Determining Performance Boundaries on High-Level System Specifications

Wouter van Teijlingen &
Rene van Leuken,
Delft University of Technology,
Circuits and Systems
Mekelweg 4
Delft, Netherlands
wouter@van-teijlingen.nl
t.g.r.m.vanLeuken@tudelft.nl

Carlo Galuzzi
Maastricht University,
Department of Data Science
and Knowledge Engineering
Bouillonstraat 8–11
Maastricht, Netherlands
c.galuzzi@maastrichtuniversity.nl

Bart Kienhuis
Leiden University, Leiden
Institute of Advanced
Computer Science
Niels Bohrweg 1
Leiden, Netherlands
a.c.j.kienhuis@liacs.leidenuniv.nl

ABSTRACT

We can significantly reduce the time required to realize designs if it is possible to find limits to the performance of an embedded system, solely based on high-level system specifications. For that purpose, we present in this paper the `cprof` profiler, which determines the number of clock cycles needed to execute a C-program in hardware. The `cprof` tool is based on the Clang compiler frontend to parse C-programs and to produce instrumented source code for the profiling. Using `cprof`, we determine a lower and upper bound limit for all 29 cases of the PolyBench/C benchmark suite. The lower and upper bound are determined using the absolute performance estimations assuming all statement are mapped onto the same processing resource and unbounded performance estimations assuming unlimited resources. We also compared the clock cycles found by `cprof` with RTL implementations for all 29 Polybench/C cases and found that `cprof` determines with 1.2% accuracy the correct number of clock cycles. It does this in a fraction of the time compared to the time needed to do a full RTL simulation.

1. INTRODUCTION

Engineers are dealing with the ever-increasing time-to-market pressure and demanding design constraints. We can significantly reduce the time required to realize designs if it is possible to find limits to the performance of an embedded system, solely based on the high-level system specification. This would bring a reduction in risk as the system designer knows very early whether a design meets its specifications. This concept is shown in Figure 1. It shows that for a particular design, it is possible to find a lower bound and an upper bound of the performance, where performance is expressed in the run-time measured to accomplish a given task. The lower bound (i.e., *Absolute performance estimate* in Fig. 1) represents the performance of the design without any specific optimizations, whereas the upper bound (i.e., *Unbounded performance estimate* in Fig. 1) represents the performance of the design assuming an infinite amount of resources, which allow to run at maximum parallelism.

SCOPES '16, May 23-25, 2016, Sankt Goar, Germany
© 2016 ACM. ISBN 978-1-4503-4320-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2906363.2906386

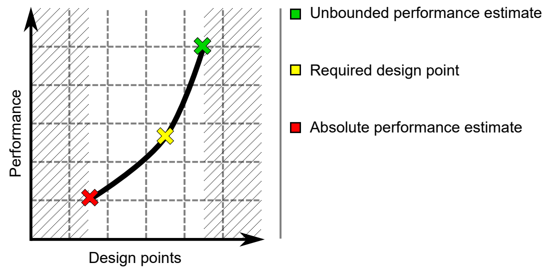

Figure 1: Exploring the design space using the estimated lower and upper bound of the design specification.

The shaded area indicates performance that is unattainable for the particular design under investigation. Most likely, the two design extremes are not the aspired performance. That will be somewhere in between these two points. By tuning the amount of parallelism that is exposed in the high-level specification, a designer can establish the curve between the two extremes and, therefore, select the required design point.

In this paper, we assume a high-level specification in C-code that is, eventually, implemented in hardware as a Polyhedral Process Network (PPN) [15]. An example of such high-level C-code is given in Listing 1.

```
for (i = 0; i < _PB_NY; i++)
    y[i] = 0;
    for (i = 0; i < _PB_NX; i++)
    {
      tmp[i] = 0;
      for (j = 0; j < _PB_NY; j++)
        tmp[i] = tmp[i] + A[i][j] * x[j];
      for (j = 0; j < _PB_NY; j++)
        y[j] = y[j] + A[i][j] * tmp[i];
    }
}
```

Listing 1: Matrix Transpose and Vector Multiplication Program (Atax) from the Polybench Benchmark.

It describes a matrix transpose and vector multiplication program (Atax) from the polybench suite [9]. For this C-code specification, we want to find quickly the two extreme design limits. For that purpose, we present in this paper the *cprof profiler*, which determines the design limits when executing the C-program in hardware. Profiling is a technique that determines the performance of programs during run-time and is many times faster then doing actual synthesis. Instead of waiting minutes or hours to get an idea of the performance of the C-program, we get it's limits in seconds.

The main contributions of the work presented in this paper are:

- the introduction of the profiler `cprof`;
- the implementation of `cprof` in Clang/LLVM;
- the validation of the results against hardware implementations of PolyBench/C benchmarks.

The remainder of the paper is organized as follows. In Section 2, we present the motivation for this work. In Section 3, we give an overview on profiling and the limitations. In Section 4, the solution approach used in `cprof` is discussed. In Section 5, we discuss the design and implementation of `cprof`. We show the results produced by `cprof` for the Polybench benchmark cases in Section 6. Finally, we present our conclusions and future work.

2. MOTIVATION

The main drive for this work is to boost engineering productivity, and to provide performance insights as early as possible to reduce design risk. Specifying designs at the Register-Transfer Level (RTL) is time-consuming and error-prone. Instead, a higher level of abstraction is used to simplify the design process. This is called High-Level Synthesis (HLS). HLS leads to the design flow shown in Figure 2. The figure shows that C-code is converted into a system-level specification that, via synthesis, is mapped either on a Field-Programmable Gate Array (FPGA) or on another platform.

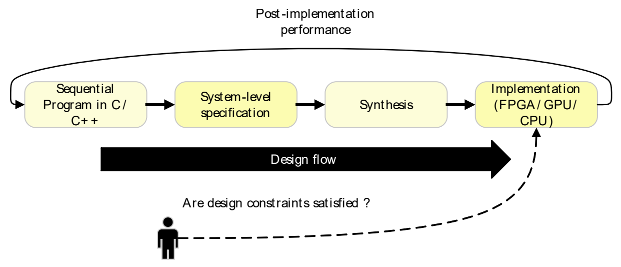
Post-implementation performance
Sequential Program in C/ C++
System-level specification
Synthesis
Implementation (FPGA/ GPU/ CPU)
Design flow
Are design constraints satisfied ?

Figure 2: Traditional design flow in high-level synthesis.

A designer is not interested in converting C-code into hardware, but he/she is interested in a system that meets the specific design constraints. As an example, let us suppose a designer needs to process 25 frames per second (FPS) for a video application. The problem with the traditional design flow, as shown in Figure 2, is that it takes long time before the designer knows if the design meets the constraints. Furthermore, if the constraints are not met, the designer needs to know if it is actually possible to meet the constraints with the given code. If so, how should the code be modified, so that the system can process the 25 FPS, for example, on an FPGA-board?

Therefore, we want to enhance the design flow by introducing profiling. Profiling is a technique used to analyze the behavior of high-level programs during their execution. This technique helps reducing the iterations, as depicted in Figure 3. In the modified design flow, a designer uses profiling to establish the design limits. This provides an immediate feedback on whether the design can satisfy the constraints at all. If 25 FPS are needed, but the upper bound turns out to be at 20 FPS, it means the design would never satisfy the constraints as 25 FPS is within the shaded area. If the upper bound is at 40 FPS, and the lower bound is 10 FPS, the designer knows a design of 25 FPS is indeed possible.

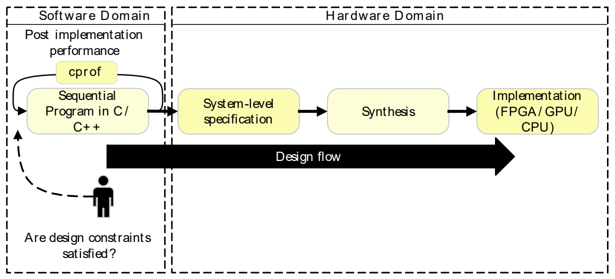
Software Domain
Hardware Domain
Post implementation performance
cprof
Sequential Program in C/ C++
System-level specification
Synthesis
Implementation (FPGA/ GPU/ CPU)
Design flow
Are design constraints satisfied?

Figure 3: Reducing the feedback loop in the design flow in high-level synthesis.

The next step would be to modify the code that results in 10 FPS to increase the parallelism in the application, until the design is capable of processing 25 FPS. These modifications can be realized via techniques like loop-unrolling, retiming, and plane-cutting as described, for example, in [11]. In general a large body of work exists about code optimization to increase parallelism. When the design is capable of processing 25 FPS, the designer commits to the very time consuming design flow to make implementable hardware. Now it is known that the design in hardware will satisfy the constraints. Notice that up till this point, no specific hardware knowledge is needed to actually realize the designs. As a result, by using `cprof`, a software designer can make design changes in the code and can estimate its performance. Once a particular design point is found, the design is committed to a hardware flow. Only at that time, specific hardware knowledge is required.

The flow presented in Figure 3 is quite general. In this paper, we use the Compaan design flow [6, 12] for the System-level Specification and use the Xilinx design tools for the synthesis for the FPGA implementation. This means that the C-code specification that we accept are so-call Static Affine Nested Loop programs (SNALP).

3. RELATED WORK

Different kinds of general-purpose profilers exist for the estimation of dynamic performance of computer programs. A well-known profiler, developed in the eighties, is gprof [3]. Gprof is used on regular basis by software engineers to collect performance metrics of computer programs. However, decisions based on general software profilers, like gprof, may give directions that are not applicable to hardware implementations. For example, let us consider the time required to finish the execution of a computer program. The sampling rate affects the accuracy of performance measurements, making it an inexact method for performance evaluation. As a result, tools, such as gprof, are not a reliable mean for estimating performance of high-level system specifications that are eventually implemented in, for example, FPGAs.

Critical Path Analysis (CPA) is a technique used for identifying parallel code regions in a program [16, 17]. It is also used to model the synchronization and communication dependencies between processes in a program. In [8], Kumar proposed the tool COMET, a tool for measuring the degree of parallelism found in Fortran programs by applying CPA. The assumption is that the program is executed on an ideal machine with an unbounded number of resources. Additionally, the ideal machine has no synchronization, communication, and no scheduling issues. After instrumenting the source code, COMET dynamically collects data to determine the absolute amount of parallelism.

Kremlin [2, 5] is a profiling tool similar to gprof. However, it is designed to discover parallelism in sequential programs. Kremlin uses a technique called Hierarchical Critical Path Analysis (HCPA), which adds hierarchy to critical path analysis. By using CPA, parallelism is measured within the complete program, whereas HCPA considers separate program regions. However, there are certain limitations to the approach taken by Kremlin. It is a tool targeting specifications written for general-purpose processors, whereas `cprof` targets specifications that are eventually implemented as PPNs in hardware. Moreover, it is not possible to automatically transform the code in such a way that the theoretical maximum degree of parallelism can be achieved.

To ameliorate the design flow in Fig 2, Haastregt [4] was looking at techniques to get quick feedback about the performance of PPNs. He looked at simulation in RTL and SystemC, at analytical models and profiling. RTL simulation is often not attractive or feasible due the amount of time and detail required to obtain a performance estimate for a given system. SystemC simulation yields accurate results in significantly less time but still requires a lot of detail similar to RTL simulation. He looked at an analytic approach for PPNs that is based on Maximum Cycle Mean (MCM) analysis, which is an established technique to assess the throughput of a Homogeneous Synchronous Data Flow (HSDF) graph [10]. The MCM method is able to deliver accurate results for a subset of PPNs, but could not define tight bounds on the inaccuracy of the MCM method, nor whether the method overestimates or underestimates the actual throughput. This model was theoretically attractive as it gives insight in the behavior of a PPN, but is impractical because of the lack of accuracy bounds. Finally, he found a profiling technique that works at the sequential code level that provides a fast, robust, and scalable performance assessment method. He transformed Kumar's approach [8] into an idea called `cprof`, which gave by far the best trade off in evaluation speed and accuracy. The original `cprof` tool was only a proof-of-concept; it was not capable of profiling any real-world applications. It had no support for complex data structures and all data was kept on the stack, thereby limiting the number of processes that can be profiled.

4. SOLUTION APPROACH

The goal of `cprof` is to determine the performance expressed in clock cycles of applications specified in the C programming language when implemented as a Polyhedral Process Network (PPN) in hardware. PPNs can be obtained automatically by the Compaan compiler for a special type of programs, called Static Affine Nested Loop Programs (SANLP). SANLPs are used, for example, for modeling time critical parts of audio/video stream-based and DSP applications. All the programs of the Polybench Benchmark are SANLP programs.

```
1  for (i = 0; i < _PB_NY; i++) {
2    proc1( &y[i] );
3  }
4  for (i = 0; i < _PB_NX; i++) {
5    proc2( &tmp[i] );
6      for (j = 0; j < _PB_NY; j++) {
7          proc3( &tmp[i],tmp[i],A[i][j],x[j] );
8      }
9      for (j = 0; j < _PB_NY; j++) {
10       proc4( &y[j],y[j],A[i][j],tmp[i] );
11     }
12 }
```

Listing 2: Outlined code for program Atax.

The performance we determine in `cprof` relates to the clock cycles of a given application in hardware, measured on a global time scale. `Cprof` models the scheduling of operations as soon as possible (ASAP) under particular resource constraints. In this section, we introduce the relevant concepts and properties related to the profiling of PPNs.

In the whole discussion, we follow the basic principle used by Compaan to model PPNs: each statement is mapped to one process. This means that all operations need to be wrapped into a single function call. This operation is called *outlining*, a step that can be done automatically by a compiler. In Listing 2, we show the Atax code of Listing 1 with each statement outlined in a function call. The code in line 9 in Listing 1 is for example outlined in `proc4` in line 10 of Listing 2. From the outlined version of Atax, the PPN shown in Figure 4 is derived. The figure shows that `proc3` and `proc4` are mapped to process P1 and P2. The functions `proc1` and `proc2` are mapped on the constant zero. The functions `proc0` and `proc5` are not shown in the outlined code. They are the result from simple intialization and data collection functions not shown in the outline code. The relationships between variables are mapped to edges. For example, the variable `tmp[i]` from `proc3` to `proc4` becomes an edge between process P1 and P2.

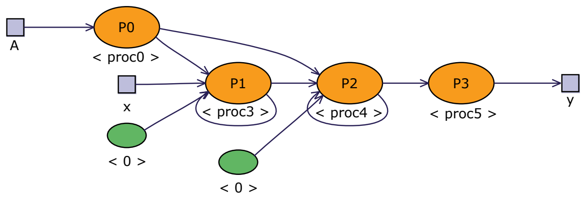

Figure 4: The Process Network for the outlined version of Atax.

From the work of van Gemund [14], we know that the performance of *any* parallel system is determined by only four properties: **basic calibration** (Section 4.1), **conditional synchronization** (Section 4.2), **conditional control flow** (Section 4.3), and **mutual exclusion** (Section 4.4). To understand these four properties, we look at each of them in the context of the program listed in Listing 2. The characteristics and definitions related to performance estimation are discussed in Section 4.5.

4.1 Basic Calibration

Each function in the outlined Atax C-programs has eventually to be implemented in hardware. If we look closely at `proc4`, we see that it describes a function with 3 inputs (tmp1, tmp2, and tmp3) and 1 output tmp0.

```
void proc4(double * tmp0, double tmp1,
           double   tmp2, double tmp3)
{
    (*tmp0) = (tmp1) + (tmp2) * (tmp3);
}
```

Listing 3: outlined function proc4.

It turns out that such a function can be modeled using only two parameters: an *initiation interval* (II_F) and an *function latency* (Λ_F). These two parameters are enough to calibrate a function in the C-programs we consider. By running the function `proc4` through the Vivado HLS tool from Xilinx, we obtain the two parameters to characterize function `proc4` as shown in Listing 4. The HLS tool indicates it can implement the function `proc4` with a 18 stage pipeline (Depth) and an initiation interval (II) of 1 at 200Mhz on a Xilinx Virtex 7.

In a PPN, each IP core expressing the actual computation is embedded in the LAURA Processor model [18]. In this processor model, there are three stages: read, execute, and write as shown in Figure 5.

```
@I [HLS−10] Starting hardware synthesis ...
@I [HLS−10] Synthesizing 'proc4' ...
@I [HLS−10] ————————————————————————
@I [HLS−10] —— Scheduling module 'proc4'
@I [HLS−10] ————————————————————————
@I [SCHED−11] Starting scheduling ...
@I [SCHED−61] Pipelining function 'proc4'.
@I [SCHED−61] Pipelining result:
              Target II: 1, Final II: 1, Depth: 18.
@I [SCHED−11] Finished scheduling.
```

Listing 4: Basic Calibration Parameters for function proc4 using Vivado HLS 2015.4 from Xilinx.

The read and write steps take care of the distribution of data in a process as a result of the data parallelization of the C-code using data-flow analysis. The execute stage integrates the actual pipelined IP block representing the function (e.g., `proc4`) in the C-code.

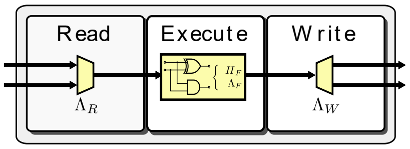

Figure 5: The Read, Write, and Execute units in a polyhedral process.

The use of the LAURA processor leads to the introduction of the read latency (Λ_R) and the write latency (Λ_W) that are relevant to the implementation of a process into hardware, where $\Lambda_R \in \mathbb{N}^+$ is the latency in clock cycles for reading input tokens, and $\Lambda_W \in \mathbb{N}^+$ is the latency in clock cycles for writing output tokens, respectively. In this paper, we assume $\Lambda_R = \Lambda_W = 1$, as all input and output tokens are read or written in one cycle to or from a Xilinx FIFO.

4.2 Conditional Synchronization

Within a PPN, a function executes only when all its data arguments are present, as happens in sequential C-code. Consider the function `proc4` in the outlined version of the Atax program. This function consumes the variables `y[j]`, `A[i][j]`, and `tmp[i]`. The execution of the function `proc4` is allowed only if all three arguments are satisfied.

Now, let us assume that variable `y[0]` and `tmp[0]` are available at time $t_{y[0]} = t_{tmp[0]} = 3$, and variable `A[0][0]` is available at time $t_{A[0][0]} = 4$, in the first iteration of the loop in line 10 of the outlined version of Atax. Even though `y[0]` and `tmp[0]` are available at $t = 3$, the function has to wait until `A[0][0]` becomes available at $t = 4$. This waiting affects the performance of the system and represents the so called *conditional synchronization* delay.

4.3 Conditional Control Flow

In a regular program, the sequence through the code is influenced by the conditional control flow. Dynamic statements, like `(i > N)` and `(a[i] < 3)` result in other sequences through the code. However, the programs that we are interested in, are specified as a SANLP. In that case, the loop bounds and conditional predicates are affine functions of enclosing loop indices and parameters. As a result, a SANLP has only static control parts. There is always a single-entry, single-exit region and conditional control flow has no effect on the performance.

4.4 Mutual Exclusion

Mutual exclusion concerns two or more tasks that are not allowed to execute their critical sections simultaneously. The critical section of a task accesses a shared resource, which can be used only by one task at a time. In a PPN no sharing takes place between processes. For example, the addition used in line 7 and line 9 in the orignal Atax program cannot be shared in a hardware implementation.

There is, however, an important form of mutual exclusion present inside each process. Inside a process, either each iteration is executed one after the other or all iterations can execute as soon as possible if data is available as each iteration has its own processor. This leads to two different performance estimations: *absolute performance* estimations and *unbounded performance* estimations. The absolute performance assumes that all iterations of a statement are mapped onto the same processing resource. Formally, it is defined as follows:

Definition 1. The absolute performance gives the number of clock cycles assuming that all iterations of a statement are mapped onto the same processing resource. Additionally, it considers an unbounded number of hardware resources.

The unbounded performance assumes an unbounded number of processing resources. That is, each execution of an iteration of a statement is mapped onto its own dedicated processing resource. Formally, we define unbounded performance as follows.

Definition 2. The unbounded performance gives the number if clock cycles assuming that the execution of an iteration of a statement is mapped onto its own dedicated processing resource. Additionally, it considers an unbounded number of hardware resources.

4.5 Performance Estimation

From the analysis of the four properties (See Section 4.1-4.4) that determine the performance of any parallel system, it is clear that we can determine the basic calibration. Condition control does not happen as we consider static programs. This means we still need to model the conditional synchronization and the two types of mutual exclusivity. To capture the conditional synchronization, we introduce the concept of the shadow variable and, to capture the mutual exclusivity, we introduce the concept of the control variable, as both expressed by Kumar [8] and later reconsidered by Haastregt [4] for profiling of PPNs.

To keep track of when a variable is written, we use *shadow variables*. The shadow variables keep track of the point in time in which a variable `$v` is written. The time stamp written to the shadow variable `$v` is the time of the write operations, including the cost of a write operation Λ_W. Shadow variables are used to model the conditional synchronization in PPNs. `Cprof` incorporates the conditional synchronization aspect by performing a `max` on the time stamps of all shadow variables of all input arguments.

Control variables, denoted as `C$`, model the moment a single processor can execute in time. A control variable stores a time stamp at which it can execute. A control variable is constructed for each statement and, thus, processes and uses the initiation interval parameter (II_F) of the basic calibration model to determine the next moment it can accept a new execution.

A control variable `C$` is updated in the read stage of a process. For all variables of the function associated with the process, the maximum time stamp is taken as a function can only execute when all input data is present. If this maximum time stamp is larger then the time stamp variable `C$`, the process is able to execute at the maximum time stamp. The next execution moment of the process is calculated based on the maximum time stamp value and the initiation interval parameter. If the maximum time stamp of the incoming data is smaller then the time stamp variable `C$`, the reading of incoming data is delayed until the time stamp value of `C$`. This explains how the absolute performance is calculated using a control variable.

For execution on an ideal machine, we assume an unbounded number of processing resources. Given an unbounded number of processing elements, there is no need to determine whether a resource is available. In this situation, only the availability of data is considered. It suffices to take the maximum over all shadow variables. This models the maximum parallel execution of a program as expressed by the unbounded performance.

4.5.1 Statement Execution Profile

To calculate meaningful performance metrics, we model each statement `s` with three one-dimensional arrays `R$s`, `E$s`, and `W$s`. These three arrays make up the statement execution profile, which models the behavior of the read, execute and write stages of a LAURA processor. For example, if `R$s[16] = 1`, it means that at time 16 there is one read operation active. Each array is initialized with zeros. If a read operation happens at statement `s`, the value `R$s` is incremented at index t_s. The variable `E$s` attains as maximum value the function latency (Λ_F) value to represent the pipeline depth. For example, if a function is implemented with a Function Latency of 18, it means that if the pipeline is utilized fully, the function runs 18 execute operations in parallel.

By using the variable `E$` of each statement, we can determine the global execution profile `G$` [4], as shown in Equation 1:

$$\texttt{GE\$[k]} = \sum_{i=0}^{|P|-1} \texttt{E\$}i\texttt{[k]} \tag{1}$$

$$0 \leq \texttt{k} < \max_{p \in P}\{f(p)\},$$

where P is the set of all processes in a PPN, and k is the maximum index of all statement profiles. As a result, the global execution profile describes the complete behavior of the process network in terms of operations. The global execution profile thus expresses how many operations happen at the same time in a network. The global execution profile is used to determine the average and maximum degree of parallelism, defined as follows.

Definition 3. The average degree of parallelism expressed in operations is the sum of all execute operations in the global execution profile, divided by the number of execute operations in the global execution profile.

Definition 4. The maximum degree of parallelism expressed in operations is the maximum number of simultaneously active execute operations in the global execution profile, at any given time t.

4.5.2 Flow Dependencies

To obtain a PPN from C-code, the Compaan compiler performs a sophisticated dataflow analysis to extract parallelism. The `cprof` profiler does not perform such analysis, which makes the profiler very robust and fast. Still, the input/output behavior of `cprof` is consistent as the profiler satisfies all data-dependencies using the shadow variable. In a sequential C-programs, three types of data dependencies exist: Read After Write (RAW), Write After Read (WAR), and Write After Write (WAW). These dependencies need to be respected inside `cprof` when determining the performance of a program. The flow dependency (RAW) is the most important dependency as it is used to build up a PPN. In `cprof`, a read operation takes the availability of a variable into account by checking the shadow variable for the last write time. Therefore, a read operation cannot start before the write operation has finished and, as a result, this dependency is accurately modeled. Anti-dependencies (WAR) in SANLPs do not affect the performance of PPNs. A write operation will simply lead to a new time stamp in the shadow variable, but this value is never used as no read occurs on the variable. An output dependency (WAW) also leads to the correct behavior. If a write operation happens after a write, the shadow variable takes the time stamp value of the last write operation.

Each dependency leads to an edge in a PPN. These edges need to be sized such that no deadlock can occur. Inside `cprof`, however, the assumption is made that edges have infinite space and that thus a blocking write can never happen. A write can always occur at a specific time stamp.

5. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of `cprof`, based on the concepts and solutions discussed in the previous sections. Figure 6 shows that the `cprof` profiler is divided into five stages that are discussed next.

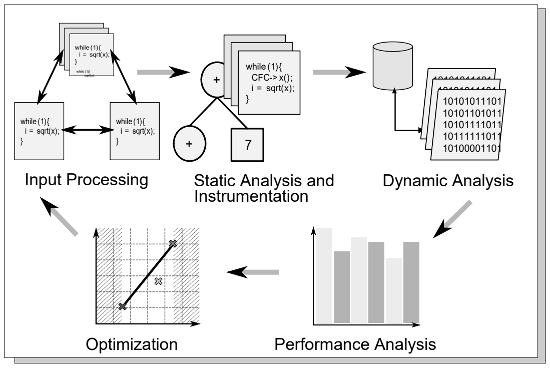

Figure 6: The architectural overview of cprof.

The first stage is responsible for **input processing** and the second stage deals with **static analysis** and **code instrumentation**. Static analysis identifies relevant code regions in the source code represented by an Abstract Syntax Tree (AST). A database is constructed with information about variables, function calls, and code structure in the program. Code instrumentation inserts specific statements into the abstract syntax tree. The result of the first two stages is an instrumented C++ program. This C++ program is compiled by a regular C-compiler into an executable. Running this executable leads to the **dynamic analysis** stage. Dynamic analysis profiles the instrumented program during run-time. Generated data is processed during the **performance analysis** stage and the total execution time is determined. Also, the average and maximum degrees of parallelism are calculated as well as pipeline efficiency. A final program profile is generated, which may be used for optimizing the source code.

5.1 Input Processing

For the input processing stage, we use the Clang compiler front-end [7]. Clang provides accessible libraries for building and modifying the AST. The input processing stage supports a subset of C programs. Programs must be valid SANLPs and the statements should satisfy the following:

- each computational statement is modeled as a function call;

- write arguments are passed as pointers to the function;
- read arguments are passed by value to the function;
- it is valid to have a single write argument as the left-hand side of the assignment operator, as long as the right-hand side is a function call;
- the **int**, **float**, **double**, **long** and **struct** (data) types are supported;
- use braces '{', and '}'for all `if` and `for` statements.

5.2 Static Analysis and Instrumentation

The static analysis and instrumentation stage identifies and collects relevant code regions in a C-program. The AST constructed by the Clang infrastructure is consumed by the `CprofASTConsumer`. This object is responsible for identifying the relevant code regions that make up a kernel. This kernel is represented by a `CprofManager`. For each kernel, the `CprofAnnotator` inserts initialization code to make dynamic analysis possible. The code for initialization is always placed above all existing statements in the kernel to assure that subsequent statements use initialized objects. Finally, the annotated AST is written out again as a C++ program. A snippet of the annotated program for the outlined Atax program is given in Listing 5.

```
  /* Initialization */
CprofSerializer* CS =
          CprofSerializer::getInstance("db");
CprofManager* CM = CS.get(0);
...
CprofFunctionCall* CFC4 = CM->get(4);
CFC4->setFunctionLatency(18);
CFC4->setInitiationInterval(1);
...
for (j = 0; j < _PB_NY; j++) {
    CFC4->updateReads(4,j,i,j,i);
    CFC4->updateExecution();
    CFC4->updateWrites(1,j);
    proc4( &y[j],y[j],A[i][j],tmp[i] );
}
...
/* Performance Analysis */
CM->collect();
CM->createVCD("waveform.vcd");
```

Listing 5: A C program instrumented by cprof.

It shows that the database with information about the orignal program is loaded. From this database, the `CprofManager` is obtained. This object instantiates for each function call in the kernel a `CprofFunctionCall` object. On this object, the basic calibration parameters can be set. In the case of function `proc4`, we set the Initial Interval to 1 and the Function Latency to 18. In the code annotation stage, the original function `proc4` is proceeded with three new functions `updateReads`, `updateExecution`, and `updateWrite`. When executing the annotated program, these three functioncalls are invoked and shadow and control variables are dynamicaly updated. Inside the `CprofManager`, the statement execution profiles are build during execution. We write these statement execution profiles out as value change dump in file *wavefome.vcd*. By using a VCD viewer, a designer can see how the process network executes in time. In Figure 7, a simplified VCD file is shown (taken from [4]). It shows the statement execution profile for three functions. It shows when read, execute and write operations take place. It also shows the Global Execution Profile `G$`. Looking at `G$`, we see that at most 3 operations takes place at the same time. This is the maximum parallelism. From the global execution profile, we can observe that a full execution of the PPN takes 41 time units. Summing up all elements in `G$` gives a total amount of work equal to 35 units. The average degree of parallelism in this execution is $35/41 = 0.85$. This means that, on average, approximately one process is active.

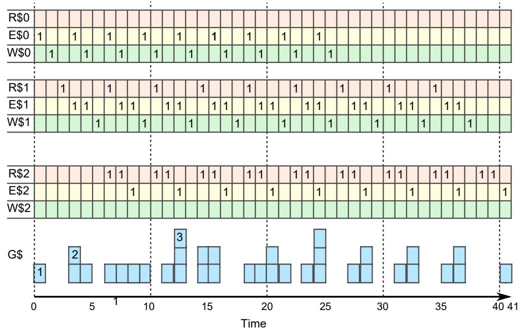

Figure 7: Overview of processing and presenting the output of dynamic analysis [4].

5.2.1 Computational Complexity of Profiling

Profiling comes with a price in terms of both space and time complexities. For each canonical declaration in the source code, a shadow variable and a control variable are created. In Table 1, the cost of profiling in terms of space and time complexities of `cprof` are listed. The memory footprint of the generated program by `cprof` is at least twice the size of the original program. The reason is that for each variable, control and shadow variables are used to keep track of the performance of the program. The space complexity of profiling scales linearly with the number of statements used in the program.

Algorithm	**Space**	**Time**
Read Operations	$\mathcal{O}(c+r)$	$\mathcal{O}(c \cdot r)$
Execute Operations	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Write Operations	$\mathcal{O}(d+w)$	$\mathcal{O}(d \cdot w)$

Table 1: Profiling cost in space and time: c is the number of dimensions of the read argument, r is the number of times the read arguments are referenced throughout the program, d is the number of dimensions of the write argument, and w is the number of times the write arguments are referenced throughout the program.

6. EXPERIMENTS

One of the goals of this work is to validate `cprof` against hardware implementations of the PolyBench/C 3.2 benchmarks [9] which consists of 29 C-programs. All these C-programs are SANLP compliant. We validate these cases against the RTL implementation generated by Compaan. Each PolyBench/C 3.2 benchmark is configured to use the `MINI_DATASET` size.

The platform used for profiling the benchmarks with `cprof` is an Intel i7-3520M operating at 2.9 GHz, with 8 GB of internal memory. The Xilinx Vivado 2015.4 simulator is used for the purpose of simulating the RTL designs of the benchmarks, and the FPGA board used is a Virtex-7 FPGA (xc7vx690tffg1761). The clock period is 5 ns and the clock frequency is, thus, 200 MHz.

Table 2: The characterization of procedures in the outlined version of the Atax program.

Procedure	II_F	Λ_F
proc0	1	1
proc3	1	18
proc4	1	18
proc5	1	1

For each outlined function of all the 29 C-programs, we have determined the function latency (Λ_F) and the initiation interval (II_F). Since we have not tried to share resources, the initiation interval for all functions is 1. For most outlined functions, the latency is 1. This is typically for functions needed to distribute data like `proc0` and `proc5` in the PPN for the Atax program. The complete characterization of procedures in the outlined version of the Atax program is given in Table 2. Overall, we found the following pipelines depths: 1, 2, 3, 7, 8, 10, 11, 14, 18, 26, 34, 38, 42, 46, 54, 57, 70, 102. The pipeline depth of 102 was found, for example, in seidel_2d. It is interesting to mention that after a few cases, we could already guess quite well the pipeline depth of functions.

6.1 Measurements

The average degree of parallelism estimates by the absolute and unbounded performance are shown in Figure 8. The Y-axis is in $\log_{10}$ scale. These measurements give a lower and upper bound of the design space, in terms of the average degree of parallelism, as presented in Figure 1. The average degree of parallelism provides a more realistic design point that is close to the maximum achievable performance [1]. Looking at the Atax program of Listing 1, we find a maximum degree of parallelism of 37 and an average degree of parallelism of 35.4 for the absolute performance of 19948 clock cycles. We find a maximum degree of parallelism of 1024 and an average degree of parallelism of 285.3 for the unbounded performance of 1286 clock cycles. This means that out-of-the-box, we run Atax in 19948 clock cycles on the FPGA. We will never be able to run Atex faster then 1286 clock cycles. The difference between the average degree of parallelism of 35.4 versus 285.3 operations per clock cycle indicates that there is a lot of opportunity to expose more parallelism to run Atax faster then 19948 clock cycles, of course at the expense of more hardware.

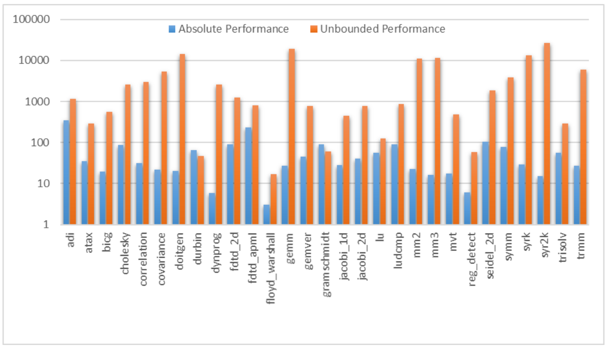

Figure 8: Design space boundaries: the average degree of parallelism in PolyBench/C kernels found by the absolute and unbounded performance estimates.

The maximum degree of parallelism found by the absolute and unbounded performance estimates gives insight to the theoretical maximum parallel performance of PPNs. The results are shown in Figure 9. The Y-axis is in $\log_{10}$ scale.

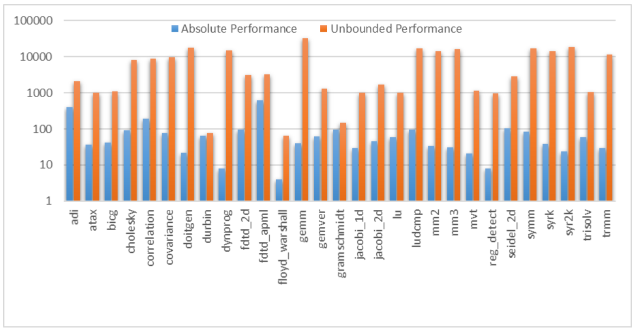

Figure 9: Design space boundaries: the maximum degree of parallelism in PolyBench/C kernels found by the absolute and unbounded performance estimates.

To determine the quality of the results of `cprof`, we determine the difference between the execution finish time given by `cprof` versus the execution finish time given for the simulation of the RTL implementations, as given in Table 3. For example, `cprof` finds for the Atax program a total of 19948 clock cycles, while the RTL simulation finds 19963 clock cycles. This is a difference of 15 clock cycles, which is a relative difference of 0.1%. Looking at all 29 experiments, on average, `cprof` overestimates the execution finish time by 1.2%. This means that `cprof` does a correct estimate of the final hardware. A few cases stand out with a large error: dynprog, floyd_warshall, gemm, lu, and reg_dect. Inside a PPN, not all edges behave as FIFOs [13]. These edges are implemented with out-of-order buffers that are not yet modeled correctly in `cprof`. In `cprof`, all edges are assumed to behave like a FIFO. This explains the differences we see in run-time. Nevertheless, we see that 24 out of 29 experiments already give very good results, showing that the developed profiler techniques work. Including out-of-order buffers is left as future work. Once realized, we are confidant that the number of clock cycles found by `cprof` will be close to the RTL clock cycles for all experiments.

In Table 3, we have also included the Unbounded execution finish time. This is the time needed to finish executing an algorithm assuming infinite resources. It is not possible to run an algorithm faster than this time. For Atax, we find an Unbounded execution time of 1286 clock cycles. If we divide the Absolute execution time by the Unbounded execution time, we get a factor that indicates the opportunity available for optimizing. This is given in the last column of Table 3. A high number means there is ample opportunity to exploit various forms of parallelism to make an algorithm run faster, using for example the techniques given in [11] although many other techniques also exist. A lower number means little opportunity. For Atax, we find an optimization factor of 16.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented `cprof`, a tool for the profiling of polyhedral process networks. `Cprof` allows us to estimate the performance expressed in clock cycles of designs specified in C-code early in the design flow. `Cprof` instruments source code to estimate the performance of sequential C-code when implemented as a PPN in hardware. To do so, no complex and time consuming synthesis step needs to be performed. As a result, we get clock cycles in seconds instead of waiting for hours on synthesis results. Also, `cprof` turns out to be very robust. To determine the absolute and unbounded parallelism, it does not have to perform a complex

Table 3: Absolute performance estimates of the execution finish time of PolyBench/C kernel versus the execution finish time of RTL implementations. Also included is the Unbounded execution finish time and the optimization factor we get when dividing Absolute by Unbounded execution finish time.

PolyBench/C Kernel	Vivado RTL	Cprof Abs.	Diff (%)	Cprof Unb.	Optim. factor
adi	5606	5590	0.3	1094	5
atax	19963	19948	0.1	1286	16
bicg	19911	19897	0.1	646	31
cholesky	91604	90552	1.1	2438	37
correlation	338579	337392	0.4	2067	163
covariance	342014	340876	0.3	1085	314
doitgen	181047	182028	0.0	209	866
durbin	10909	10743	1.5	152	71
dynprog	253593	187874	25.9	289	650
fdtd_2d	2140	2125	0.7	150	14
fdtd_apml	4609	4412	4.3	725	6
floyd_warshall	2086	746	64.2	116	6
gemm	99251	33762	66.0	46	734
gemver	55050	54972	0.1	1838	30
gramschmidt	4721	4510	4.5	2903	2
jacobi_1d	1045	1031	1.3	65	16
jacobi_2d	1951	1938	0.7	97	20
lu	31571	10743	66.0	1866	6
ludcmp	122773	122656	0.1	13905	9
mm2	521222	519096	0.4	838	619
mm3	631730	627683	0.6	646	972
mvt	10724	10649	0.7	326	33
reg_detect	4571	3649	20.2	143	26
seidel_2d	181242	181228	0.0	9574	19
symm	281367	279435	0.7	1510	185
syrk	508979	508958	0.0	527	966
syr2k	1080382	1016878	5.9	1039	979
trisolv	9518	9497	0.2	1909	5
trmm	420323	418625	0.4	1742	240

data-dependency analysis.

The average degree of parallelism found in the absolute and unbounded performance estimates give the expected lower and upper bound of the design space. The PolyBench/C benchmarks were profiled, and showed that `cprof` gives reliable estimates of the execution finish time of the PolyBench/C benchmarks implemented in hardware. It thereby helps increasing engineering productivity and reduces risk as design limitations are made explicit early in the design process. The use of `cprof` results in a design flow that is equal to a regular software design flow for which special hardware skills are not required to analyze and optimize a design that is eventually implemented as a PPN in hardware.

Future work is to include out-of-order buffering in `cprof` to improve accuracy. We also want to extend `cprof` with Hierarchical Program Analysis (HPA), which is about profiling programs with inter-procedural relations. As an example, the Polybench/C kernels will typically be used as part of larger applications and we want to be able to determine the performance of such applications. Overall, `cprof` provides a promising basis for further research in C-code optimization to automatically determine the curve in Figure 1 for FPGA implementations.

8. REFERENCES

[1] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[2] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *ACM SIGPLAN Notices*, volume 46, pages 458–469. ACM, 2011.

[3] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 39(4):49–57, 2004.

[4] S. J. J. Haastregt. *Estimation and optimization of the performance of polyhedral process networks*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2013.

[5] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor. Kremlin: Like gprof, but for parallelization. In *ACM SIGPLAN Notices*, volume 46, pages 293–294. ACM, 2011.

[6] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, may 2000.

[7] O. Krzikalla. Performing source-to-source transformations with clang, 2013. Poster presented at the European LLVM Conference Paris 2013, Paris, France.

[8] M. J. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[9] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *http://www.cs.ucla. edu/˜ pouchet/software/polybench/*, 2012.

[10] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[11] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Tenth International Symposium on Hardware/Software Codesign CODES'2002*, may 2002.

[12] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using kahn process networks: The compaan/laura approach. In *Design, Automation and Test in Europe conference (DATE04)*, feb 2004.

[13] A. Turjan, B. Kienhuis, and E. Deprettere. Classifying interprocess communication in process network representation of nested loop programs. *ACM Transactions on Embedded Computing Systems*, 6(2), may 2007.

[14] A. J. C. Van Gemund. *Performance modeling of parallel systems*. PhD thesis, Delft University of Technology, Faculty of Electrical Engineering, 1996.

[15] S. Verdoolaege. Polyhedral process networks. In *Handbook of Signal Processing Systems*, pages 1335–1375. Springer, 2013.

[16] X. Wu. *Performance evaluation, prediction and visualization of parallel systems*, volume 4 of *The International Series on Asian Studies in Computer and Information Science*. Springer, 1999.

[17] C.-Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *8th International Conference on Distributed Computing Systems*, pages 366–373. IEEE, 1988.

[18] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. Laura: Leiden architecture research and exploration tool. In *International Conference on Field Programmable Logic and Applications (FPL)*, sept 2003.