DSA Monitoring Framework for HW/SW Partitioning of Application Kernels leveraging VPs

Christoph Hazott
Institute for Complex Systems
Johannes Kepler University Linz, Austria
christoph.hazott@jku.at

Daniel Große
Institute for Complex Systems
Johannes Kepler University Linz, Austria
daniel.grosse@jku.at

***Abstract*—Rapid and efficient design space exploration is the key enabler for effective *Domain Specific Architectures* (DSAs) development. In this paper, we present a DSA monitoring framework that supports users in determining an effective HW/SW partitioning for SW application kernels. Our framework leverages *Virtual Prototypes* (VPs) to record HW metrics when executing the application kernels. The core of our framework is a novel monitoring approach using runtime code manipulation on the VP binary. The approach does not necessitate any modification of the VP or SW. We demonstrate the quality of our approach on a RISC-V VP platform running canny edge detection on video frames.**

I. Introduction

While the consensus is that technology-driven improvements are slowing down, Moore's Law is still driving the electronic industry [1]. To continue making exponential progress, a lot of innovation is required challenging process technology (e.g. three dimensional transistors, multi-patterning) as well as the system architecture. More specifically on the latter, *Domain Specific Architectures* (DSAs) fall into the class of heterogeneous architectures that integrate specific *Hardware* (HW) accelerators to meet the performance and power improvements for the considered application *Software* (SW) [2]. Among many research challenges [3], this approach brings in a **very deep interdependence of HW and SW** as the data flow of the application is optimized via HW acceleration while maintaining programming flexibility.

The development of DSAs renewed the importance of HW/SW co-design techniques, in particular for rapid and efficient design space exploration (see e.g. [4], [5].) In this context *Virtual Prototypes* (VPs) are of major importance. A VP is a high-level, executable model of the entire HW platform which runs unmodified production SW [6], [7]. As an industry-proven methodology VPs are utilized to facilitate concurrent development of HW and SW, alongside exploration of design alternatives [8]–[10].

The predominant language for creating VPs is SystemC, a standardized C++ class library [11]; for a broader overview on SystemC we refer the reader to [12]–[14]. SystemC-based VPs allow for orders of magnitudes faster simulation in comparison to *Register Transfer Level* (RTL) models [6]. The key here is *Transaction Level Modeling* (TLM) [15] which abstracts unnecessary communication details but fully captures the HW behavior while the SW is executed on the simulated processor. Such a processor is modeled in form of an *Instruction Set Simulator* (ISS) and is surrounded by accelerators and peripherals forming the VP platform. For simulation, SystemC offers the event-driven SystemC kernel which is compiled together with the VP platform into the *VP binary* for a host (e.g. a x86 machine). Executing this VP binary simulates the system. From the application SW as well as the DSA design perspective, a major task is to determine an optimal HW/SW partitioning early in the design process. Specifically, it is necessary to identify and potentially transfer CPU-intensive and time-consuming application kernels (hotspot functions invoked frequently in loops) into the HW.

Contribution: In this paper, we present a DSA monitoring framework that supports users in determining an effective HW/SW partitioning for application SW. Our framework leverages VPs to record HW metrics when executing the application kernels. These metrics include simulation time as well as memory accesses patterns, necessary to analyze the SW kernels. By this, we enable early design space exploration of DSAs wrt. application kernels.

The proposed monitoring approach for recording HW metrics in relation to SW execution is based on tracing in the sense that we record information of the current system state at regular intervals. Given the crucial importance of very fast VP-simulation speed, two main requirements emerge: (1) only minimal additional overhead for monitoring during simulation, and (2) all HW/SW interactions wrt. kernel execution have to be visible. Current tracing approaches only partially meet these requirements. By implementing the tracing technologies directly into HW or SW, these solutions focus either on the HW perspective or the SW perspective, lacking the full system view. Approaches which combine both do this by creating a trace from the HW perspective and a trace from the SW perspective and combine these two in an expensive post-processing step. **In contrast, our approach takes an external monitoring perspective that encompasses both, the HW and the SW, facilitating a holistic view as a unified system.** To realize our approach we use the runtime code manipulation system *DynamoRIO* [16], [17] for monitoring. DynamoRIO offers an interface for performing custom instrumentation at the binary level during runtime. The interface serves as basis to instrument the unmodified VP binary with our monitors at runtime. These monitors record the value of the ISS program counter (keeping track of the current position within the executed SW) as well as read/write accesses to memories (to built memory access statistics). Overall, this allows to maintain high simulation performance and manageable data amounts to be recorded while keeping the precise HW/SW relations.

To evaluate our DSA monitoring framework, we consider a RISC-V VP platform running a canny edge detection application SW consisting of a hierarchy of SW kernels. We run canny with different image resolutions and demonstrate that the performance overhead of recording the monitoring dataset is less than a factor of 2 for large image resolutions. Moreover, we show that the analysis of the dataset allows the user to identify and transfer SW kernel parts of canny into HW, resulting in a significantly improved system performance.

Summarizing, the major contributions of this paper are:

- Development of a DSA monitoring framework leveraging the observability of VPs
- Use of runtime code manipulation on the VP binary for monitoring, thereby maintaining high simulation performance
- Implementation of specialized monitors for SW kernel profiling, keeping recorded data amount low

This paper is structured as follows: First, related work is discussed in Section II. Thereafter, in Section III, we present the core of our approach. We describe the general Host-to-SW memory hierarchy of a VP simulation followed by the presentation of our monitoring approach through runtime code manipulation on the VP binary. Section IV introduces the proposed DSA monitoring framework. The evaluation of our framework is given in Section V. Finally, the paper is concluded in Section VI.

II. Related Work

Recently, in [3] a survey has been published which presented various research directions and challenges in designing and developing DSAs. The importance of virtual emulation frameworks and TLM-based VPs for DSAs is highlighted. According to the authors, there is a clear demand for dynamic analysis and transformation techniques to scope applications and extract kernels and flow graphs to support design space exploration. In contrast to approaches like [18], [19] and [20], our framework aims to fulfill this demand by leveraging SystemC-based VP platforms for these tasks.

In general, to support the user in design space exploration of a DSA VP platform, the dynamic behavior of the VP simulation has to be taken into account. For VPs, different approaches in this direction have been developed. The first approach we want to mention is [21], which implements a custom C++ library. Due to the high customization which is necessary when using the library, this solution is very intrusive in terms of source code. An alternative approach which removes the necessity to instrument the SystemC source code was presented in [22]. This approach is changing the dynamically linked SystemC library to a custom library, which includes tracing capabilities at runtime. However, using a custom SystemC library limits the use in industrial settings. Moreover, extracting information on the running application SW also requires extensions of the SystemC models. The already mentioned solutions have in common that the trace is generated while a SystemC simulation is executed. [23] is implementing a different approach by tracing the execution flow of an application outside SystemC and in a second step simulates this trace in SystemC to generate the HW metrics. This solution introduces a large overhead because multiple tools are necessary to generate the desired output. Tracing of HW and SW separately and combining the data has also been considered (see e.g. [24] and [25]), but is overall too costly.

The works [26] and [27] uses the *GNU Debugger* (GDB) to trace the simulation. While these approaches allow for dynamic insights without instrumenting the source code, due to the utilization of GDB, these methods introduce significant execution time overhead, rendering them impractical for tracing complex HW/SW interactions.

Our framework in contrast builds upon a comprehensive understanding of the Host-to-SW memory hierarchy which forms the basis to realize our monitoring via runtime code manipulation of the VP binary; this facilitates a holistic view as a unified system. In addition, we tailor our monitors specifically to identify SW kernels and compute metrics to support HW/SW partitioning.

III. Host-to-SW Memory Hierarchy in VP Simulation and Data Monitoring

In this section, we present the core of our monitoring approach. First, we analyze the precise memory locations of the data, e.g. *Program Counters* (PCs), within the Host-to-SW memory hierarchy designated for monitoring during the simulation (Section III-A). Based on these locations, we then introduce our monitoring approach and show how the VP binary is instrumented at runtime using DynamoRIO (Section III-B). Last, we show how our framework can be tailored to SW kernel monitoring (Section III-C).

A. Locating Simulated HW/SW Data in Host-to-SW Memory Hierarchy

Simulation at early design stages is done by representing the HW in form of a VP platform which is typically modeled in SystemC. As SystemC is a standardized C++ class library including the simulation kernel, the VP platform is compiled into the VP binary for simulation. Loading this binary on a host system and executing the contained VP instructions simulates the HW behavior. The VP platform, i.e. the DSA design, is composed of a processor surrounded by accelerators and peripherals. The processor itself is modeled in the form of an ISS, enabling the execution of application SW within the VP platform.

Like the HW, the SW is also present in the form of a binary. Consequently, the VP platform additionally includes a mechanism to load the application SW into the HW memory at the beginning of the simulation. After the SW has been loaded, the ISS can execute the contained SW instructions. Fig. 1 visualizes the Host-to-SW memory hierarchy when executing the VP binary, i.e. simulating a DSA design.

The gray boxes represent the **Host**, the **HW** and the **SW**, respectively. The host is an abstract representation for a machine executing the simulation. An example for such a machine is an x86 workstation running Linux. The host contains registers ($registers_{host}$) as well as an instruction memory ($imem_{host}$)

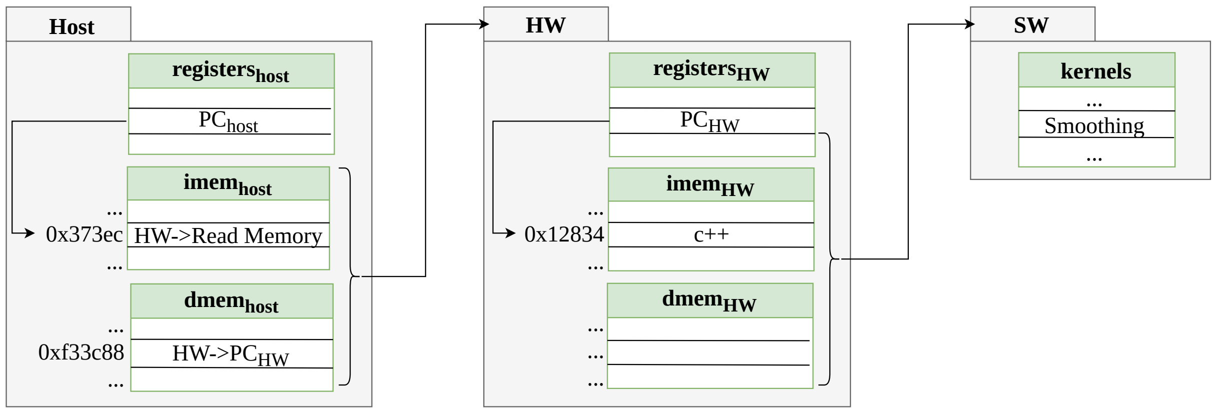


Fig. 1: Host-to-SW memory hierarchy showing simulated HW/SW data locations as basis for monitoring

and a data memory ($dmem_{host}$). These memories contain the HW, i.e. the VP binary (cf. arrow starting from Host box to the HW box).

Like the host, the HW also contains registers ($registers_{HW}$) as well as an instruction memory ($imem_{HW}$) and a data memory ($dmem_{HW}$). In contrast to the host, these memories contain the application SW binary. From the HW perspective, the components $registers_{HW}$, $imem_{HW}$ and $dmem_{HW}$ are part of the processor. From the host perspective these components are variables stored in $dmem_{host}$.

Diving deeper, we can see that the $registers_{host}$ contain the PC_{host}. This PC_{host} register contains an address pointing to a VP instruction (representing the HW of the DSA design) within $imem_{host}$. As an example in Fig. 1, we see that the PC_{host} points to address $0x373ec$ which contains the instruction responsible to simulate a read from $dmem_{HW}$.

As stated above, $dmem_{host}$ contains variables representing the $registers_{HW}$. One of these registers is the PC_{HW} containing an address pointing to a SW instruction within $imem_{HW}$. In the concrete example of Fig. 1, PC_{HW} is pointing to address $0x12834$, where we find the SW instruction to increment the SW variable `c`.

Understanding the respective memory locations and their meaning wrt. the HW and SW of the considered DSA design in the Host-to-SW memory hierarchy is essential for our monitoring approach. Leveraging this knowledge, we can take an external perspective for monitoring and get a holistic view as a unified system. The concrete implementation in terms of monitoring is presented in the next section.

B. Monitoring via Runtime Code Manipulation on the VP Binary

A major goal of our monitoring approach is to maintain high simulation performance. Although the data of interest could be monitored by implementing tracing technologies directly into the HW/SW system (i.e. the VP and/or application SW), such an implementation would introduce a large overhead during simulation and influences the HW/SW interaction since additional code has to be interpreted by the ISS. Leveraging the locations of the data along the Host-to-SW memory hierarchy as described in the previous section and illustrated in Fig. 1, we now aim to record the current position and state of the HW simulation and the contained application SW by monitoring registers and memories of the host. One solution would be to use technologies like *ptrace* [28] which allows to stop and to continue the simulation with host interrupts and record the current system state. However, the extremely frequent stopping and resuming of the host execution results in a tremendous overhead due to the required context switches on the host system. To solve this problem, we leverage the *Dynamic Binary Instrumentation* (DBI) tool DynamoRIO. DynamoRIO allows users to insert custom instrumentation code into a running program without the need to modify its source code or recompile it. DynamoRIO operates at the binary level, making it possible to observe and modify program behavior during execution. When DynamoRIO is dynamically instrumenting a target program, it allows users to specify certain events for which they want to be notified. When the target program encounters one of these events during its execution, DynamoRIO invokes the corresponding *event handler*. Essentially, these event handlers are callback functions that users can define to respond to specific events. In this work we make use of *Basic Block*[1] (BB) events. A BB event is triggered in DynamoRIO, whenever the program enters or exits a BB of code. In our approach, we register respective event handler and realize our monitors as follows:

a) Monitoring PC_{HW}: The code in Listing 1 shows part of the code which performs dynamic code insertion into the VP binary for monitoring the PC_{HW}. Given the VP binary instructions of a BB as identified by DynamoRIO, the code in Listing 1 is run for each `instr` of the BB. It first checks whether the current BB-instruction of the VP binary writes into the $dmem_{host}$ by calling the `instr_writes_memory()` function of the DynamoRIO API. If this is the case, the address of the memory write operand of the instruction is retrieved. Next, at Line 6 of Listing 1 it is checked whether the address matches the PC_{HW} (`pc_hw`) address to be monitored (see Fig. 1 and address $0xf33c88$). If so, a clean call to the function `clean_call_pc_hw` is inserted which will records the PC_{HW} value. A clean call is a function call which stores

[1]A basic block is a block of host instructions that are executed sequentially. Additionally, these instructions contain only one entry point and one point for leaving the block, meaning only the last instruction is allowed to be a branch.

```
if(instr_writes_memory(instr)){
    addr = opnd_get_addr(
            instr_get_dst(instruction, i)
            );

    if(addr == pc_hw) {
        dr_insert_clean_call(
            ..., clean_call_pc_hw, ...);
    }
}
```

Listing 1: DynamoRIO code fragment performing instrumentation for monitoring the PC_{HW}

```
pc_host = instr_get_app_pc(instr);
if (monitor_pc_host[pc_host]) {
    dr_insert_clean_call(...,
        clean_call_hw_mem_read, ...);
}
```

Listing 2: DynamoRIO code fragment performing instrumentation for monitoring HW memory access

the host state when entering, and restores the host state when leaving. This ensures that the simulation is not affected by the function call.

b) Monitoring HW Memory Access: For monitoring the access to $dmem_{HW}$, we also use the BB event of DynamoRIO and therefore again for each `instr` of a BB we run the code in Listing 2. Note that the VP source code contains several locations where the $dmem_{HW}$ is accessed. For these locations we are able to derive the PCs, i.e. for each a concrete PC_{host}; in Fig. 1 see address $0x373ec$ as an example. For instrumentation via DynamoRIO Listing 2 starts by retrieving the PC_{host} of the VP binary instruction in the current BB. Then, it is checked whether this PC_{host} should be monitored, i.e. if it is a location like $0x373ec$. The Boolean array `monitor_pc_host` provides exactly this information and is filled as described in Section IV. Finally, a clean call to `clean_call_hw_mem_read()` will be executed which is recording a VP memory read event.

How we tailor the monitoring for SW kernels is described in the next section.

C. SW Kernel Monitoring

Application kernels are fragments of the application SW which are frequently executed, i.e. typically in loops. Listing 3 shows an example of nested application kernels implementing a Gaussian filter for smoothing image rows. The information within this code fragment we are interested in is the runtime behavior based on the contained kernels. In the example, we see three for-loops. The first one ranging from Line 1-12 looping through the image columns. This first kernel contains a nested kernel looping through the image rows and ranges from Line 2-11. Line 4-9 contain the deepest kernel of our example.

As already said before, when compiled this code is stored in form of SW instructions within the $imem_{HW}$. Typically, identifying how much time has passed and how many memory

```
1  for(c=0;c<cols;c++) {
2      for(r=0;r<rows;r++) {
3          ...
4          for(rr=(-center);rr<=center;rr++) {
5              row = r + rr;
6              if(row >= 0 && row < rows) {
7                  ...
8              }
9          }
10         smoothedim[r*cols+c] = ...
11     }
12 }
```

Listing 3: Nested kernels implementing gaussian filter for smoothing image rows

accesses occurred is done by recording all HW instructions and extracting the information in a post-processing step. However, this results in an exceedingly large volume of data (reaching hundreds of GBs). In contrast, we only record the advancing PC_{HW} alongside accumulated simulation time and memory access metrics. This significantly reduces the amount of data (few hundreds of MBs).

Based on the described monitors, we introduce the proposed DSA monitoring framework in the next section.

IV. DSA Monitoring Framework

In this section, we present the proposed DSA monitoring framework. The main purpose of the framework, besides monitoring and analyzing the results, is to translate the address based interaction of our monitoring approach into source code based interaction for the user.

We begin by showing the framework overview and then proceed to describe its individual components in the subsequent subsections.

A. Overview

Starting point of our DSA framework, as presented in Fig. 2, builds the *Framework Configuration* file. This file is manually created by the user and contains the links to the HW and SW binaries as well as the source code references tailoring the framework for application kernel profiling.

As our monitoring approach expects host addresses but a user usually wants to interact based on source code references, a *Source Code Translator* is implemented and a resulting *Address Table* is handed over to the DSA Monitor.

Next, the *DSA Monitor* executes our monitoring approach as described in Section III, creating the *Monitoring Dataset*.

In the last step, the monitoring dataset is processed by the *DSA Analyzer*. In this step, the monitoring results are analyzed in correlation to the source code of the SW.

The following sections elaborate these components in more detail.

B. Source Code Translator

The first component of our framework which is executed is the source code translator. The framework configuration, as shown in Listing 4, is parsed at the beginning of the translation and is built up as a list of tags with corresponding values.

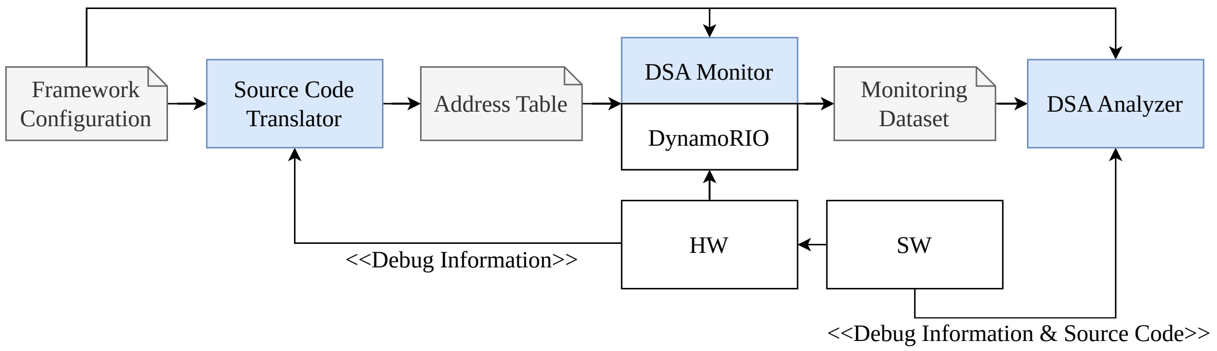


Fig. 2: DSA monitoring framework: Boxes highlighted in blue are the main components of our framework. Boxes highlighted in gray show the input/output exchanged between them. Boxes drawn in white are given.

```
1 HW:'riscv_vp'
2 ...
3 PC_HW:'PC_VP'
4 ...
5 MEM_READ_HW:'/riscv_vp/memory.h:76'
6 ...
```

Listing 4: Framework Configuration excerpt

```
1 PC_HW:'0xf33c88'
2 ...
3 MEM_READ_HW:'0x373ec'
4 ...
```

Listing 5: Address Table excerpt

Line 1 contains the HW tag with the value of the binary to be simulated, in our case a RISCV-VP binary (`riscv_vp`). As we want to instrument our VP binary on a Linux machine, it is compiled as *Executable and Linkable Format* (ELF) binary. To be able to translate the source code references into host addresses, the binary also needs to be compiled with enabled *DWARF* debugging information. DWARF was developed along with ELF and is the default format used in debuggers like the *GNU Debugger* (GDB). It's important to note that enabling DWARF debug information does not have any impact on performance.

Line 3 of Listing 4 tells the framework the name of the PC_{HW} variable within the simulator, in our case the variable is called `PC_VP`. The `MEM_READ_HW` tag in Line 5 tells the framework, that executing this line of code performs a memory read within the simulated HW. Utilizing the debug information extracted from the VP binary, the configuration is translated into corresponding host addresses, resulting in an address table as shown in Listing 5. Line 1 gives an example where we see that the variable name `PC_VP` was translated into the address $0xf33c88$. Furthermore, the code line reference of `MEM_READ_HW` was translated into address 0x373ec in Line 3.

With this information, the DSA Monitor is able to insert our DSA tailored monitors.

C. DSA Monitor

Similar to the source code translator, the DSA monitor obtains the links to the HW and SW binaries from the

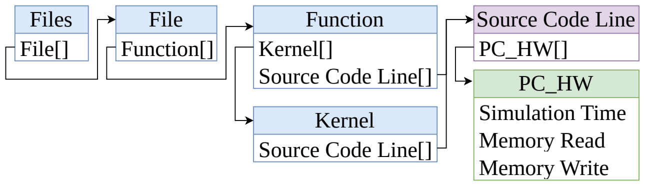


Fig. 3: Data structure used for analysis

framework configuration for simulation. The DSA Monitor is implemented in accordance with the approach from Section III, leading to the generation of the monitoring dataset. The monitoring dataset is a binary file containing a sequential list of PC_{HW} values with corresponding HW metrics and is handed over to the DSA analyzer.

D. DSA Analyzer

The DSA analyzer is designed as a standalone application to separate the monitoring from the analysis. This allows to conduct further analysis on the recorded data without the need to rerun the simulation. Additionally, this separation enables easier comparison between multiple simulations.

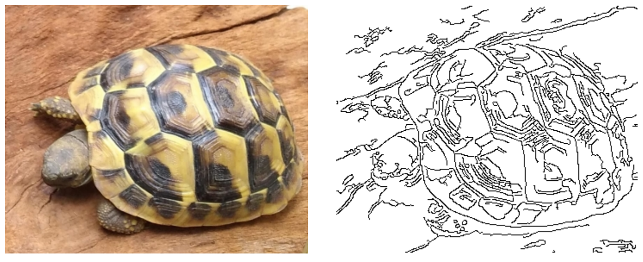
The DSA analyzer parses the framework configuration and the monitoring dataset. The framework configuration contains the SW binary link to parse the debug information of the SW. This debug information is used to analyze the monitoring dataset entries based on the SW source code. To do this, the source code is statically analyzed to generate a data structure containing a hierarchical representation of files, functions and kernels as shown by the blue boxes in Fig. 3.

Functions and Kernels contain the references to source code lines (purple box in Fig. 3). As a source code line usually is compiled into multiple instructions, it contains an array of correpsonding PC_{HW} addresses. This allows us to link the list entries from the monitoring dataset(green box in Fig. 3) to the corresponding functions and kernels.

After the data structure is created, the analyzer becomes capable of generating and correlating HW/SW interactions for files, functions and kernels. In the next section, the results of such analyses are shown.

V. Evaluation

In this section, we present the evaluation of the proposed DSA monitoring framework. We start to briefly describe the



(a) Original (b) Result

Fig. 4: Canny edge detection example

DSA design used in the experiments as well as the experimental setup (Section V-A). Thereafter, we evaluate the costs and scalability of our monitoring approach (Section V-B). Finally, we demonstrate how our frameworks helps in HW/SW partitioning (Section V-C).

A. DSA Design and Experimental Setup

The HW of our DSA design was modeled on top of the open-source *RISC-V VP* from [29], [30]. More precisely, we used the open-source *RISC-V VP++* which provides several improvements (e.g. [31]) and is available on GitHub[2]. The RISC-V VP was configured with 32 MB of memory and an instruction execution time of 1 ns which corresponds to a 1 GHz clock frequency.

As SW application we selected the well-known *canny edge detection algorithm* which processes incoming video frames. The respective images for the canny SW are captured via a camera peripheral connected to the TLM bus. The camera can be configured via corresponding registers to set up image resolution and capture intervals. As basis for our experiments, the canny algorithm is implemented fully as SW (consisting of 35 kernels) and compiled with the RV32I base instruction set. The main stages of the canny algorithm are:

- Gaussian **smoothing**
- Computing **derivatives**
- Computing **magnitude** of gradiant
- Performing non-maximal **suppression**
- Applying **hysteresis**

Applying canny to the image shown in Fig. 4(a) produces the output image depicted in Fig. 4(b).

All evaluations have been performed on an Intel Core i7-10700 with 64 GB RAM under Ubuntu 22.04.2 LTS.

B. Costs and Scalability of Proposed Monitoring

As the monitoring approach introduced in Section III builds the heart of our framework, we wanted to evaluate it separately in terms of scalability and costs. To obtain a representative amount of samples, we varied the number of executed RISC-V instructions and memory accesses. For our HW/SW setup, this can be easily achieved by setting different frame resolutions. We picked six resolutions from the CIF standard format [32]. To determine the upper bound for the CIF resolutions which our system can compute, the expected memory usage for storing a frame while processing is calculated. For a resolution of 9CIF, the canny application SW needs 18.27 MB of memory, fitting into the configured RISC-V memory. We also evaluated small resolutions since we expected much faster simulation times. Table I summarizes the results running canny for a single frame. The different configured resolutions are given in the columns. The first row lists the number of executed RISC-V instructions. Row two and three show the host time needed for simulation without monitoring (*Host time - no monitoring*) in minutes and the host time needed for simulation with monitoring (*Host time - monitoring*), respectively. The resulting overhead factor is given in row four (*Overhead*).

Finally, the last row lists the size of the monitoring dataset in MB. Please note that starting from a resolution of 352x288 and beyond, already billions of RISC-V instructions are simulated. As can be seen, the performance overhead for simulation stabilizes at the resolution of 44x36, i.e. from this point onward, it always remains below a factor of 2. In addition, we plotted the results. They are shown in Fig. 5. The first plot in Fig. 5(a) illustrates the number of pixels needed to be calculated (y-axis) in relation to the corresponding resolutions (x-axis). We generated similar plots which relate the resolutions to host time without monitoring (Fig. 5(b)), to host time with monitoring (Fig. 5(c)), and to the monitoring dataset size (Fig. 5(d)). As becomes evident from these plots, the same quadratic growth as the number of pixels can be observed, in other words our approach scales with number of to be computed pixels.

C. HW/SW Partitioning

To demonstrate that our DSA monitoring framework supports the users in determining an effective HW/SW partitioning, we use the results of our DSA analyzer.

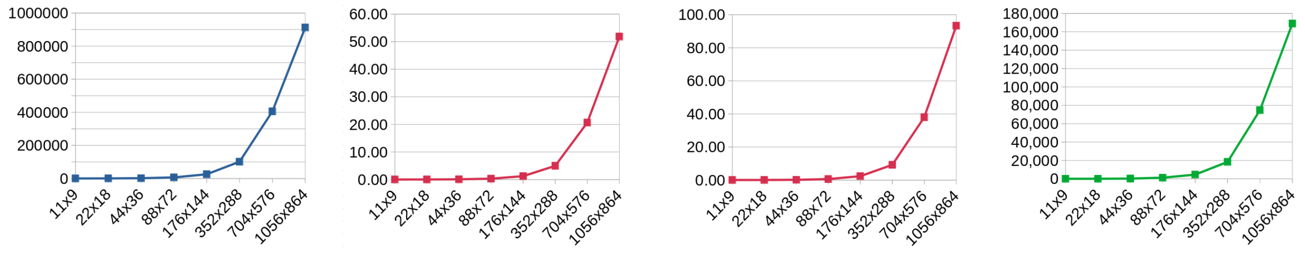
However, before we can start to move SW parts of canny (in particular SW kernel parts) into HW, we first have to determine for which resolution we run the simulations and still get representable results. This means out of the used resolutions we want to find the one where the number of pixels becomes the dominant factor. We did this by computing the average deviation of the stages between two resolutions utilizing our proposed framework. The deviation between 88x72 and 176x144 is only 0.59%, indicating that the resolution of 88x72 produces a sufficiently accurate result, enabling us to extrapolate the findings to higher resolutions.

A natural approach for HW acceleration is to leverage additional RISC-V ISA extensions [33], such as multiplication, floating point, etc. We used our DSA monitoring framework to analyze the SystemC simulation time for the different filter stage kernels of the pure SW canny on RV32I as well as on RV32IMAFC. The bar graphs are shown in Fig. 6(b) and Fig. 6(c), respectively. Note that the results for RV32IMAFC have been obtained by activating the Multiplication/division, Atomic, Floating-point, and Compressed ISA extensions in the RISC-V VP and recompiling the SW with this architecture setting. The results of our framework with respect to memory accesses of canny can be seen in Fig. 7(b) and Fig. 7(c), respectively. We can see that the simulation time (Fig. 6(a)) and the number of memory accesses (Fig. 7(a)) has decreased by comparing the bar for *RV32I* with the bar for *+MAFC*.

[2]https://github.com/ics-jku/riscv-vp-plusplus

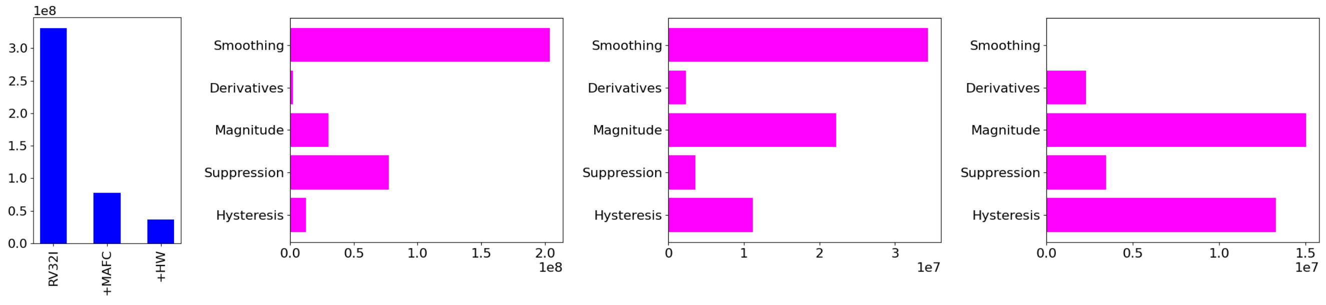
TABLE I: Costs and scalability of monitoring for different resolutions running canny edge detection

	11x9	22x18	44x36	88x72	176x144	352x288	704x576	1056x864
Executed RISC-V instructions	2,918,759	7,350,190	25,637,385	98,984,371	395,274,305	1,584,436,030	6,502,028,739	14,746,998,363
Host time - no monitoring [min]	0.01	0.03	0.09	0.32	1.28	4.97	20.66	51.84
Host time - monitoring [min]	0.04	0.06	0.17	0.60	2.39	9.20	38.02	93.34
Overhead factor	3.45	2.19	1.91	1.84	1.86	1.85	1.84	1.80
Size of monitoring dataset [MB]	34	85	294	1,229	4,608	18,432	74,752	168,960



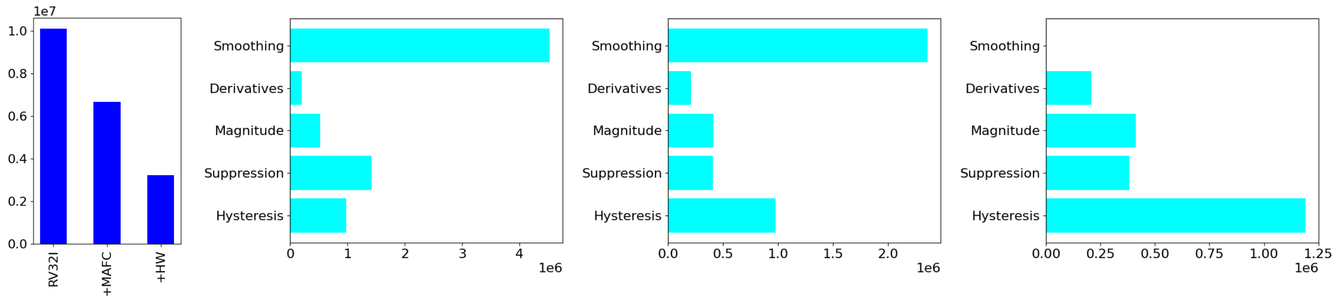
(a) Number of pixels (b) Host time - no monitor. [min] (c) Host time - monitoring [min] (d) Monitoring dataset size [MB]

Fig. 5: Plotted scalability for monitoring



(a) Overall (b) RV32I (c) +MAFC extensions (d) +HW smoothing

Fig. 6: SystemC simulation time [ns]; overall (blue) and separated simulation times of filter stage functions (magenta)



(a) Overall (b) RV32I (c) +MAFC extensions (d) +HW smoothing

Fig. 7: Memory accesses; overall (blue) and separated memory accesses of filter stage kernels (cyan)

The plus in performance for the stages are mainly caused by the fact that without multiplication/division and floating point support in HW expensive SW emulations are included (e.g. a soft-float library).

The most significant performance improvement, when enabling the MAFC extensions, was observed for smoothing and suppression, but still smoothing remains the hotspot of the canny SW. Therefore, we decided to move smoothing into HW. More precisely, the camera module performs smoothing directly on the image. The results of this acceleration are shown in Fig. 6(d) and Fig. 7(d), respectively. We observed a strong reduction in total simulation time (Fig. 6(a)) as well as memory accesses (Fig. 7(a)) by comparing the bar for *+MAFC* with the bar for *+HW*. Additionally, we can see that smoothing is not part of our SW kernel analysis anymore. It is crucial to highlight that even though there are similarities, we observe a distinct change in the filter stages that are still implemented in SW. It appears that parts of magnitude have moved into hysteresis. Given that we are simulating a VP platform with peripherals, our simulation also incorporates interrupts along with their corresponding interrupt service routines. As we have moved smoothing into HW, the remaining filter stages complete earlier, leading to interrupts occurring at a subsequent filter stage. In this case the interrupt

TABLE II: Final FPS results for canny

	RV32I	+MAFC	+HW
Kernel 0 [ms]	329.71	77.28	37.88
FPS	3	12	26

did not appear in the magnitude stage but in the hysteresis stage. This highlights the valuable insights we obtain through our DSA monitoring framework.

Finally, we evaluate the performance of the different HW/SW partitionings based on the *Frames Per Second* (FPS). Table II reports the results. Each canny implementation is shown in a column, i.e. *RV32I*, *+MAFC*, and *+HW*, respectively. The simulation time for the frame capture kernel for the different implementations are given in row *Kernel 0* in milliseconds while the resulting FPS is shown in row *FPS*. As can be seen, the first implementation achieves only 3 FPS, but using an RV32IMAFC core and moving HW smoothing into HW leads to 26 FPS.

VI. Conclusions

In this paper we introduced a DSA monitoring framework for HW/SW partitioning of application kernels leveraging VPs. The core is a novel monitoring approach that centers around the Host-to-SW memory hierarchy in SystemC VP platform simulations. This approach enabled us to observe the simulation from a holistic view, perceiving it as a unified system. Leveraging the dynamic binary instrumentation tool DynamoRIO allowed us to insert monitors into the simulation with a very low overhead. Instrumenting only application kernel specific monitors kept the data amount manageable, tailoring our approach for DSAs.

In our framework, the user specifies the code to be monitored, which is subsequently translated into PC addresses for the monitoring process. To analyze the dataset collected during monitoring, we have developed a DSA analyzer as part of our framework. The analyzer correlates HW/SW interactions for files, functions and kernels and generates graphs.

In our evaluation, we considered a RISC-V VP platform running a canny edge detection application SW. Using our framework, we were able to make informed decisions about the minimum resolution, which is also valid for HW/SW partitioning at higher resolutions. Furthermore, we successfully implemented HW accelerations to enhance system performance.

In future work, we plan to integrate advanced user-specific analysis programs via [34] into our framework.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria. Finally, we would like to thank Daniel Pekarek for interesting discussions.

References

[1] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *CiSE*, vol. 19, no. 2, pp. 41–50, 2017.
[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan 2019.
[3] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge, "Domain-specific architectures: Research problems and promising approaches," *TECS*, vol. 22, no. 2, Jan 2023.
[4] X. Zheng, J. Wu, X. Lin, H. Gao, S. Cai, and X. Xiong, "Hardware/software co-design of cryptographic SoC based on RISC-V virtual prototype," *CAS*, pp. 1–1, 2023.
[5] V. Muttillo, L. Pomante, M. Santic, and G. Valente, "SystemC-based co-simulation/analysis for system-level hardware/software co-design," *Comput. and Electr. Engin.*, vol. 110, p. 108803, 2023.
[6] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.
[7] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *DATE*, 2012, pp. 685–690.
[8] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems," in *DATE*, 2015, pp. 1698–1707.
[9] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast virtual prototyping for embedded computing systems design and exploration," in *RAPIDO Workshop*, 2019, pp. 3:1–3:8.
[10] L. Klemmer and D. Große, "An exploration platform for microcoded RISC-V cores leveraging the one instruction set computer principle," in *ISVLSI*, 2022, pp. 38–43.
[11] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.
[12] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[13] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.
[14] M. Hassan, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.
[15] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009.
[16] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, USA, 2004.
[17] "Dynamorio," https://github.com/DynamoRIO/dynamorio, 2023.
[18] Y. Xiao, S. Nazarian, and P. Bogdan, "Plasticity-on-chip design: Exploiting self-similarity for data communications," *TC*, vol. 70, no. 6, pp. 950–962, 2021.
[19] R. Uhrie, C. Chakrabarti, and J. Brunhaver, "Automated parallel kernel extraction from dynamic application traces," *CoRR*, 2020.
[20] B. Boroujerdian, Y. Jing, D. Tripathy, A. Kumar, L. Subramanian, L. Yen, V. Lee, V. Venkatesan, A. Jindal, R. Shearer, and V. J. Reddi, "Farsi: An early-stage design space exploration framework to tame the domain-specific system-on-chip complexity," *TECS*, vol. 22, no. 2, 2023.
[21] W. Hong, A. Viehl, N. Bannow, C. Kerstan, H. Post, O. Bringmann, and W. Rosenstiel, "Cult: A unified framework for tracing and logging c-based designs," in *S4D*, 2012, pp. 1–6.
[22] N. Bosbach, J. M. Joseph, R. Leupers, and L. Jünger, "NISTT: a non-intrusive SystemC-TLM 2.0 tracing tool," in *VLSI-SoC*, 2022, pp. 1–6.
[23] G. Callanan and F. Gruian, "Estimating stream application performance in early-stage system design," in *ACSCC*, 2022, pp. 816–823.
[24] C. Koehler, A. Mayer, and M. Wurm, "Combined hardware and software tracing of real and virtual embedded system parts," in *MIXDES*, 2012, pp. 340–345.
[25] H. Duan, L. Yu, H. Zhou, and S. Zhang, "An embedded tracing debug implementation for crossbar type bus in muti-core SoC," in *ICET*, 2020, pp. 63–67.
[26] M. Goli, J. Stoppe, and R. Drechsler, "AIBA: an automated intra-cycle behavioral analysis for SystemC-based design exploration," in *ICCD*, 2016, pp. 360–363.
[27] ——, "Automated nonintrusive analysis of electronic system level designs," *TCAD*, vol. 39, no. 2, pp. 492–505, 2020.
[28] P. Padala, "Playing with ptrace," *Linux Journal*, no. 103, 2002.
[29] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.
[30] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, vol. 109, p. 101756, 2020.
[31] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in *GLSVLSI*, 2023, pp. 599–605.
[32] ITU, *H.261 : Video codec for audiovisual services at p x 64 kbit/s*. ITU, 1993.
[33] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.
[34] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.