI we no longer incur any overhead to



(a) Execution of a remote procedure call of the jacobi stencil with BAAR$_{tcp}$ for different input sizes N.



(b) Execution of a remote procedure call of the jacobi stencil with BAAR$_{mpi}$ for different input sizes N.

Fig. 3: Direct comparison of communication mechanisms.

marshal or unmarshal the data at the client or server side. We can also see that by using MPI, we also benefit from the raw data transfer phase as it takes us less time to transfer less data between the client and server as compared to the encoded string.

C. Overall Performance

In this section, we look at the overall performance of our MPI approach with respect to the TCP/IP approach and the CPU. Figure 4 represents the total execution time of the 2D Jacobi stencil for the different approaches. The bars are split based on the computation time (red) and the communication time (blue). The CPU has no communication overhead as all the code is executed locally without any remote procedure calls. In the figure, we can clearly see the difference in communication overheads between the TCP/IP and MPI approaches. By using the MPI approach, we are able to achieve a speedup of around 1.6× over the TCP/IP approach and 3× over the CPU approach.

VI. Availability

BAAR is open-source software and is licensed under the MIT license. The code can be found at https://github.com/pc2/baar.

VII. Conclusion

In this paper, we have presented the transparent and automatic offloading of computational intensive parts (hotspots)

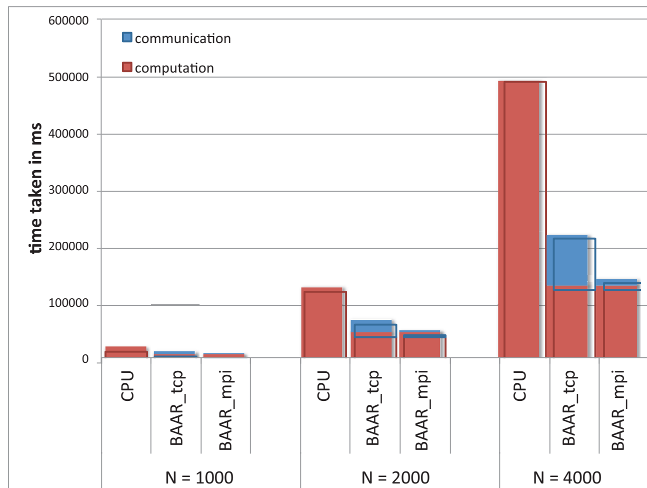


Fig. 4: Direct comparison of the overall performance for different input sizes.

at application runtime. Our approach *Binary Acceleration At Runtime (BAAR)* consists of a client-server architecture and is based on the LLVM Compiler Infrastructure. We have developed a acceleration mechanism which is able to utilize parallelization and vectorization on the Intel Xeon Phi accelerator card and has an efficient communication mechanism over MPI.

Our measurements show that, without any developer-provided hints, a speedup of $3\times$ is achievable when comparing the execution of the Jacobi 2D stencil from Polybench with BAAR to the execution of native code generated with the Intel Compilers with optimization level O2. Furthermore we reduced the communication overhead from 40% to 8% for remote procedure calls. As there is currently no Xeon Phi backend available in LLVM, we need to use a detour over C code generation and use of the Intel native compiler to obtain Xeon Phi binaries from LLVM IR. This impairs the quality of the vectorization and the performance of BAAR in general. BAAR is however prepared to utilize the native Xeon Phi LLVM backend once available, promising even higher speedups.

In future work, we plan to investigate in integrating techniques for decompilation of real x86 binaries into LLVM IR. Currently, we obtain LLVM IR using Clang to compile C/C++ source files. Previous works have shown that decompilation of binaries to LLVM IR is feasible [2]–[4], but it is unclear if the semantical richness of the result is sufficient in respect to the optimizations performed. It could worsen or prevent some optimizations. Another interesting topic is the execution of client and server in a network environment in which client and server do not necessarily execute on the same machine. One interesting application scenario is a set of rather weak micro-server CPUs that offload computationally expensive operations to dedicated servers with accelerators. In this case, MPI should automatically chose the most efficient means of communication (e.g., Inifiniband, Ethernet, or PCIe), which was our main motivation to build the communication mechanism on MPI.

Acknowledgement

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and the European Union Seventh Framework Programme under grant agreement no. 610996 (SAVE).

References

[1] M. Damschen and C. Plessl, "Easy-to-use on-the-fly binary program acceleration on many-cores," in *Proc. Int. Worksh. on Adaptive Self-tuning Computing Systems (ADAPT)*, Jan. 2015.

[2] "Dagger - decompilation framework," Website: http://dagger.repzret.org/, [Online; accessed 20-October-2014].

[3] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proc. Europ. Conf. on Computer Systems (EuroSys)*. New York: ACM, 2013, pp. 295–308.

[4] V. Chipounov and G. Candea, "Enabling sophisticated analyses of x86 binaries with RevGen," in *Proc. Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 211–216.

[5] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *Proc. Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–13.

[6] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström, "KernelGen the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs," University of Lugano, Tech. Rep. 2013/02, Jul. 2013.

[7] K. Streit, C. Hammacher, A. Zeller, and S. Hack, "Sambamba: A runtime system for online adaptive parallelization," in *Proc. Int. Conf. on Compiler Construction (CC)*. Springer, 2012, pp. 240–243.

[8] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," in *Proc. Design Automation Conference (DAC)*. New York: ACM, 2004, pp. 659–681.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, Palo Alto, California, Mar. 2004.

[10] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet, "Polly – polyhedral optimization in LLVM," in *Proc. Int. Worksh. on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[11] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 1998, pp. 106–111.

[12] G. Vaz, H. Riebler, T. Kenter, and C. Plessl, "Deferring accelerator offloading decisions to application runtime," in *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE Computer Society, Dec. 2014, accepted for publication.

[13] "Polybench/C – the polyhedral benchmark suite," Website: http://web.cse.ohio-state.edu/~pouchet/software/polybench/, [Online; accessed 20-October-2014].