Potential and Methods for Embedding Dynamic Offloading Decisions into Application Code

Gavin Vaz, Heinrich Riebler, Tobias Kenter, Christian Plessl

Department of Computer Science, Paderborn University, 33098 Paderborn, Germany
Email: \{gavin.vaz — heinrich.riebler — kenter — christian.plessl\} @uni-paderborn.de

Abstract

A broad spectrum of applications can be accelerated by offloading computation intensive parts to reconfigurable hardware. However, to achieve speedups, the number of loop iterations (trip count) needs to be sufficiently large to amortize offloading overheads. Trip counts are frequently not known at compile time, but only at runtime just before entering a loop. Therefore, we propose to generate code for both the CPU and the coprocessor, and defer the offloading decision to the application runtime. We demonstrate how a toolflow, based on the LLVM compiler framework, can automatically embed dynamic offloading decisions into the application code. We perform in-depth static and dynamic analysis of popular benchmarks, which confirm the general potential of such an approach. We also propose to optimize the offloading process by decoupling the runtime decision from the loop execution (decision slack). The feasibility of our approach is demonstrated by a toolflow that automatically identifies suitable data-parallel loops and generates code for the FPGA coprocessor of a Convey HC-1. We evaluate the integrated toolflow with representative loops executed for different input data sizes.

Keywords: Runtime System, Runtime Decision, Hotspot Offloading, Convey HC-1, LLVM, SCEV

1. Introduction

With the rise of heterogeneous computing using accelerators in mobile and general-purpose systems and the advent of Field Programmable Gate Arrays (FPGAs) in the data center [?], the question of optimal application partitioning across the HW/SW boundary is as important as ever. As FPGAs become increasingly accessible through techniques like high-level synthesis or overlay architectures, fast and automated yet accurate techniques are required to determine which parts of an application can benefit from being offloaded to an FPGA accelerator. Traditional methods have relied either on static code analysis or on profiling data to characterize suitable application hotspots. If the application’s behavior cannot be statically determined at compile time or if it is heavily input data or parameter dependent, partitioning decisions need to be taken heuristically, possibly leaving significant optimization potential on the table.

Therefore we propose to defer the actual decision to offload a particular code section to an accelerator to program runtime, where dynamic information about the execution is available, for example, the loop trip count for data-dependent hot loops. We propose to automatically insert code that performs these runtime decisions into the application code. During a static partitioning process at compile time, code for both targets (CPU and FPGA accelerator) is generated. At runtime, before some potentially acceleratable code section is entered, the amount of computation and the size of data to be worked on can be determined in many cases and can be used for making better offloading decisions than static analysis only. For this purpose, we build a toolflow based on the LLVM compiler infrastructure [?], which allows us to target all applications that can be compiled to, or are distributed in the LLVM intermediate representation (LLVM IR).

In this work we investigate the possible benefits of offloading decisions at runtime in three connected experiments.

First, we evaluate how often common compute workloads offer an opportunity for runtime decisions under the assumption that offloaded code typically corresponds to loops or loop nests. For this evaluation we let our LLVM based toolflow statically (offline) analyze the code of three real-world benchmark suites and determine which loops can be precisely characterized at runtime (RT), but not at compile time (CT). It turns out that this is the case for a considerable fraction of all detected loops. However, to have a reasonable fraction of runtime dependent loops gives us only a necessary precondition for the usefulness of our approach. A runtime depended loop serves only as a good offloading candidate if, in addition, the execution frequency is sufficiently large. Therefore we also present a dynamic (online) evaluation of the application behavior where we analyze the execution frequencies of the different types of loop (nests) in more detail. We show that of all the detected loops not only are a considerably fraction runtime dependent, but that many of those loops also impose large execution frequencies. We also evaluate the overheads that are inserted into the applications taking offloading decisions at runtime.

Second, after showing the usefulness of runtime decisions for the offloading process we propose a new approach of rearranging or moving the decision point within the application. For a loop to be characterized as runtime decidable, all the infor-
mation required to take a runtime decision needs to be available at least right before the beginning of the loop (nest). We show that for most real-world applications from various benchmark suites, this information is available much before its actual usage in the loop. Hence, the decision could be taken earlier and the possible coprocessor execution could be prepared in the timespan gained moving the decision up. We created an LLVM-based toolflow which is able to detect and characterize these code movement opportunities. We present different cases where this information can be exploited to enhance the runtime decision and improve the offloading process.

Third, we evaluate the benefit of runtime decisions with an actual toolflow targeting the reconfigurable Convey HC-1 compute platform. The integrated toolflow can automatically identify loops suitable for vectorization and generate efficient code for the host CPU and a vector coprocessor implemented as an FPGA overlay. We have extended this toolflow to analyze for detected loops whether their dynamic offloading decision should be taken at runtime and insert the required decision into the application code. When offloading decisions are deferred to runtime, data movement between CPU and coprocessor also needs to be organized at runtime as well. Alongside our runtime decisions we generate code for proper data movement and again profit from analysis that uses runtime information to determine which amount of data needs to be transferred.

We apply our toolflow to a set of different, runtime-dependent loops with different nesting structures and evaluate the performance for different data dimensions. Our results show that runtime decisions can improve the overall performance of the system as compared to always executing vectorizable code either on the CPU or the coprocessor.

This paper extends our previous conference publication [2] with its static analysis of the potential of runtime decisions in real-world benchmarks by an in-depth dynamic analysis of these applications. We also propose for the first time the concept of decision slack and present a first static analysis of its potential. Finally, we provide additional details of our case-study and extend our coverage of related work.

The remainder of this paper is structured as follows. In Section 2 we discuss related work, in particular other approaches of systems with offloading decisions at runtime and related work targeting the Convey HC-1. In Section 3 we present our approach of inserting the offloading decision right into the code of the application and show an analysis of common benchmark suites and how widely applicable runtime decisions are. In Section 4 we introduce alternative insertion points for runtime decision and discuss our approach to prepay the decision to improve the offloading process. This new approach is evaluated in the same section with real-world applications. In Section 5 we present the integrated toolflow with runtime decisions and code generation for a reconfigurable coprocessor and describe the data migration strategy implemented for this platform. Then, we compare the performance of special aspects and the overall integrated toolflow in Section 6, and finally draw a conclusion in Section 7.

2. Related Work

In this section, we describe related work for the topics covered in this paper. First, we present background information on static HW/SW partitioning and describe other efforts in building systems with dynamic HW/SW partitioning. Then we discuss related research on data migration at runtime and complete this section with other research effort in characterization of common benchmark suites.

2.1. Static HW/SW Partitioning

One important challenge in the acceleration of applications with the help of specialized hardware is to decide which parts of the application shall be executed in hardware. To tackle this HW/SW partitioning problem, researchers proposed exact methods and heuristics that can be used for automated partitioning toolflows. Apart from the design space exploration strategies, the methods also differ in the granularity of the partitioning objects that are considered. Depending on the target architecture, methods work at the level of functions, loops, basic blocks and even instructions.

Static HW/SW partitioning approaches have been well-studied and are widely used to achieve performance improvements as well as energy savings as compared to a software-only or hardware-only approaches. Methods from this domain use code analysis and/or profiling information to determine, where each partitioning object will be executed. Once the partitioning decision has been made, however, the partitioning object will always be executed on the selected resource. Related work in this area is extensive, hence, we point out only a few representative works. For example, López-Vallejo and López present a survey on heuristic approaches [3], and Cardoso et al. [4] discuss partitioning from the perspective of compilers for reconfigurable computing.

Our work uses heuristics based on static code analysis and profiling for identifying promising code regions (loops and loop nests). The key difference to related work is that the partitioning decision, that is, whether to offload a loop nest is not static but taken at runtime based on the actual trip count for each entry.

2.2. Dynamic HW/SW Partitioning

Given a fixed HW/SW partitioning decision, the achievable speedup from offloading a kernel, such as a loop nest, to an accelerator is not constant. Instead the speedup depends on the raw speedup and the required overheads, i.e., control and data transfer times which can also result in an overall slowdown [5]. To ensure an effective use of accelerators resulting in a net speedup, the HW/SW partitioning decision can thus not always be made statically, but needs to consider for each kernel invocation the expected gain (performance, energy reduction) and overheads (data and control transfer latency) considering the current context and parameters of the invocation.

Runtime systems that support such dynamic HW/SW partitioning have so far received only limited attention. Beisel et al. [6] have proposed a runtime system that takes dynamic partitioning decisions at the library function level. Their approach is based on a shared library interposing mechanism that
transparency intercepts shared library calls. For each library call, a runtime system decides whether offloading the call to an accelerator will result in a speedup by considering data in a lookup table that contains information about the execution time for each supported function and for different data sizes. The benefit of this approach is that it can be applied completely transparently without any modification of the application. The scope of this approach is however limited, because only functions that are exposed in shared libraries are supported as partitioning objects and only function arguments can be used to take the partitioning decision, which prohibits a detailed analysis of the function code.

Recently, Wen et al. [2] have studied a similar problem for dispatching OpenCL kernels to a GPU in a CPU + GPU system. Like the previous work, they target a coarse-grained level of partitioning objects. The main contribution of their work is the application of machine learning techniques to estimate the execution time of a kernel on the GPU based on static and a few dynamic code features.

Stitt et al. [3] tackle the problem of dynamic HW/SW partitioning from a different angle. They not only move the partitioning process to runtime but also the process of identifying kernels and automatically synthesizing a suitable FPGA accelerator for accelerating them. Their approach targets kernels which can be expressed as applying a combinational function to a 1D data stream. After completion of the synthesis process, the original binary application is changed to henceforth use the hardware accelerator, thus the kernels are statically offloaded.

Similarly, Bispo et al. [4] also presented a just-in-time hardware acceleration approach. They extract patterns of machine instruction traces (loops) at basic block level and map them to a reconfigurable processing unit implemented with an FPGA. They use offline simulation to detect suitable loops and then make use of offline partitioning and synthesis, while the replacement is done online (at runtime). The replacement is done transparently by monitoring the instructions during execution and modifying the program counter. Like Stitt et al., there is no runtime decision whether the loop shall be offloaded.

While our approach is related to these works in the overall strategy to move decisions from compile time to runtime, it differs in several fundamental aspects. We share the goal to make the offloading process seamless to the developer and not require application modifications, but in contrast to [2, 4] we do not assume that applications are available in a CPU-specific binary format. Instead, we target LLVM IR as the input language, which can be considered an abstract binary format but directly exposes control flow information without decompilation. Using LLVM IR also allows our approach to be much more fine-grained than an approach targeting library calls [4] or OpenCL kernels [3]. Library calls are restricted to function level, so the granularity of this approach is limited compared to our approach, which can directly extract hot loops. Additionally, many kernels cannot be targeted by their approach, because they are directly embedded in the source code and not implemented through library calls. Finally, our static code analysis changes the original code to include expressions that are evaluated at runtime and allow to characterize the dynamic behavior of the application, in particular, the trip count of data dependent loops. This approach allows us to get accurate information about the application behavior that can be used for making well-founded dynamic partitioning decisions, without resorting to approximations like table lookups or machine learning.

2.3. Data Migration

Additionally, the locality of data also plays a major role in the overall system performance. The required data should always be present as near as possible to the processing unit to avoid costly recurring data transfers over the bandwidth. The overall system performance is maximized, when the data transfer overhead can be minimized. Thus fast and accurate data migration methods become a crucial factor. Lange et al. [5] present an approach to handle pointer-based algorithms in a common address space between CPU and coprocessor. Meswani et al. [6] propose a simple performance model for the Convey HC-1. However, they just outline the importance of data migration with a preliminary analysis and do not support any runtime decisions.

Wang et al. [7] present a system for data migration at runtime similar to ours, combining static compile-time analysis with the insertion of calls to a placement engine that considers data sizes at runtime, when they are known prior to a kernel invocation. However, they consider a very different target system. Their single compute platform is a GPU, which contains hybrid memory consisting of Phase Changing Memory (PCM) and DRAM. Thus, in contrast to our work, their data placement and migration decisions are not related to an offloading decision, but rather just update the placement depending on the sequence and data access properties of executed kernels. Wang et al. consider constrained sizes of each distinct memory type, which we did not need to do in our case study because of sufficiently large coprocessor memory.

2.4. Benchmark Characterization

Another part of our contribution is related to the benchmark characterization domain. Jaleel et al. [8] analyzed workloads from SPEC CPU2000 and CPU2006 to determine the memory system requirements with the help of binary instrumentation. Their approach is based on instrumentation-driven simulation with Pin whereas we leverage the LLVM infrastructure on LLVM IR. Packirisamy et al. [9] use a profile-driven compiler to identify loops in SPEC2006 that can be speculatively parallelized. Kashnikov et al. [10] present a static loop analyzer to extract from binary loops a number of low-level assembly features like the ratio of integer/fp vectorization ratio, or the number of registers used. They propose to use this information to tune the compilers and architectures for the most frequently patterns. In contrast to ours, their approach requires the binary to be compiled with debug information (flag -g) to be able to map between the binary and source code.

De Alba et al. [11] show for SPECint2000 that a static compiler can miss a great share of loops for hardware unrolling. In comparison to our work they also use additional runtime information to use bridge the limitations of a static-only approach.
Similar studies [?] exist for the embedded domain i.e. MediaBench and MiBench, partly with a stronger focus on power analysis and energy efficiency [?].

3. Runtime Decisions

In the first part of this section, we look at how loops can be classified and how we make use of the LLVM infrastructure to generate runtime decisions. We then analyze real-world general-purpose and embedded applications first statically and then dynamically to determine the applicability of our approach. Finally, we look at the overhead of such a runtime decision.

3.1. Generating Runtime Decisions

Regardless of the actual accelerator and acceleration method used, the payoff of offloading the execution of a loop may heavily depend on its trip count. When the loop’s iteration space depends only on constants, we follow the LLVM terminology and denote this property as compile-time (CT) computable trip count, meaning that decisions can be taken statically at compile time. In traditional approaches, all other loops need to be characterized by profiling. However, the iteration space of many loops depends on variables that are not changed during the loop execution itself, so their trip count is actually fixed at the point where the loop is entered. We denote this loop property as runtime (RT) computable trip count. In a third class of loops however, for example when a complex, data-dependent exit condition is computed in every loop iteration, the loop trip count is not computable (NC).

Our analysis is implemented with the help of LLVM’s Scalar Evolution (SCEV) analysis pass1. SCEV analysis is specialized to recognize induction variables in loop iterators. Induction variables are variables that are increased/decreased by a specific (fixed) value for each iteration of a loop, or can be expressed as a function of other induction variables. By using the LLVM API, we are able to get an expression for the trip count for a specific loop. For trip counts that can be computed at compile time, the API returns a single constant value SCEVConstant defining the number of iterations for that loop or an expression containing with constant operands. In other cases, the trip count is expressed as being dependent on a runtime variable SCEVUnknown or an expression containing at least one of those variables. If however SCEV analysis cannot express the trip count, it returns SCEVCouldNotCompute.

Listing 1 shows a code example for a function with a loop nest whose inner trip counts cannot be computed at compile time, because they depend on the runtime variable height. Manually analyzing those trip counts would determine the individual trip counts from outer to inner loops as denoted in Equation (1) to (3). The corresponding trip counts as determined by the SCEV analysis are represented as simple binary expressions for the first two outer-loops and a more complex nested expression for the most-inner third one. The first expression depends only on the named constant WIDTH. Therefore the expression count1 can be statically evaluated at compile time (CT).

\[
\begin{align*}
\text{count}_1 &= \text{WIDTH} & \text{// CT} (1) \\
\text{count}_2 &= \text{height} & \text{// RT} (2) \\
\text{count}_3 &= \max(0, \left\lfloor \frac{\text{WIDTH}+\text{height}}{2} - \frac{\text{max}}{2} \right\rfloor) & \text{// RT} (3)
\end{align*}
\]

However, the other two expressions count2, count3 depend on the runtime variable height and at analysis time they are only available in our custom LLVM pass, so we require a mechanism to integrate it into the binary so that they can be evaluated at runtime. The idea is to generate LLVM IR for the expression so that we can insert it into our runtime decision. For this, we make use of LLVM’s SCEVExpander class, which is capable of generating LLVM IR from a given ScalarEvolution (SCEV) expression. The generated LLVM IR is a sequence of LLVM instructions that computes the trip count, whereas the last instruction holds the actual value of the loop trip count at runtime, as illustrated for the computation of count3 in the top part (line 1 to 6) of Listing 2.

Listing 2: Generated LLVM IR code for computing trip count count3 and associated simple runtime decision as comparison with threshold.

Utilizing the thus computed loop trip count, we can now similarly insert LLVM IR to perform a runtime decision on where to execute the loop in question. In the simplest form, as demonstrated in the bottom part of Listing 2 (line 8 to 10) and later in our integrated toolflow in Section 5, this can be a comparison of the trip count to a general or loop-specific threshold. Depending on the outcome of the comparison instruction in line 9, either a jump to the code that dispatches execution to the accelerator %call_acc is executed, or execution continues in the host binary, labeled as %cont_cpu.

3.2. Static Analysis of Loop Types

In order to investigate how often the method of computing trip counts at runtime is applicable for real-world applications,

---

1llvm.org/docs/Passes.html#scev-aa-scalar-evolution-based-alias-analysis
we analyze common benchmarks for general-purpose computing and embedded systems: SPEC CPU2006 [? ] and MediaBench [? ]. The MediaBench source code is obtained from the LLVM 3.6 testing infrastructure (test-suite2) and the other code corresponds to the original sources. The tooflow of our approach is illustrated in Figure 1. We first compiled all the benchmark sources into LLVM IR, as it is the required input format for our analysis pass. To do this, we used the clang/clang++ (version 3.6) front-end to compile the C/C++ files and the DragonEgg plugin (version 3.5) for the F/F90 files. The tooflow of our previous conference publication [? ] was based on the older version 3.4. We updated all tools to the latest version and present the most recent results. It is interesting to note that before we analyze the LLVM IR, we normalize it by transforming the loops into their canonical (normal) form by applying the mem2reg, loop-simplify, lcssa and indvars LLVM transformation passes3. These passes promote memory operations to registers, canonicalize natural loops, transform loops into LLVM’s loop-closed static single assignment (SSA) form and canonicalize the induction variables respectively. Additionally, we also perform some code optimizations like move loop invariant code outside the loop, merge duplicate global constants, combine redundant instructions and eliminate dead code.

Table 1: Static analysis of loop trip count properties for different benchmarks.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>number of loops</th>
<th>compile-time</th>
<th>runtime</th>
<th>not computable</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC CPU'06</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- C/C++</td>
<td>27,176</td>
<td>3,466 (13%)</td>
<td>5,4987 (20%)</td>
<td>18,213 (67%)</td>
</tr>
<tr>
<td>- F/F90</td>
<td>46,499</td>
<td>5,010 (11%)</td>
<td>24,933 (54%)</td>
<td>16,556 (35%)</td>
</tr>
<tr>
<td>- TOTAL</td>
<td>73,675</td>
<td>8,476 (12%)</td>
<td>30,430 (41%)</td>
<td>34,769 (48%)</td>
</tr>
<tr>
<td>MiBench</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- C</td>
<td>2,686</td>
<td>446 (17%)</td>
<td>349 (13%)</td>
<td>1,888 (70%)</td>
</tr>
<tr>
<td>MediaBench 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- C</td>
<td>669</td>
<td>177 (26%)</td>
<td>186 (28%)</td>
<td>306 (46%)</td>
</tr>
<tr>
<td>TOTAL</td>
<td>77,027</td>
<td>9,099 (12%)</td>
<td>30,965 (46%)</td>
<td>36,963 (46%)</td>
</tr>
</tbody>
</table>

3.3. Detailed Dynamic Analysis of Runtime Decidable Loops

In the previous section, we saw that the trip count of a relatively large number of loops could be determined at runtime. However, in our (static) analysis we only considered the overall opportunity of a runtime decision for real-world applications but did not take into account the suitability of loops for vectorization or any other form of acceleration. It rather shows the overall opportunity of a runtime decision for real-world applications. The results are presented in Table 1. The ratio of compile-time, runtime and not computable trip counts for C/C++ code is somewhat similar across all three considered benchmarks. About a quarter of the loops can be described by scalar expressions (SCEV) for a runtime-check. For Fortran (F/F90) the ratio differs significantly. The majority of all loops (54%) can be used for a runtime-check. Overall (40%) of loops across all three benchmarks qualify for a runtime-check.

In order to understand the character of loops with uncomputable trip counts (NC), we looked at the source code of a few dozen randomly picked examples from the benchmark suites. Most of the loops we encountered had a data-driven exit condition, for example they are iterating over dynamic list or graph data structures, have input driven loops that exit when encountering some termination symbol, or they exit when a computed value meets a given condition. For some loops we found, the trip count can not be definitely computed, so LLVM classifies them as cannot compute, for example because the exit condition depends on a global variable that may be changed from some other module. However, after analyzing the code’s purpose, we assume that such side effects are either very rare of fully unintended. For these cases, a likely trip count, something that is currently not computed by LLVM, but could be integrated, might enable educated guesses for the offloading decision. Also, further likely trip counts might be obtained by automated or manual code transformations, for example by taking file sizes into account, when a loop actually terminates at some end-of-file symbol.

Figure 1: Toolflow for loop classification.

Utilizing the LLVM analysis infrastructure, we detect all loops in the benchmarks and classify their trip count into the three categories: compile-time (CT), runtime (RT) and not computable (NC). Note, that this examination does not take into account the suitability of loops for vectorization or any other form of acceleration. It rather shows the overall opportunity of a runtime decision for real-world applications. The results are presented in Table 1. The ratio of compile-time, runtime and not computable trip counts for C/C++ code is somewhat

---

2http://llvm.org/docs/TestingGuide.html
3http://llvm.org/docs/Passes.html
To better understand this, Equation (4) to (6) denote the execution frequencies for the loops represented in Listing 1 and the corresponding trip counts as denoted in Equation (1) to (3). In order to decide if the code should be executed on the accelerator or on the host, we actually compare the execution frequency for nested loops and not just the plain trip count (as demonstrated later in our toolflow in Section 5).

\[
\begin{align*}
\text{exe}_1 &= \text{count}_1 \quad &\text{// Loop 1 (4)} \\
\text{exe}_2 &= \text{exe}_1 \times \text{count}_2 \quad &\text{// Loop 2 (5)} \\
\text{exe}_3 &= \text{exe}_2 \times \text{count}_3 \quad &\text{// Loop 3 (6)}
\end{align*}
\]

To validate the suitability of our runtime decision in real-world applications, we analyze the loop execution frequencies for the SPEC CPU2006, MiBench and MediaBench benchmarks. This enables us to validate if our static analysis holds when an application is actually executed. It also enables us to determine if the loop execution frequencies are sufficiently large for runtime decidable loops. As each benchmark consists of applications across different domains having contrasting behaviors, we chose to analyze the benchmarks at an application level.

To get the real loop execution counts, we first instrument and then run the benchmarks. We use LLVM to instrument the code. We first compile the C/C++ and F/F90 code into LLVM IR (as described in Section 3.2). After transforming the loops into their canonical form and performing basic code optimizations, we instrument all the loops with our custom LLVM instrumentation pass. All canonicalized loops in LLVM are guaranteed to have the following basic blocks with their corresponding properties (see example in Figure 2):

- **pre-header**: always executed before entering the loop
- **latch**: always executed before starting a new iteration
- **exit-blocks**: the loop is guaranteed to exit through one of the exit blocks

With our instrumentation, we create a global variable that holds the execution frequency for each loop. Code to initialize this variable is inserted into the loop **pre-header**, while the code to increment it by one is inserted into the loop **latch**. Finally, code that outputs the value of the variable to standard output is inserted into every loops **exit-block**. We need to do this, as a loop may have multiple exit-blocks depending on its exit conditions. In addition to outputting the value of the variable, we also output the loop type (CT=compile-time, RT=runtime or NC=not computable) along with the loop (pre-header) name and function name which are represented as comma separated values. Once the code has been instrumented, we follow the normal benchmarking compilation toolflow to obtain an executable. We then execute it and pipe the output to a trace file.

An excerpt from one such trace can be seen in Listing 3.

---

*Note: The MiBench applications office-ispell and telecom-adpcm are excluded because they are not working correctly. This issue is known and reported in 2012. Source: [http://lists.cs.uiuc.edu/pipermail/livm-dev/2012-December/056570.html](http://lists.cs.uiuc.edu/pipermail/livm-dev/2012-December/056570.html)*

---

In order to have a large data set, we use the train data-set for evaluating the SPEC benchmark. We collate the trace data into a condensed representation of the loop execution frequencies at an application and benchmark level, represented in Figure 3. The first column (**loop classification**) gives us the percentage of how many CT, RT and NC loops an application is comprised of, for the given execution instance (normalized to the number of executed loops). The second column (**loop execution frequency**) represents the execution frequencies for the different types of loops (normalized to the cumulative loop execution frequency). Cumulative loop execution frequency just reflects the fact that a loop (nest) can be called from different locations (i.e. from different functions), whereas the execution frequency from Equation 4 to 6 is basically the static multiplication of trip counts to improve our runtime decision. The last column represents the cumulative execution frequency of all the loops plotted on a logarithmic scale. This helps us determine the relevance/weight of the execution frequencies presented in the **loop classification** column, which is reflected in two ways in which we summarize the data. The **Overall Summation** section shows us the overall result that is summed up over the classified loops and summed frequencies of all the benchmarks, which after normalization effectively represents a weighted average with summarized loop counts and cumulative execution frequencies as respective weight factors. For the **Overall Average** section in contrast, we averaged the per benchmark fractions of each loop class, resulting in an unweighted average across all bench-
marks.

The Overall Summation section in the loop classification column in Figure 3 corresponds in summation method to the previous analysis in Table 1, however it contains only the loops that are actually executed with the benchmark input data sets, whereas the previous data contained all loops present in the application code, including loops that are never executed during benchmarking. Comparing the dynamic data to our static results, we see that for this particular input set, the percentage of runtime decidable loops for SPEC is around 28% as compared to the 41% in our static analysis in Table 1. MiBench also has fewer runtime decidable loops, only MediaBench has nearly the same number of runtime decidable loops as determined by static analysis. Apparently, some of the runtime decidable loops tend to reside in parts of applications, which are never executed during benchmarking, possibly code for error or exception handling, but maybe just additional functionality of the applications, which is not covered by the input data sets. Even though we see a lower overall percentage of runtime decidable loops (26%, see Figure 3) for this particular run as compared to our static analysis (40%, see Table 1), we believe that it is still significantly high to profit from our runtime approach. If we look at the overall average for runtime decidable loops, we also see a similar trend.

Investigating the same data in more detail, now at the application level, we see a lot of diversity. Applications like 410.bwaves, 416.games, 437.leslie, 459.gemsFDTD or telecom.fft have a very high percentage of runtime decidable loops making them prime candidates that could benefit from our runtime approach. On the other hand, applications like 470.ibm, network-dijkstra, office-stringsearch or g721 do not have any, or in fact have a very small percentage of runtime decidable loops and hence cannot benefit by using our runtime approach.

Apart from determining dynamic and per application data of loop classifications, we also investigated loop execution frequencies of the runtime decidable loops. Looking first at the Overall Summation section in the loop execution frequency column in Figure 3, we can see how much the runtime decidable loops contribute to the whole loop execution frequencies of the benchmarks. If we look at all the benchmarks together, we see that a large percentage (53%) of all loop execution frequencies belong to runtime decidable loops. High execution frequencies of the actually executed runtime decidable loops seem to make up for the runtime decidable loops that are not executed at all. However, we can see that this Overall Summation is dominated by SPEC with MiBench and MediaBench having a very low percentage of execution frequencies for their runtime decidable loops. As SPEC has a very high overall loop execution frequency (last column in Figure 3) we believe that it can benefit the most from our runtime approach. On the other hand if we look at the average over all benchmarks, we can see that the runtime decidable loops are responsible for around 30% of all the loop execution frequencies.

Finally, looking at the execution frequencies at application level, we see that the diversity trends observed when just classifying the loops appear to be even more prominent. For several applications with a high fraction of runtime decidable loops, including all of the previously mentioned, almost all of their cumulative execution frequency accounts to these runtime decidable loops. Applications without runtime decidable loops naturally preserve this property when analyzing execution frequencies, but for the applications with a small fraction of runtime decidable loops, their cumulative execution frequencies are sometimes higher (e.g. 400.perlbench), sometimes lower (e.g. 433.milc).

3.4. Overhead of Runtime Decisions

As presented in the last subsections, we saw that in order to be able to take offloading decisions at runtime, our LLVM pass modifies the application by injecting instructions before each applicable loop. In order to measure the overhead for executing this additional code, we consider the SPEC benchmarks. We use SPEC as it has the most number of loops (see Table 1) and is well supported by the LLVM test suite. We first compiled the SPEC benchmarks and executed them on the CPU. We then used our LLVM pass to inject the runtime code into the applications. However, we used a sufficiently high threshold such that the code is always executed on the CPU. In order to have a sufficiently large data set, we executed the benchmark 100 times for both the native as well as the modified test sets. On comparing the results, we found that the overhead for computing the trip counts and for a simple threshold comparison was negligible (less than 1%).

4. Moving Runtime Decisions

In the last section we presented our concept of runtime decisions and demonstrated for real-world applications that a considerably large amount of loops can benefit from this approach. For runtime (RT) dependent loops, all required information needs to be available at least right before the beginning of the loop (nest). However the insertion point of our runtime decision relies on control- and data-flow dependencies within the application. In many real-world applications, as demonstrated later in Section 4.2, this information is available many basic blocks ahead of the actual loop nest. Hence, the runtime decision for those cases can be performed at an earlier point of time, which opens an interesting vector of opportunities to improve the decision and offloading process. The idea can be compared to instruction scheduling, performed by compilers. Gibbons et al. [?] change the order of instructions to improve instruction-level parallelism, avoid pipeline stalls or hide latencies for memory operations. Panait et al. [?] reschedule instructions to prefetch future accelerator configurations. The motivation and demonstrated objective of their work was to reduce the number of reconfiguration steps, but along the way, this also overlaps the reconfiguration overhead with useful computation, which is the proposed benefit of the decision slack we investigate. Whereas our analysis focuses on intra-procedural dependencies, they proposed both, an inter- and intra-procedural static scheduling algorithm that tries to minimize the number of reconfigurations while respecting placement conflicts.
Figure 3: Loop classification and execution frequencies for different benchmarks and their applications.
Depending on the timespan that the offloading decision can be preponed, the target accelerator architecture and type of application, we propose different strategies to exploit the decision slack. In general the timespan can be used to check as early as possible the availability of the accelerator and if necessary to re-schedule assigned tasks. In a scenario with FPGA accelerators we see opportunities in starting the reconfiguration process to hide the offloading overheads. For a complete synthesis, the gained timespan is usually too small, but could be sufficiently large with an overlay architecture. For GPUs one could start the data migration process to reduce latencies or heat-up the GPU (increase clock frequency) to make the GPU attractive for small computing jobs. If the binary for a suitable accelerator is not available yet, one could also start just-in-time compilation to get optimized code for the loop (nest).

In the beginning we describe the moving of runtime decisions to an earlier point with the help of an example and define the required terminology. Afterwards we demonstrate that this new idea can actually be applied for real-world applications to improve the offloading process.

4.1. Decision Slack

The control flow graph (CFG) of an example application with one loop is schematically depicted in Figure 4. The loop (colored in blue) is separated into the loop pre-header and body (header, latch, exit-blocks, see Figure 2). In the pre-header, the trip count computation and runtime decision (green) is performed. Our novel idea is to try to move this part as far as possible to the top of the CFG (indicated by the green arrows), while respecting dependencies (in red) to create a timespan which we denote as decision slack to improve our offloading process.

The decision slack (see Figure 4) is basically the span or distance between the earliest possible evaluation of the trip count \( T_{\text{comp}} \) and the actual usage in the loop pre-header \( T_{\text{use}} \). In order to compute the earliest possible evaluation point \( T_{\text{comp}} \), we first save the insertion point where we would have originally inserted the decision. This typically is the first non-PHI and non-LandingPad instruction in the loop pre-header. The PHI instruction is used to select a value depending on the predecessor block (\( \varphi \) node) and the landingpad instruction is used for LLVM’s exception handling system. In order to determine how far back the decision can be moved, we need to analyze the Scalar Evolution (SCEV) expression and determine on what variables the earliest possible evaluation of the trip count \( T_{\text{comp}} \) it depends on. To do this, we walk through the SCEV expression and build a list of all the dependent variables. The dependent variable can either be a function argument or the result of an instruction. If it is a function argument, we can directly move the decision to the starting block of the function. On the other hand, if the dependent variable is an instruction, we can move our runtime decision right to the position after the instruction has been defined. This works because LLVM is based on the static single assignment form (SSA) \([?]\). In both cases, we store the potential insertion points. If the earliest possible evaluation of the trip count \( T_{\text{comp}} \) (of the corresponding SCEV expressions) has more than one dependent variable, we can only move the decision back to the point were all the dependencies are satisfied. After the insertion point is computed, it is possible to define the decision slack from a static (compile-time) or a dynamic (runtime) point of view.

**Static Decision Slack** The static decision slack is retrieved by code analysis. We propose to use the number of LLVM IR instructions or the number of basic blocks between \( T_{\text{comp}} \) and \( T_{\text{use}} \) to describe the static decision slack. As indicated in Figure 4 the path to the earliest possible trip count computation \( T_{\text{comp}} \) can be ambiguous. The path over the basic block \( BB_j \) increases the number of instructions/basic blocks but it is not guaranteed that this path is taken at runtime. Therefore we chose a conservative approach by selecting the shortest possible path from \( T_{\text{use}} \) to \( T_{\text{comp}} \) (indicated by the green arrows). This restriction reflects in fact the worst-case (minimal) number of instructions/basic blocks as the decision slack but suffices as a lower bound to prove the usefulness of our idea. To increase the accuracy of the decision slack for runtime decision, one could instead use the built-in branch prediction pass of LLVM and compute the expected slack. Additionally, the distance metric can be refined, for example by weighting each instruction by its type.

**Dynamic Decision Slack** For the dynamic decision slack we propose to consider the actual runtime (online) behavior of the application in terms of absolute time between \( T_{\text{use}} \) and \( T_{\text{comp}} \). Therefore the code can be instrumented with counters at compile-time and measured at runtime. This approach automatically considers the actual path taken in the CFG for a specific input because it relies on application runtime.

![Figure 4](image-url)

*Figure 4: Moving the runtime decision to the earliest possible position in the control flow graph (CFG) creates a decision slack allowing us to improve the offloading process.*
After defining the decision slack from a static and dynamic perspective we analyze the characteristics of the slack in real-world applications. We use the static decision slack in the next subsection to analyze real-world applications. The dynamic decision slack relies on specific input data and is very hard to measure for a very small decision slack. In particular, the instrumentation required to measure the decision slack can change the CPUs caching behavior and thus may introduce noise into the measured execution time, that might have a higher magnitude than the actual slacks to be measured.

4.2. Decision Slack in Benchmark Applications

In this subsection we present an analysis of the possible decision slack in real-world applications. Therefore we analyze runtime dependent loops of the same benchmark suites as presented in Section 3.2 (SPEC CPU 2006, MiBench and MediaBench) and try to move the runtime decision as far as possible upwards in the CFG. For moving the decision we still use the conservative approach (shortest path in the CFG) for calculating static decision slack as discussed in the previous subsection. For this analysis, we want to take a deeper look at the static decision slack in terms of average number of instructions and average number of basic blocks between $T_{use}$ and $T_{comp}$. The results are presented in Table 2.

First we like to look at how many runtime decisions can be moved at all (at least one instruction; third column of the table). For SPEC most of the loop decisions (88%) can be moved. The number is significantly less for MiBench (53%) and also smaller for MediaBench (66%). Next we evaluate the average decision slack in the last column of the table. Again SPEC has the highest values. On average the decision slack for SPEC is over 79 instructions and over 10 basic blocks. For MiBench and MediaBench this number is about three quarters smaller, roughly 22 instructions or about 3 basic blocks.

These results show that the decision slack actually appears in real-world applications and could be exploited to improve the overall performance of applications using hardware accelerators as an additional feature to enhance the runtime decision and offloading process. As discussed, the actual timespan available when moving offloading decisions up, additionally depends on the types of executed instructions and may also be much larger, depending on actual control flow. As preliminary assessment, the observed decision slacks seem suitable to check the availability of the accelerator and possibly call the scheduler to reassign tasks. It may also be used to heat-up a GPU (increase clock frequency) or to transfer some individual scalar inputs to a coprocessor. In the differential context of a just-in-time compilation and execution engine, some of the larger observed decision slacks may even be useful to start just-in-time compilation to generate optimized code of the loop (nest). On the other hand, the reconfiguration process of an FPGA or the transfer a considerable amount of data for data-dominated hotspots may be initiated earlier, but will not be completed within the observed decision slacks, let alone a full FPGA synthesis toolflow including placement and routing.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>number of RT loops</th>
<th>movable decisions</th>
<th>average decision slack in</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC CPU’06</td>
<td>30,430</td>
<td>26,711 (88%)</td>
<td>79.63</td>
</tr>
<tr>
<td>MiBench</td>
<td>349</td>
<td>185 (53%)</td>
<td>21.31</td>
</tr>
<tr>
<td>MediaBench I</td>
<td>186</td>
<td>122 (66%)</td>
<td>22.32</td>
</tr>
</tbody>
</table>

5. Case Study: Runtime Decisions for a Reconfigurable Vector Coprocessor

In this section, we present a case study that demonstrates how the runtime offloading decisions have been integrated into a toolflow that automatically partitions applications between a host CPU and a vector coprocessor implemented on the FPGA of a Convey HC-1 system.

The foundations of this toolflow have been laid in our earlier work [2], which focused on creating a fully automated, hardware/software partitioning toolflow and vectorizer for the Convey vector coprocessor. For this earlier toolflow, we investigated only offloading decisions for compile-time trip count loops and statically selected all loops for offloading that exceed a fixed iteration threshold. As shown in Sections 3.2 and 3.3, such static decisions are not sufficient in many real-world applications. In the remainder of this section we briefly describe the target architecture and toolflow and then present the integration of runtime decisions for offloading and data migration into the code generation backend. An evaluation of the effect of runtime decisions will be presented in Section 5.3.

5.1. Convey HC-1 Target Platform

Our target system is the Convey HC-1 hybrid-core platform [2], which is based on an x86 Intel Xeon CPU that is attached to a configurable FPGA based coprocessor via the Intel Frontside Bus (see Figure 5). A defining characteristic of this system is that both the CPU and the coprocessor have their own dedicated physical memory, which can be transparently accessed by one another through a common cache-coherent virtual address space. This system effectively forms a heterogeneous NUMA architecture, which offers, through the use of 8 parallel memory controllers, particularly high bandwidths of up to 80 GB/s between the coprocessor and its local memory.

The configurable coprocessor implements application-specific instructions denoted as personalities. A personality is a loadable bundle of optimized instructions to augment the x86-64 instruction set for different types of applications [2] and a corresponding FPGA configuration to implement this instruction set extension. In this work we target the Convey Vector Personality, which implements a floating-point vector coprocessor. Vector Personailties have been used in various related studies, i.e. by Meyer et al. [2] for stencil computation or Kenter et al. [2] for stereo matching.

Using this personality eases the programming effort considerably compared to other heterogeneous CPU-FPGA platforms, because both targets are software programmable. The toolflow proposed by Convey is to write or use existing C/C++ code,
augment it with compiler directives (pragmas) to advise the compiler about the code blocks to be vectorized and offloaded and the data structures involved and to compile this annotated code with the Convey vectorizing compiler. The result of the compilation is a dual target executable for the host CPU and the coprocessor.

Two of the studies using this Convey toolflow report problems with vectorization and efficient code generation for more complex loops and data structures [? ]. We tackled some of these issues and additionally removed the need for pragmas in an alternative toolflow, first presented in [? ]. In order to let this toolflow take qualified decisions in absence of both statically determinable trip counts and of pragmas, we integrated the runtime decisions presented here.

5.2. Compilation Toolflow

Our toolflow generates heterogeneous binaries that can be executed on the host CPU and the coprocessor. A schematic overview of our toolflow is depicted in Figure 6. The input to the toolflow is an LLVM IR (intermediate representation) binary of the application. The main component of our toolflow is the PartitionPass, which is implemented as an LLVM ModulePass that works on the LLVM IR and performs the following steps:

1. Identify basic blocks that are feasible for execution on the coprocessor (no system calls, recursive function calls, I/O, etc.).
2. Extract basic blocks that belong to loops into separate functions.
3. Determine if the loops can be vectorized (inner/outer loop vectorization).
4. Add a function call interface via which the coprocessor code can be called.
5. Partition the code into: coprocessor functions, modified host code functions, and host-coprocessor interface.

Once the application has been partitioned, the modified host functions are translated into x86 assembly code by using the standard LLVM backend (llc). For implementing the function call interfaces through which the host can call the offloaded code on the vector coprocessor, we use the Convey Compiler (cnyCC). The Convey Compiler generates the function signatures, sets up the corresponding stacks and inserts coprocessor availability checks. By using the Convey Compiler instead of LLVM for creating the function call interface, we can avoid reimplementing the calling conventions of Convey’s application binary interface. Hence, the Partition Pass generates a host-coprocessor interface which is a .cpp file that contains stubs with the appropriate signatures of all functions that can be run on the coprocessor. The Code Generation Pass generates vectorized coprocessor assembly code for the coprocessor functions based on the vectorization strategy that was previously determined by the Partition Pass. We then use a Python script that integrates our assembly code (generated by the Code Generation Pass) with the coprocessor (assembly) stubs that were generated by the Convey Compiler. We finally use the Convey Compiler tools to link all the generated assembly and object files to generate the heterogeneous binary. For a more detailed discussion of the toolflow and vectorization approach refer to [? ].

5.3. Runtime Decisions for Offloading and Data Migration

For integrating runtime decisions into the application, we have followed the methods presented in Section 3 and Section 4. Using LLVM’s ScalarEvolution information, we add code to the application that computes the trip count for all loops. To integrate this information into the application, we have extended the PartitionPass with a Runtime Code Injection module (see Figure 6) to modify the CPU functions that are suitable candidates for offloading by inserting runtime decision code before the entry block of the CPU function. To support dynamic runtime decisions in our toolflow, we add a flag to the corresponding function to indicate that a runtime-check is required and afterwards insert a new entry block to each flagged function.

For simple loops our runtime comparison is just a single compare instruction with the threshold as described above in Section 3.1. If we encounter a nested loop, we use a slightly advanced approach as shown in the example in Listing 2. Thus, whenever the function is called, the runtime-check evaluates the loop execution frequency and decides whether it is beneficial to run the code on the coprocessor. If the runtime-check decides that the code should run on the coprocessor, we then migrate the required data to the coprocessor, call the corresponding coprocessor function, and then migrate data back to the host.

Otherwise the execution is performed on the CPU. Figure 7 shows the control flow of the application with our runtime-check. Whenever coprocessor code is available for a part of the application, the runtime-check is performed to decide if the computation should be offloaded or not.

In addition to offloading decisions we also use the availability of information about the trip count of offloaded loop nests for optimizing the data transfer between CPU and coprocessor. From a functional point of view, explicit data migration is not required, because the Convey HC-1 provides a cache-coherent, shared memory between CPU and coprocessor memory. However, the memory subsystem of the HC-1 has a NUMA (non-uniform memory access) characteristic, that is, bandwidth and
latex

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{figure6.png}
\caption{Workflow for generating heterogeneous binaries (blue: our implementation; yellow: Convey Compiler infrastructure; green: LLVM infrastructure).}
\end{center}
\end{figure}

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{figure7.png}
\caption{Flow chart of the interaction between the Host CPU and Coprocessor for the runtime-checks.}
\end{center}
\end{figure}

\begin{lstlisting}[language=cpp]
1 // ### Migrate data to the coprocessor memory.
2 @cny_migrate_data(%aa, %size_aa, %coproc)
3 @cny_migrate_data(%a, %size_a, %coproc)
4 // ### Dispatch call to coprocessor.
5 %cny_call =
6 call @heavyFunc_cny(%N, %height, %width)
7 // ### Migrate data back to the host memory
8 @cny_migrate_data(%aa, %size_aa, %cpu)
9 @cny_migrate_data(%a, %size_a, %cpu)
10 // ### CPU code.
\end{lstlisting}

Listing 4: An example for the runtime data migration of the two dimensional arrays \textit{aa} and \textit{a}.

In this example, arrays \textit{aa} and \textit{a} were identified to be migrated during the static analysis of the \texttt{a2dBuffer\_cny} function.

In our approach, we make use of a buffer to store images. The size of the buffer is set to the maximum image resolution that we support (7680x4320). We do this as the image size is known only at runtime and we need to allocate a buffer that is large enough. In the case of smaller images, they only occupy a portion of the buffer. However, as the image resolutions can vary dramatically (e.g., DVD 720x480, Blu-Ray 1920x1080, and UHDTV 7680x4320), migrating the entire buffer to the coprocessor is very expensive. In our approach, we only migrate those portions of the buffer that are required by the coprocessor code. We determine the section of the buffer that has to be migrated at runtime by taking into account the maximum loop bounds within which the buffer is accessed. We demonstrate how the data size is computed with the help of a simple two dimensional array \texttt{aa[y][x]} (buffer) as shown in Figure 8. The array \texttt{aa} has dimensions of 7680x4320. Let us assume that the coprocessor function \texttt{heavyFunc\_cny} works on an image of dimensions 1920x1080. This only uses a subsection of the buffer i.e. elements \texttt{aa[0][0]} up to \texttt{aa[1079][1919]} as shown in the figure. By analyzing the code, we can determine this, and migrate only the required subsection of the buffer. Ideally we would only want to move those elements that are required, but the memory is organized in pages and during data migration, the entire memory pages are migrated. To avoid multiple transfers, we prefer to have one large structure as compared to several smaller ones and so we migrate a 7680x1080 subsection of the buffer. By using this approach, we are able to save the cost of migrating the entire buffer to and from the coprocessor memory, which in turn results in better overall performance.

\begin{figure}[h]
\begin{center}
\includegraphics[width=\textwidth]{figure8.png}
\caption{Flow chart of the interaction between the Host CPU and Coprocessor for the runtime-checks.}
\end{center}
\end{figure}

Latency of memory accesses are strongly dependent on whether the data is stored in the local memory attached of the computing resource.

To ensure that the required data is present in the local memory of the computing resource requesting it, Convey provides API calls that migrate the data to either the host memory or coprocessor memory (see Figure 5). Hence, the data is migrated from the host to the coprocessor memory before executing the coprocessor code, and once the execution on the coprocessor is complete, the data is migrated back to the host memory so that the remaining CPU code is not slowed down by having to read data from the coprocessor memory.

For optimizing these memory transfers, we analyze each function that can be offloaded to the coprocessor and determine what data will be required during its lifetime. For this purpose, we do not only consider statically available information but also trip count information that is known at runtime. The \texttt{Partition-Pass} then modifies the LLVM IR to encapsulate the call to the coprocessor with calls to Convey’s data migration API to first migrate the data to the coprocessor and then back to the host memory. An example for the modified LLVM IR can be seen in Listing 4. We use the \texttt{cny\_migrate\_data} function to migrate the data. The function parameters are; the starting address of the data to be migrated, its size, and where it needs to be migrated.
6. Evaluation

In this section we first evaluate the effectiveness of our data migration scheme and then evaluate our runtime approach with respect to the traditional partitioning approach. For our evaluation, we use our toolflow for the Convey HC-1 described in Section 5.2. We evaluate our approach by using real world image dimensions as inputs to our kernels. We have chosen the image dimensions based on popular image resolutions used in the real world.

6.1. Test Suite

To evaluate the impact that different kinds of compute patterns have on the system, we used a synthetic test suite of representative loops based on stereo matching algorithms. The test suite was first utilized in [?] to demonstrate partitioning and generation of vectorized code through our toolflow. For some insights into the computation steps and dependency patterns that inspired our test suite, we refer to [? ?]. For our test suite, we substituted loops with corresponding dependency patterns through representative, sometimes simplified loops, and then systematically varied the dependency patterns.

For example, the loop a3dInner mimics the dependency pattern of a scanline optimization step. A value inside a 3-dimensional cost volume a[d][y][x] depends on three inputs: its own previous value, a value along a column inside the volume a[d][y-1][x] and some difference in a 2-dimensional image b[y][x]. This enables vectorization of the inner volume dimension, iterated in the inner loop of a loop nest. For the loop a3dMiddle, the dependency is changed to the row dimension of the nest. The a3dInnerS and a3dMiddleS are simpler variants of the same pattern: the value from the 2-dimensional image is replaced by a scalar. The a2dInner and a2dOuter mimic the computation of integral sums over a 2-dimensional image, the a2dAggInner and a2dAggOuter select elements of these integral sums according to some region sizes depending on local color properties of the image at that position. Together these two pairs of loops represent the two steps of a cost aggregation phase in stereo-matching. The a2dCensus and a2dAD represent patterns of the two elements of a cost initialization step, census referring to a census transformation of the image running over small local windows around each pixel, AD referring to absolute color differences between a left and right image.

Of the latter, we added a variant computing an entire cost volume in one loop a3dADInner, allowing for better data reuse. For systematics, we also interchanged the dependency pattern in a3dADMiddle.

6.2. Data Migration

Our test suite functions make use of a buffer from which they read/write data during their computational lifecycle. The size of the buffer that is actually used is dependent on the image size. In the traditional approach, this size cannot be determined at compile-time as the function might operate on images of varying dimensions. Hence, in the traditional approach, the entire buffer needs to be migrated to the coprocessor memory.

The aim of our new data migration strategy is to improve the performance of the coprocessor code by determining the amount of data to be migrated to the coprocessor memory at runtime instead of migrating the entire buffer (7680x4320) to the coprocessor memory. In order to determine the amount of absolute time saved, we record the amount of time it takes to transfer data for different image sizes while using our new data migration strategy. Figure 9 depicts the absolute amount of time required for transferring only a part of the buffer for a given image dimension. The maximum time required to transfer the entire buffer (7680x4320) is represented by the bars on the extreme right. By comparing the times required to selectively transfer data for all other image dimensions with the maximum transfer time, we can see that we save a significant amount of time (especially for smaller images) by utilizing our new data transfer approach instead of transferring the entire buffer. This in turn helps boost the overall performance of the application.

To evaluate the benefits of our data migration strategy on the performance of the application, we first measure the time taken by the coprocessor (including data transfer times) to execute the test suite code for different data sizes. In this case, the entire buffer is migrated to the coprocessor. We then enable our runtime data migration module and record the execution time of...
the test suite. The speedup of our new data migration strategy is computed by taking the ratio of the coprocessor execution time when the entire buffer is transferred to that of the coprocessor execution time with our new data migration strategy.

In Figure 10, we can see that by using our on demand data migration strategy, we achieve up to 9x speedups for smaller image dimensions. We can also observe the trend that as the image dimensions get closer to that of the buffer size, the speedups seen are reduced as more and more data has to be migrated to the coprocessor in such cases. It is also interesting to note that the speedup is not only dependent on the amount of data migrated but also on the computation pattern. We see that our data migration strategy produces only a marginal performance improvement for \( a2dCensus \), which is computationally intensive with a high local data reuse and thus overcomes non-optimal data migration easily. The same trend with a lesser extent can be observed for \( a2dInner \) and \( a2dOuter \). Thus, by migrating data on demand based on the runtime values, we reduce the data migration overhead and functions that were previously not profitable to run on the coprocessor (because of large data migration overheads) can now be executed on the coprocessor.

6.3. Runtime System

In the traditional approach, once code has been vectorized and compiled for the coprocessor, it is always executed there. Our approach considers the size of the data to be processed at runtime and determines if it would be beneficial to run the code on the CPU or on the coprocessor. In order to evaluate this approach, we first measure the time taken by the coprocessor to execute the test suite code for different loop iterations. We then measure the execution time for our runtime approach. In order to evaluate the performance gain attributed only to the runtime system, the measurement is performed with the data migration module enabled in both cases. The speedup is computed by taking a ratio of the coprocessor execution time to the execution time using our runtime system. It is important to note that in our test suite, the loop iteration space is bound by the size of the image dimensions (see Figure 11). We also compute the speedup of our runtime system to that of the CPU in Figure 12.

Figure 11 shows the performance gain that we achieved by using the runtime system. We see that we can achieve speedups of up to 10x depending on the loop iteration space. The maximum gain is achieved for smaller image dimensions as they are run on the CPU instead of on the coprocessor. As the loop iteration space increases, the code is executed on the coprocessor (Figure 12) where we can achieve speedups of up to 3x by executing the code on the coprocessor instead of the CPU.

Table 3 shows us where a specific function is executed when our runtime system is used (gray cells). We can see that most of the functions are executed on the CPU for smaller loop iterations. And as the loop iteration size increases above a threshold where the data migration is amortized, the computation is performed on the coprocessor. The computation driven \( a2dCensus \) is always executed on the coprocessor as the data migration overheads can be amortized even for small loop iterations. On the other hand, the \( a2dAggInner \) and \( a2dAggOuter \) functions
are always run on the CPU as even though they can be vecto-
rized, it is faster to execute them on the CPU instead of the
coprocessor for the given range of loop iterations in our evalua-
tion. Thus our runtime approach benefits from both worlds and
is able to maximum the performance of the application.

<table>
<thead>
<tr>
<th>Function</th>
<th>Proc. Unit</th>
<th>DVD</th>
<th>HD</th>
<th>Blu-Ray</th>
<th>4K</th>
<th>UHD-TV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2dInner</td>
<td>CPU</td>
<td>0.03</td>
<td>0.07</td>
<td>0.25</td>
<td>1.10</td>
<td>3.89</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.06</td>
<td>0.13</td>
<td>0.19</td>
<td>0.46</td>
<td>1.23</td>
</tr>
<tr>
<td>a2dOuter</td>
<td>CPU</td>
<td>0.10</td>
<td>0.29</td>
<td>0.66</td>
<td>2.8</td>
<td>10.41</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.11</td>
<td>0.20</td>
<td>0.50</td>
<td>1.53</td>
<td>4.74</td>
</tr>
<tr>
<td>a3dInnerS</td>
<td>CPU</td>
<td>0.05</td>
<td>0.12</td>
<td>0.26</td>
<td>1.07</td>
<td>3.83</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.22</td>
<td>0.36</td>
<td>0.54</td>
<td>1.13</td>
<td>2.64</td>
</tr>
<tr>
<td>a3dMiddleS</td>
<td>CPU</td>
<td>0.12</td>
<td>0.29</td>
<td>0.66</td>
<td>2.79</td>
<td>10.38</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.27</td>
<td>0.43</td>
<td>0.84</td>
<td>2.22</td>
<td>6.12</td>
</tr>
<tr>
<td>a3dInner</td>
<td>CPU</td>
<td>0.06</td>
<td>0.18</td>
<td>0.40</td>
<td>1.66</td>
<td>5.96</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.25</td>
<td>0.42</td>
<td>0.64</td>
<td>1.33</td>
<td>2.94</td>
</tr>
<tr>
<td>a3dMiddle</td>
<td>CPU</td>
<td>0.11</td>
<td>0.30</td>
<td>0.68</td>
<td>2.87</td>
<td>10.62</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.30</td>
<td>0.49</td>
<td>0.94</td>
<td>2.46</td>
<td>6.68</td>
</tr>
<tr>
<td>a2dCensus</td>
<td>CPU</td>
<td>3.89</td>
<td>10.61</td>
<td>24.14</td>
<td>104.30</td>
<td>393.21</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>2.65</td>
<td>7.97</td>
<td>12.44</td>
<td>49.96</td>
<td>198.65</td>
</tr>
<tr>
<td>a2dAggInner</td>
<td>CPU</td>
<td>0.02</td>
<td>0.05</td>
<td>0.12</td>
<td>0.48</td>
<td>1.71</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.13</td>
<td>0.21</td>
<td>0.33</td>
<td>0.73</td>
<td>1.83</td>
</tr>
<tr>
<td>a2dAggOuter</td>
<td>CPU</td>
<td>0.02</td>
<td>0.05</td>
<td>0.11</td>
<td>0.48</td>
<td>1.69</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.14</td>
<td>0.21</td>
<td>0.32</td>
<td>0.72</td>
<td>1.74</td>
</tr>
<tr>
<td>a2dAD</td>
<td>CPU</td>
<td>0.01</td>
<td>0.04</td>
<td>0.08</td>
<td>0.33</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.10</td>
<td>0.13</td>
<td>0.20</td>
<td>0.41</td>
<td>0.86</td>
</tr>
<tr>
<td>a3dADInner</td>
<td>CPU</td>
<td>0.08</td>
<td>0.25</td>
<td>0.56</td>
<td>2.3</td>
<td>8.2</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.29</td>
<td>0.48</td>
<td>0.71</td>
<td>1.48</td>
<td>3.3</td>
</tr>
<tr>
<td>a3dADMiddle</td>
<td>CPU</td>
<td>0.08</td>
<td>0.25</td>
<td>0.56</td>
<td>2.32</td>
<td>8.25</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>0.29</td>
<td>0.48</td>
<td>0.72</td>
<td>1.48</td>
<td>3.42</td>
</tr>
</tbody>
</table>

7. Conclusion

An automated runtime system for heterogeneous systems has
many advantages over traditional static SW/HW partitioning
systems. In this paper, we have presented our approach of deferring
offloading decisions to application runtime, where dynamic
information about execution is available (trip counts, execution
frequencies, input data-sizes). To achieve that we leveraged the
LLVM compiler infrastructure to automatically analyze and in-
strument an application without user interferences. This way,
our runtime decisions are directly integrated into the applica-
tion code and come with minimal overheads even when em-
ployed extensively, i.e. in all loops that support this pattern.

We have demonstrated the potential of this new idea with
regard to real-world applications by analyzing common bench-
mark suites and have seen that some statically detected loops for
runtime decisions are not actually executed during our bench-
mark runs, but that those which are executed contribute a large
share of actually measured execution frequencies. We outlined
that moving the runtime decision to earlier positions in the pro-
gram flow can additionally open an interesting vector of oppor-
tunities to improve the decision and offloading process.

Afterwards we have presented our runtime system that can
decide to execute application code on the CPU or on an FPGA
based coprocessor depending on a threshold that can be derived
from the behavior of the application. In a case-study we show
that our runtime approach is superior to static approaches that
either execute code only on the CPU or on an FPGA. By
moving the decision of where the code should be executed from
compile-time to runtime, we achieve speedups that were pre-
viously not achievable by using the traditional approach. We
show that by using our runtime system along with our adapted
data migration strategy, a speedup of up to 3x compared to
“CPU only” and up to 10x compared to “coprocessor only” can
be achieved.

In this paper, we focus on the performance aspect. How-
ever, we also believe that, depending on the heterogeneous tar-
get platform, this approach could result in significant energy
savings and we will explore this aspect in our future work.
In this regard, we plan to extend our approach to other heteroge-
neous and FPGA architectures. When the static analysis can
additionally provide estimates about the suitability of different
target coprocessors, runtime decisions can become even more
interesting, but the overheads might increase and thus need to
be kept track of.

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

Vitae

Gavin Vaz received his Masters Degree in Computer Science
from Paderborn University and is currently working as a re-
search associate at the Paderborn Center for Parallel Comput-
ing. His main research interests include compilation techniques
for heterogeneous systems, just-in-time compilation techniques
and high-performance computing.

Heinrich Riebler works as a research associate in the High-
Performance IT Systems group at the Paderborn University. He
received his M.Sc. in Computer Science at the same university.
His research interests include heterogeneous high-performance
computing, compilers and dynamic runtime systems.

Tobias Kenter received his Diploma in Computer Science from
Paderborn University, and now also works as research associate
in the High-Performance IT Systems group. His research inter-
ests include compilation techniques and overlay architectures
for FPGAs, as well as system architectures with FPGAs.

Christian Plessl is a professor for High-Performance IT Sys-
tems at Paderborn University. He earned a PhD and a M.Sc. in
Computer Engineering from ETH Zurich. His research inter-
ests include architecture and tools for high-performance paral-
lel and reconfigurable computing, hardware-software codesign,
and adaptive computing systems.