IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 1 October 2022, accepted 24 October 2022, date of publication 3 November 2022, date of current version 8 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3218710

DynPath–Non-Intrusive Feature-Rich Hardware-Based Execution Path Profiler

MANPREET KAUR JASWAL AND SUBIR KUMAR ROY
Department of VLSI Systems Design, International Institute of Information Technology Bangalore (IIITB), Electronic City, Bengaluru, Karnataka 560100, India
Corresponding author: Manpreet Kaur Jaswal (manpreet.jaswal@iiitb.ac.in)

This work was supported in part by the Visvesvaraya Ph.D. Scholarship Scheme conceived by Digital India Corporation (formerly Media Laboratory Asia), Ministry of Electronics and Information Technology, Government of India.

ABSTRACT Current state of the art application profilers for function/loop (process) based execution paths perform static profiling during design space exploration. Dynamic optimization is a must for heterogenous platforms to achieve low silicon area and better performance in real-time. Hence, it becomes necessary to have a mechanism in the target platform hardware to identify run-time characteristics of various processes of the application code in execution. This should be at both, an individual process level and in the relationship with other co-processes. This paper proposes DynPath - a hardware-based execution path profiler capable of detecting and identifying the processes along with the execution paths, both in a static and dynamic context. It has a rich set of features in comparison to other process-based profilers reported in the literature. Being non-intrusive it does not cause any run-time overheads. To handle the dynamic nature of input data size versus fixed memory row width, the paper proposes a unique approach based on Content Addressable Memory (CAM) for path identification and optimal hardware utilization. Considering that a process can be part of multiple executions paths, the profiler tracks both, the path invocation count, as well as, path-specific process invocation count (PSPIC) for each process. PSPIC logic being computationally intensive, it also proposes an alternative implementation of the PSPIC logic based on resource sharing which helps reduce the total area overhead of the execution path profiler by 87%.

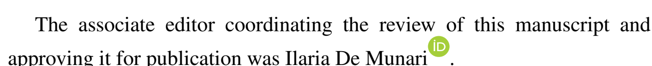
INDEX TERMS Design, dynamic profiler, hardware-software partitioning, non-intrusive, VLSI.

I. INTRODUCTION

Technology is ubiquitous. The applications that we have today are evolving very fast and their design goals are becoming more complex than ever before. Research in emerging technological areas such as machine learning, bioinformatics, robotics, computer vision and artificial intelligence are taking impactful leaps. From an engineering point of view, there are a specific set of challenges to be met for the same. These include design complexity, handling big data in real-time and meeting specific design goals in a dynamic context based on the execution of an application. Profiling the execution statistics offline helps achieve an optimal target platform for deployment [1], [2], [3]. To optimize execution in a dynamic context, the system must be aware of the run-time execution statistics, i.e., the flow of information in a dynamic on the fly context compared to a static offline context.

Static profiling in the context of this paper, refers to the profiling which can be performed during design time to aid the design space exploration phase (refer Section-I-2). It helps the software, as well as the platform designer, analyse the flow of information in an application code and identify potential bottlenecks. Dynamic profiling, on the other hand, is done on-the-fly after the application code has been implemented on the target hardware platform in order to aid on the fly or dynamic optimization during execution of the application code. This is explained further in Sections-I-2, I-3 and I-4.

Flow of information can be further categorized as flow of data and flow of control. Flow of data values could be further used in optimization techniques such as value prediction in microprocessors. Flow of control could aid value prediction


The associate editor coordinating the review of this manuscript and approving it for publication was Ilaria De Munari.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

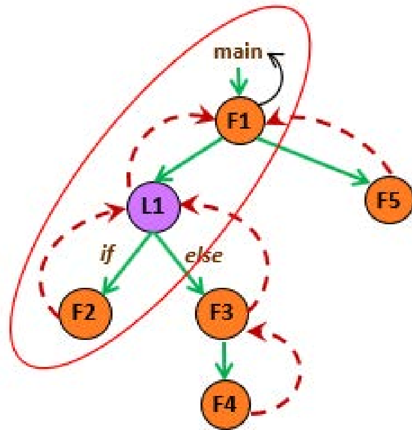


FIGURE 1. Graphical representation of a sample application code.

as well as dynamic reconfiguration or dynamic partial reconfiguration in heterogeneous platforms [2], [4], [5], [6], [7], [8], [9].

1) ACCELERATION - HARDWARE V/S SOFTWARE

Any application code written in a high-level language comprises of complex logic blocks often structured as loops and functions. This in turn forms different paths of execution while the code is executing in an underlying hardware platform unit. The hardware platform may include a processor, or an accelerator, or a reconfigurable logic fabric. We refer to loops and functions as processes for the rest of this paper. For example, Fig. 1 shows a graphical representation of a sample application code comprising of functions denoted as F1 to F5 and a loop L1. As can be seen in the figure, L1 is nested inside F1 and is a parent to F2. Hence the encircled F1-L1-F2 forms one path of execution. These denotations and graphs are further explained in Section III. As per literature [10], [11], computationally complex logic blocks or those with commonly used functionality are often put as functions. Loops, which are the iterative logic blocks, often consume 90% of execution overhead. For example, a multiply and accumulate operation on software could take 1000s of clock cycles based on the number of loop iterations and the latency of every instruction that is executed sequentially; whereas, in a hardware optimized for the multiply and accumulate operation, it will take lesser number of clock cycles due to concurrent or parallel execution based on the number of resources available for carrying out the multiply and accumulate operations. The speedup can be calculated based on a set of equations that can compare the computation and communication costs involved when a process is put in hardware versus implementing it in software. While the hardware execution time can be estimated using total loop iterations, the communication requirements depend on the number of times the loop is executed and can significantly impact overall speedup. Loops with more extensive executions and fewer iterations per execution will have more significant communication requirements than

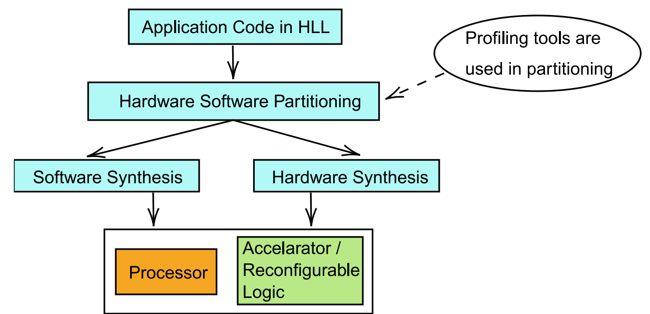


FIGURE 2. Design flow in modern embedded system.

similar loops with fewer executions but more iterations per execution [12].

2) DESIGN FLOW - MODERN EMBEDDED SYSTEMS

Traversing from application-level abstraction to the hardware level abstraction, profiling tools play a significant role in deciding which process is implemented in software or in hardware in the final implementation. The final implementation is based on the target area and performance goals to be achieved when the hardware is deployed. This process of exploration for finalizing the implementation hardware from an application-level considering the design goals is called design space exploration (DSE) [1]. The design flow for developing CPU/FPGA hybrid systems is depicted in Fig.2 [11]. The application specified in a high-level language (HLL) such as C/C++ needs to be partitioned into hardware and software parts with the help of a profiling tool. Design space exploration is carried out to find the suitability of modules in application code to be implemented in the software (Processor) end or the hardware (Accelerator/Reconfigurable Architecture) end. This is a crucial step in designing embedded systems as an optimal partition leads to an optimal system implementation. To partition the system, the application needs to be profiled to identify critical regions. The reconfigurable architecture could be based on static reconfiguration (i.e. pre-configured e.g., usually in FPGAs) or dynamic reconfiguration (i.e. reconfigurable on-the-fly).

3) SCOPE OF PROFILING

The principal objective of this research is to enable hardware/software partitioning decisions for the optimal implementation of a system on a heterogeneous computing platform. The partitioning decisions are taken by following a profiling-based flow that produces accurate measurements of desired metrics. The quantifiable metric required to maximize the throughput is the percentage of time spent in executing a code segment. However, this will only generate information regarding the computation overhead of a specific code segment, function or loop. The frequency of interaction with the neighbouring processes becomes equally important when deciding what portion of code gets implemented in different computation blocks of the target heterogeneous platform.

Profiling, in general, can be categorized as software-based profiling and hardware-based profiling [13]. Software profilers modify the application code by adding instrumentation code to derive profiling information. Hardware profilers typically modify an existing processor implementation on which the process to be profiled is executed. This is done by adding counters that trigger and update counts on the occurrence of specific events. Both approaches contribute to overheads during run-time. With the advent of reconfigurable architectures, we have a third category called the FPGA based profilers. These are hardware-based profilers, augmented to the processor on the same die and do not intend to modify existing processor implementation or contribute to run-time overheads. Having a hardware-based profiler helps perform static analysis on the final target platform. Apart from that, it also helps open opportunities to not just profile an application code in one processor on a platform, but also profile the code regions executing in other computing elements and provide a holistic view of the overall computation and communication. After the product is deployed, the execution characteristics, statically profiled, may differ based on real-time inputs/constraints, leaving room for the possibility of modifying the implementation and dynamic optimization of design goals.

Going further, in future, hardware-based profiling, as described above, can also aid in dynamic partial reconfiguration (DPR) based on dynamic profiling and real-time decisions, as compared to most of the DPRs today, which are proposed based on static profiling and DSE [2], [9]. If the run-time execution overheads can be traced or if it can be found out which process can become active based on past execution history, such processes can be reconfigured to execute on a reconfigurable hardware fabric for acceleration. The processes or execution paths with higher mutual invocation counts can be reconfigured for dynamic placement in the vicinity of each other [14]. This can help in judicious resource utilization, along with power and performance optimization. To enable true DPR, it is required for the system to be aware of exactly which process is it currently executing and calculate some specific run-time parameters, to help enable better decision making as to whether it is worth swapping a process between hardware and software, or vice-versa. A hardware-based dynamic profiler is crucial for such real-time analysis. Subsection V-A points to further details on this. Some specifications have also been laid out in DynRP [15] to aid the same.

4) SCOPE OF DynPath

In the context stated above, non-intrusive quality of a profiler, i.e. being able to automatically detect and identify the target parameter (in our case, processes and their inter-relationship), without intruding into the application code or computing element, becomes a foremost requirement for the following reasons. Firstly, it makes a profiler independent of the application code, or portion of application code executing, or portion of the application code to be executed in a target computing element, as well as, of any instrumentation to be done to the code or the computing element. This is valid both in the case of static and dynamic profiling. Secondly, it is helpful for dynamic profiling in the context of enabling actual dynamic reconfiguration where the processes can be swapped between different computing elements based on real-time execution statistics, for dynamic optimization of design goals.

This paper proposes a hardware-based non-intrusive execution path profiler, which is not only capable of automatically detecting and identifying the processes and the execution paths during DSE, but can also be used in a dynamic scenario to profile an application code executing in a processor, without causing any run-time overheads. The aim is to provide a better granularity of the process inter-relationship by profiling execution paths, the parent-child call hierarchy, and identifying processes contributing to multiple paths. This can lead to a better analysis of processes' computation and communication costs for the target heterogeneous platform during design space exploration. When provided to the Operating System or Job Scheduler [16], [17], [18], [19] targeted for dynamically reconfigurable architectures, this information can help enable seamless execution, resource allocation, and scheduling between a processor and a dynamically reconfigurable architecture to achieve dynamic optimization of design goals.

The outline for the remaining paper is as follows. Section II shares highlights of the comparison with contributions made by existing profilers and the proposed one. Section III dives deeper into the proposed execution path profiler, and it outlines the objectives and terminologies for the rest of the paper. The proposed architecture, the design decisions of various blocks, and the FSM controller are discussed. Fig. 5 and Table 2 give a clear comparison of the proposed blocks as compared to the existing techniques. This section also describes an alternate design logic to reduce profiler area overhead. The proposed execution path profiler results are presented in Section IV and critically analyzed. Conclusion and future scope are discussed in Section V.

II. EXISTING WORK

Table 1 shows the feature comparison of DynPath to other similar function/loop-based profilers. Different features considering dynamic execution path profiling, in the context as stated in Section I are noted.

DynPath currently doesn't support profiling in predefined code region boundaries like SnoopP [20], as it is intended for non-intrusive profiling. But this can be included as an additional feature for desired logic blocks, while continuing support for other loops and functions identified along.

GProf [21] profiles the execution of application code (functions-only) during design time to aid DSE. During compilation the generated application code is instrumented in the software backend based on user-defined flags to enable profiling. Profiling accuracy is directly proportional to the sampling frequency. Thus, for higher accuracy, execution

TABLE 1. Comparison with existing profilers.

Feature		Process based Profilers				
No.	Description	Proposed DynPath	GProf	SnoopP	Kumar	DynRP
1.	Hardware Based Profiling	✓	X	✓	✓	✓
2.	Real Time Profiling	✓	X	✓	✓	✓
3.	Profile regions in user defined code bounds	X	X	✓	X	X
4.	Non-Intrusive Detection and Identification	✓	X	X	✓	✓
5.	Nested Function Detection and Identification	✓	✓	X	X	✓
6.	Function Execution Cycle count	X	✓	X	✓	✓
7.	Function Invocation count	✓ Re-structured to feature 19	✓	X	✓	✓
8.	Recursive Function Identification	✓	✓	X	X	✓
9.	Nested Loop Detection and Identification	✓	X	X	✓	✓
10.	Loop Execution Cycle count	X	X	X	✓	✓
11.	Distinguish Loop iterations from Loop Invocations	✓	X	X	✓	✓
12.	Loop Invocation count	✓ Re-structured to feature 19	X	X	✓	✓
13.	Loop Iteration Count	✓ Partially re-structured to feature 19	X	X	X	✓
14.	Function and Loop in a common profiler	✓	X	X	X	✓
15.	Non-intrusive Execution Path / Call Graph Formation	✓	✓ (only function based)	X	X	X
16.	Calculation of depth of path in runtime to aid identification circuit (can later aid in further deducing code structure, etc.) + Path identification	✓	X	X	X	X
17.	Execution Path Invocation Count	✓	X	X	X	X
18.	Parent-specific Child-Invocation Count (PCIC)	X	✓ (for functions only)	X	X	X
19.	Path Specific Process Invocation Count (PSPIC)	✓	X	X	X	X
20.	Hardware optimizations for handling dynamic depth of paths and other related counts	✓	X	X	X	X

overhead increases. It is able to support complex profiling features such as recursive function detection and the number of invocations of a child process specific to a parent process (PCIC). However, this is not a straightforward task when a profiler is made using hardware logic blocks. The execution path based call graph information and the Path Specific Process Invocation Count (PSPIC) can be inferred later from the Gprof output through manual inference or any visualizing tools and cannot be obtained directly through GProf. DynPath - a hardware based profiler generates the dynamic execution path information which is based on both, functions and loops. This is explained much deeper in Section III along with the design decisions and architecture to support this in a dynamic context and without need for any form of code instrumentation.

In contrast to DynRP [15], DAProf [12], Kumar [11], SnoopP and GProf, DynPath intends to present a hardware implementation specifically focused on features necessary for execution path profiling only. Hence it doesn't include execution count storage memories for functions and loops. The process invocation count memories are efficiently restructured to the paths these belong to. The loop iteration count can be deduced from the invocation counts of its child processes, except for if a loop is an innermost child. Hence this is marked as partial restructuring in Table 1. If considered desirable for the target system, these blocks can be easily plugged in. These will operate independently without causing any run-time latency overhead.

To the best of our knowledge, there also exist performance profilers which focus on finding performance

bottlenecks in hardware accelerators synthesized using High Level Synthesis (HLS) based design flows. For example, in HLS_Profiler [3] the profiler framework uses static and dynamic information made available by the HLS compiler along with special associative rules to establish accurate Source Code (C/C++) to Clock Cycle mapping. The algorithm takes design source code, compiled RTL files, waveforms and synthesis report as inputs. The performance profile of the source code is available in a text file, through which the designer can identify, optimize bottlenecks and update the source code with appropriate pragma directives. These steps are then repeated till the performance target is met. However, such performance profilers are not the current focus in this paper as these are mostly targeted toward static design optimization.

Another profiler that makes use of execution tracing is Intel Processor Trace (Intel PT) [22], [23]. It is an extension of Intel® Architecture that has to be triggered to capture information about software execution using dedicated hardware facilities developed to cause minimal performance perturbation to the software being traced [22]. The intention is to understand why and how did software get to a certain point, or behaved in a certain way for debug purposes. It offers control flow tracing (eg. branch taken/not taken). This in turn generates information in form of a variety of packets to be processed by a software decoder [23]. It targets branching instructions, primarily functions for tracing purpose. It doesn't require to be recompiled with every new build or release, however, the executed images are needed - which makes its use in JIT-compiled environments, or with self-modified code, a challenge [22]. Loops are reported to consume 90% of execution overhead [10]. The proposed DynPath targets to profile the dynamic execution paths and inter-relationship between processes (at the granularity of, and including both loops and functions). This will help understand the execution flow, as well as, communication overhead (inter-process invocation frequency) to target dynamic optimization by making use of technologies such as Dynamic Reconfiguration and JIT compilation. This is explained in Section-I Introduction and Section-V-A Future Scope

III. PATH PROFILING

Path detection foremost requires detecting and identifying various processes (loops and functions) in execution and their relationship. While the tracking unit as proposed in the hardware-based profiler -*DynRP* does an excellent job in detecting various individual and nested processes, it does not preserve the information related to the path of execution these belong to, or whether the path of execution has branched into a separate sub-path or switched to a new path of execution at the root node which is assumed to be the main process itself. This section describes the specifications and design for the path profiler.

A. OBJECTIVE AND TERMINOLOGY

The objectives are stated as follows: -

1) Detection of the dynamic paths of execution in an application code as opposed to static paths of execution. Different inputs (parameters and environmental inputs) for the application code can result in different paths of execution being taken dynamically.
2) Carry out path identification of newly detected paths
3) Calculate the invocation count for each identified path.
4) Calculate path specific process invocation count

Fig. 3 shows the call graph capturing the execution relationship for all the processes (depicted as nodes) existing in a sample exhaustive application code. This sample code is designed after studying multiple benchmark application codes [24], [25] to include and consider the various corner cases that could exist and is purely for illustration alone. The calling sequence of processes as static analysis is shown from left to right in the call graphs, represented by their unique index numbers. Though terminology for call graphs such as nodes, edges, iterative, branching, recursive etc. have existed for decades [26], [27] below we define the terms specific to the inter-relationship of process for the cases specific to path profiling with reference to Fig. 3. The granularity of our call graph is the function (Fn) and loop (Ln) processes as detected by the mentioned hardware profilers, the output of which is input to the path profiling circuit. Path of execution consists of calling sequence of such processes from the root node (int main). For example, Main → F1 → F2 → L1 → F3 → L2 constitutes one path. Similarly, Main → F11 constitutes another path.

- **Nested Nodes-** Child processes that are invoked from within a parent process and are meant to return to the parent process on completion of their execution, at the same point from where they were invoked. For example, in F11 is nested inside L6.
- **Conditional Nodes-** Nodes where a path of execution within the node can branch into several sub-paths of execution due to the presence of either a case statement or multiple if-then-else statements in such nodes. For example, at node L4 we can have three different paths of execution, L4 → F5, or L4 → F6, or L4 → F7, where F5, F6 and F7 respectively, are the three different sub-paths rooted in the node L4.
- **Non-Conditional Nodes-** Nodes where a path of execution within the node does not branch into several sub-paths of execution due to the presence of either a case statement or multiple if-then-else statements in such nodes. Thus, processes which result in non-conditional nodes have linear execution of process statements without any branching. For example, node F4 is a non-conditional process node.
- **Level Nodes-** Nodes representing processes which are not conditional nodes. The completion of execution of such processes initiates the execution of subsequent processes in the application code. Such processes lie in the same process hierarchy and hence form different

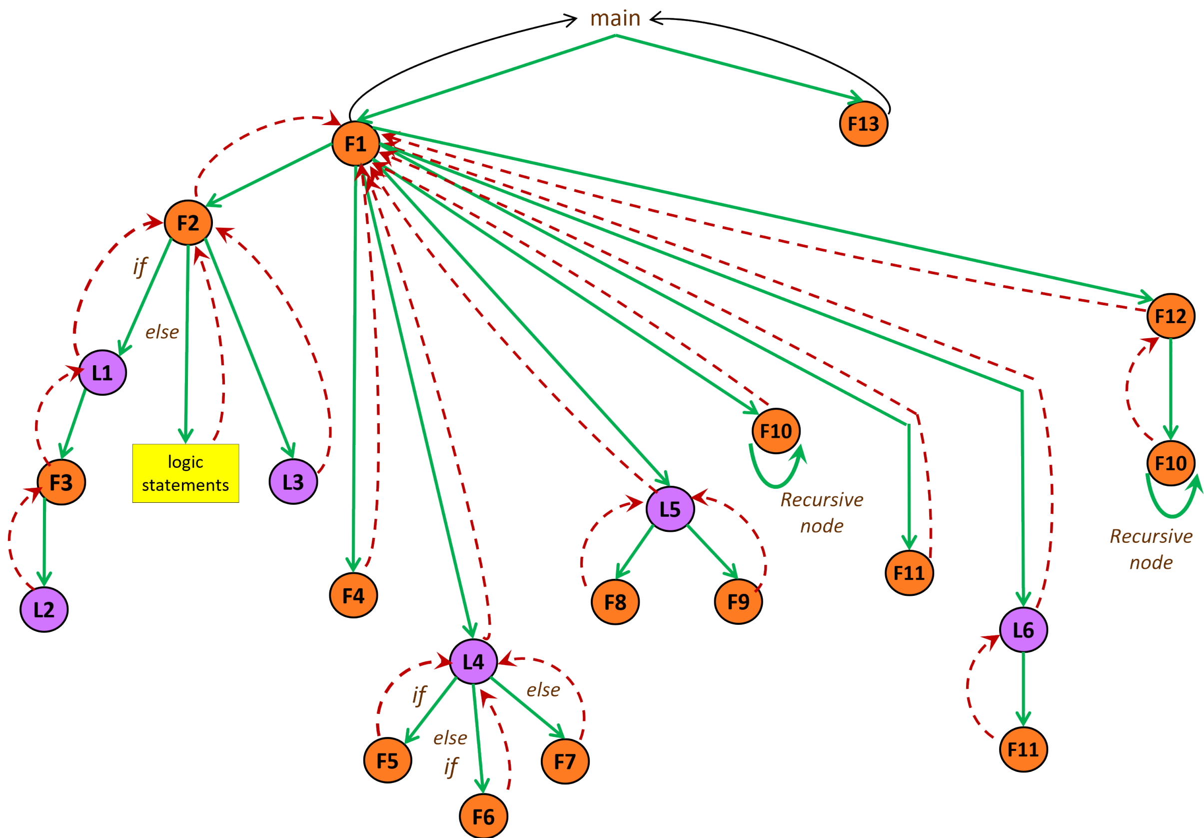


FIGURE 3. Sample exhaustive code structure.

paths from the parent node in a sequential fashion. For example, processes F1 and F13 are level nodes in the same hierarchy under the main node. Once all nested branches within F1 are exhausted, the code execution returns to the main and invokes node F13. Similarly, F2, F4, L4, L5, F10, F11, L6, and F12 respectively are level nodes under F1.

- **Merger Node-** Processes representing a special type of level node, called the merger node on which every conditional branched sub-path in a parent hierarchy seems to terminate in an application code. This, if verified by a code graph is like a non-condition level node which is executed after a condition block in a sub-path. For example, L3 is the merger node for the set of sub-paths rooted in the conditional node F2.

Based on the above context, following types of path formations could exist:-

1) Paths formed from Non branching and Non iterating nodes/edges
2) Paths formed from Non-branching and Iterating nodes/edges; iterating node is result of the process being a loop or a recursive function; iterating edge is result of parent node being iterative.
3) Paths formed from Branching and Non-iterating nodes/edges; the branching sub-paths may or may not merge back into a merger node.
4) Paths formed from Branching and Iterating nodes/edges
5) Process invoked in more than one paths

Further subsections give a glimpse on the path profiler hardware capable of detecting and identifying the above mentioned types of nodes, and path formations.

B. PATH PROFILER CIRCUITRY

a: BLOCK DIAGRAM

The path profiler has an instruction decoder and a control logic unit which takes the runtime trace signals from the microprocessor (Xilinx Microblaze [28]) as input, detects the occurrence of processes (loops and functions) and shares it with rest of the Function and Loop CAM to uniquely identify each. In this process the CAMs generate a unique tag for each process that is detected. This implementation is similar to as done in DynRP [15]. We further extend on the usage of these tag bits and include these as primary inputs for our Path Tracking Unit along with other control signals. Our DynPath implementation supports unique identification for maximum of 32 processes (16 loops and 16 functions). Fig. 4 shows the block level implementation diagram of the path profiler. This mainly includes the Instruction Decoder and Control Unit, the Tracking Unit, the Process Content Addressable Memories, the Path Invocation Count Storage, and the Path specific Process Invocation Count Storage.

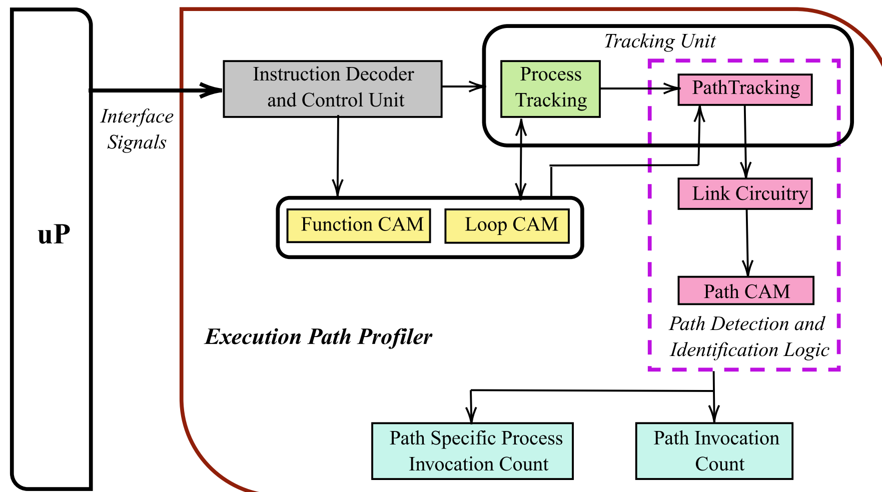


FIGURE 4. Execution path profiler block diagram.

This helps in obtaining a very detailed path profiling information. Path Detection and Identification Logic (PDIL) consists of Path Tracking Unit, the Link Circuitry, and the Path Content Addressable Memory (Path CAM) with a unidirectional flow of information. There is a separate Finite State Machine (FSM) acting as the control unit to synchronize these three modules.

1) PDIL-PATH TRACKING UNIT

The Path Tracking Unit operates concurrently with the Process Tracking Unit in the block diagram shown in Fig. 4 to detect the formation of paths while the individual processes are in execution. With the help of a sample code structure given in Fig. 1, we illustrate the differences between the proposed path profiling approach and the one given in DynRP. F1 is the first process to be invoked inside the main program. Execution of F1 is followed by a nested process L1. L1 in its multiple iterations goes through different execution paths. For initial iterations, it goes through F2, for the remaining ones it goes through F3, which in turn has F4 nested inside it. After the iterations of L1 are completed, execution returns to F1, which then calls another nested process F5. After completion of F5, code execution again returns to F1. In absence of any more process, F1 exits, and execution returns to the main program.

The sequence of execution as detected, and the output generated by *DynRP*'s function and loop tracking logic is compared with those generated by the proposed path tracker logic is captured and shown in Fig. 5. We summarize below the key design decisions which helped implement the proposed path tracking logic.

1) *Pointers:* The following two pointers, play a significant role:
   - **Current location Pointer (cells marked in green in Fig. 5)-** The address pointed to by this primary pointer increments and decrements, based on entry and exit of processes during their execution.
   - **Next location Pointer (cells marked in red in Fig. 5)–** The movement of this pointer is only in one direction as the path depth increments. The pointer is reset only when the entire path exits. It all other cases it marks the position where the fresh entry of the path should be written. The pointer also enables computation of depth of an existing path during the path switch process and helps in giving an unambiguous information related to path depth.
2) Apart from having a record of the traversed processes, in case of a deviation from the current path, or beginning of a fresh path, the following differences are implemented: (i) Copy the existing path from the tracker as an output along with the path depth (marked in yellow in Fig. 5) and raise the path switch indication. (ii) Retain the parent path information, i.e. the portion before a deviation which resulted in detection of a new sub-path. No parent information is retained if an entirely new path begins inside the main program. (iii) Update the current path pointer location with a new process tag. (iv) Reset the trailing portion of the tracking register, to avoid false data entry for a newly detected path (Cells with blue border are reset in Fig. 5).

2) PDIL-LINK CIRCUITRY

Once a path deviation is detected, it is necessary to copy the existing path in the Path Tracker to enable storage of any new path resulting from the deviation. A path switch signal is generated for the same. Link Circuitry acts as an interface between Path Tracking Unit and Path CAM. Link Circuitry primarily serves two purposes. First, it preserves the detected paths until sent for identification, to avoid any possible overwriting. This is done with help of dual temporary path register logic (TPR). Having the consideration of this design requirement enables easy enhancement to a FIFO implementation in future over having a single TPR, in case it becomes necessary from the perspective of performance.

	DynRP Tracking Logic							Path Tracking Logic					
	Tracking Register (Contains the immediate parent of the process in execution)					Output :- New Process Detected		Tracking Register (Creation of paths in runtime)					Output :- Current Path copied on detection of Path change, Size of Path
Initial State of Tracking Register	0	0	0	0	0			0	0	0	0	0	
	F1	0	0	0	0	{F1}		F1	0	0	0	0	
	F1	L1	0	0	0	{L1}		F1	L1	0	0	0	
Iteration of L1 with IF condition	F1	L1	F2	0	0	{F2}		F1	L1	F2	0	0	
	F1	L1	F2	0	0			F1	L1	F2	0	0	
Iteration of L1 with ELSE condition	F1	L1	F3	0	0	{F3}		F1	L1	F3	0	0	{F1, L1, F2}; Size=3
	F1	L1	F3	F4	0	{F4}		F1	L1	F3	F4	0	
	F1	L1	F3	F4	0			F1	L1	F3	F4	0	
	F1	L1	F3	F4	0			F1	L1	F3	F4	0	
When L1 Invocation ends and replaced by F5	F1	F5	F3	F4	0	{F5}		F1	F5	0	0	0	{F1, L1, F3, F4}; Size=4
	F1	F5	F3	F4	0			F1	F5	0	0	0	
Final Register State after code execution returns to int main	F1	F5	F3	F4	0			0	0	0	0	0	{F1, F5}; Size=2

FIGURE 5. Difference between the run-time content of the tracking units based on Fig. 1 (assuming tracking register can store up to 5 processes for illustration purpose).

Secondly, along with the path, the Link Circuitry calculates and shares the depth of path in terms of the number of Path CAM rows that specific path will occupy. The requirement for this is explained in the following subsection.

3) PDIL-PATH CONTENT ADDRESSABLE MEMORY

The Path CAM stores newly detected paths if they do not already exist in the Path CAM by comparing the newly detected path with the paths already stored in it. The Path CAM is different from the Function CAM or the Loop CAM.

One option to cater to dynamic input size is to fix the number of rows for each path assuming a maximum bound of say, 3 rows for a path. This can lead to wastage of memory capacity when a path occupies less than 3 rows. To avoid this in the proposed implementation the path configuration is chosen such that each entry can occupy multiple rows where that the number of rows is dynamically selected based on depth of each individual path, to ensure better Path CAM capacity utilization. To enable a better balance between depth of paths and maximum CAM memory utilization a constraint of a maximum of 4 rows for any path is imposed after studying several benchmark codes. Though, the implementation of the control logic for the maximum rows for any path enables parameterizing this constraint to take care of applications where there can exist paths which need more rows than allowed by the parameter. This enables easier path detection and path matching in the Path CAM logic. In the proposed implementation, path detection and path matching are carried out outside the Path CAM. Thus, the Path CAM in effect acts as a storage register file with matching happening outside of it. Table-2 highlights the primary differences between the Function/Loop CAM and the Path CAM.

a: PATH CAM CONFIGURATION

To enable the above Path CAM logic in the proposed implementation with dynamic path depth, a CAM of size 32-bit x 32-locations is chosen with following bit encoding and bit payload information:

- **Bit0:** This bit in a memory location encodes the path folding information. A value of 1 indicates start of a new path, while a value of 0 indicates folding or continuation of the path information in the earlier memory location.
- **Bit1:** Reserved bit for future use (set to 0 in present implementation).
- **Bit2-Bit31:** These 30 bits store data from the Path Tracking Unit. Considering the size of each process tag to be 5 bits wide (32 processes) as mentioned in Section - III-B, these data payload bits in a Path CAM memory location or row can contain a maximum of 6 process tags.

The above Path CAM configuration ensures using minimum number of encoding bits for path start/folding (1 bit) and maximum data payload bits used for storing path information (30 bits). It is amenable to easily accommodating any changes in the Path CAM memory parameters (for example, 64 bits/128 bits in any memory location) without needing much change in the implementation logic or information flow. Table- 4 shows how the CAM will be able to accommodate maximum of 32 paths (1 row for each path having up to 6 process tags) and minimum of 8 paths (4 rows for each path having 19 to 24 process tags). The calculation parameters for the same are first are defined in Table- 3.

TABLE 2. Key differences usual CAM v/s proposed path CAM.

Parameter	Traditional CAM used for function/loop	Path CAM
No. of Rows an entry occupies	Fixed; Typically 1	Decided dynamically based on size of entry v/s CAM row size.
Matching mode	Concurrent	Check whether CAM filled by atleast one path or not before proceeding for sequential matching.
Condition for the last entry in CAM	Based on availability of the last location of memory	Depending on number of memory rows available and how many will current path occupy/require.
Matching location	Matching happens inside CAM	CAM acts as a storage element with matching happening outside of CAM.

TABLE 3. List of path CAM parameters.

Parameter	Definition	Value
T	Total number of CAM rows	32
C	Size of one CAM row	32 bits
P	Size of a path tag	5 bits
R	Maximum number of CAM rows that can be occupied by a detected path	set the maximum limit
D	Maximum depth of path that can be accommodated. i.e. maximum number of processes possible in a path = number of processes in a nested fashion	$D = (((C - (C modulo P))/P) \times R)$ (1)
{L,U}	The lower and upper limit to the Path CAM capacity, i.e. number of paths that can be accommodated using dynamic row limit allocation	$L = ((T - (T modulo R))/R)$ (2) $U = T$ (3)

TABLE 4. Path CAM capacity calculation (refer Table-3 for parameter acronyms).

R (No. of CAM rows occupied by a path)	D (Maximum possible depth of path)	{L,U} (No. of paths possible to accommodate in Path CAM)
1	6	32
2	12	16 to 32
3	18	10 to 32
4	24	8 to 32

To implement matching outside Path CAM, the following circuitry is chosen to act as an interface between, the Common Path Register and Path CAM.

b: COMMON MATCHING REGISTER (CMR)

The common matching register (CMR), externally stores the path information of different paths, which are dynamically stored, in the memory locations of Path CAM. One way is to pull one row of Path CAM at a time, to mimic sequentially the concurrent matching that happens in the Function/Loop CAM. In this type of external sequential matching, it is necessary to align correctly, the portions of path stored for any full path in the Path CAM rows with the right section of path detected by the Path Tracking Unit. This is achieved with path encoding bits and a counter which keeps a count of the number of sectors of the Common Path Register that are filled. This information is provided by the Path Tracking Unit. Unfortunately, the sequential matching approach as described above will incur a very large latency value, considering the fact multiple rows of CAM are allowed to store information pertaining to a single path. Another constraint which needs to be met is the fact that matching of a path detected during the execution of an application needs to be completed with every existing path in the Path CAM, before the next detected path arrives. In the proposed approach matching of an entire path (1 to 4 rows) is implemented to overcome the above two problems described above for sequential matching. This is shown in Fig. 6. The proposed approach is based on a fixed matching delay allocation of delta clock cycles, after which the next detected path is taken up for matching.

For a path in the Path CAM of depth spanning four rows, each of the first bit in each row will be appropriately encoded, which when copied concurrently into the CMR will result in the first bits of each sector inheriting the same encoding from the first bits of each of the four consecutive Path CAM rows. We use these encoded sector bit values of A, B, C and D respectively, to decipher the presence of either path which span four rows, or less than four rows as shown in Table- 5 for the example given in Fig. 6.

In Table-5, C is zero in 3 cases, viz., 1, 2 and 4 respectively, out of which it belongs to the path started by A in case 1 and 2, but not in case 4. Hence a direct mapping between the row positions and the CMR sectors would yield false results. It is evident that every row position marked zero needs to be aware of previous row position values, to know where the path actually starts from and also to find read address for the next path. In case of it being part of a different path, it should not be considered for current matching. We term this technique as '*history-based masking*'.

Parallel multiplexing logic block shown in Fig. 7 interfacing the Path CAM and the CMR register implements the mentioned path matching approach. It checks whether the Path CAM rows belong to a single path; if not, it appropriately sets the sector bit values in the CMR register to zeroes using the select bit logic expressions for the multiplexers as given below. The select bits for the multiplexers are driven from

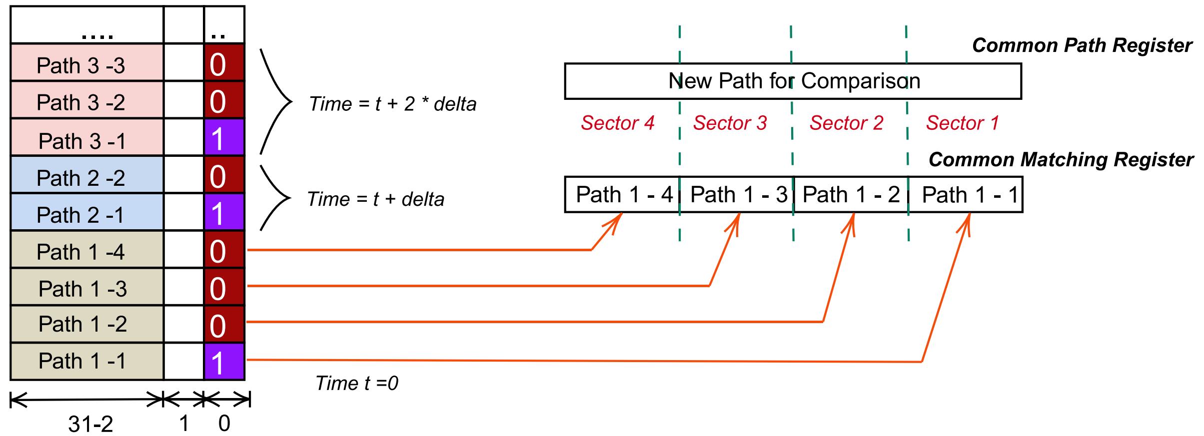


FIGURE 6. Matching logic overview for path CAM module.

TABLE 5. Decoding path folding bit combinations -2^4=16 combinations.

Case No.	A	B	C	D	Paths which the row positions represent
1	1	0	0	0	Path of depth equal to four rows starts from A.
2	1	0	0	1	Path of depth equal to 3 rows starts from A. Another path begins from the fourth row D.
3	1	0	1	0	Path of depth equal to 2 rows starts from A. Another path begins from the third row C.
4	1	1	0	0	Path of depth equal to 1 row starts from A. Another path begins from the second row B.
5	1	0	1	1	Path of depth equal to 2 rows starts from A. Two different paths start from C and D respectively, with the path from C spanning only a single row.
6	1	1	1	1	Path of depth equal to 1 row starts from A, B, C and D respectively.
...	...	...	...	...	The table would continue for total 2^3=8 combinations, for A fixed at 1.

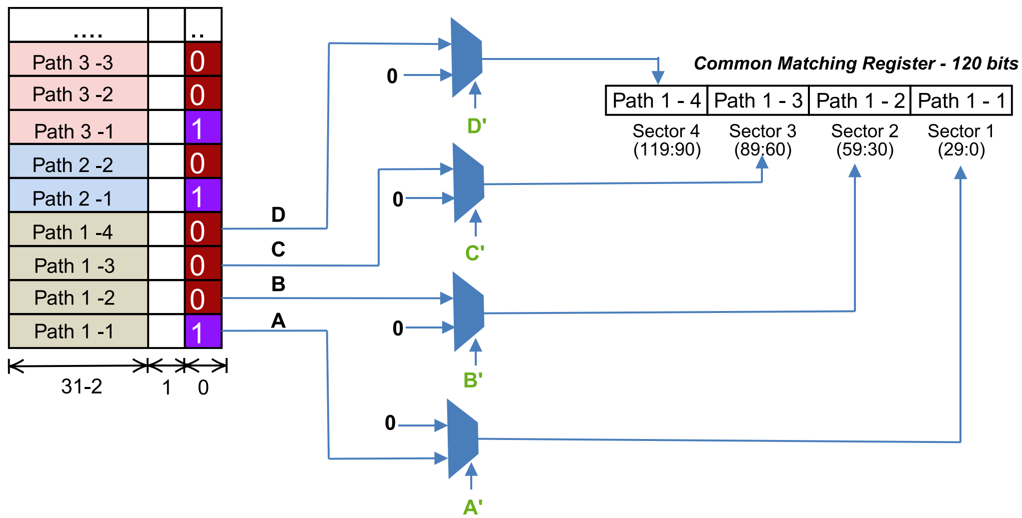


FIGURE 7. Logic to ensure valid path is loaded into match register.

combination of current and previous sector bits using the masking technique. From the entries for bits A, B, C and D shown in Table- 5, the boolean logic gate expression for each of the multiplexer select line A', B', C' and D' are easily derived and are given by eq. (4) to (7).

$$A' = A \tag{4}$$

$$B' = B \tag{5}$$

$$C' = B|C \tag{6}$$

$$D' = B|C|D \tag{7}$$

Assuming the current pointer at A is the *read_addr* pointer value, the address for the first row of next path based on current path depth becomes (refer (8) to (11)):

$$\begin{aligned} new_read_addr &= read_addr + 1, \\ &\quad if\ A.B(where, \mathbf{ABCD = 11xx}) \end{aligned} \tag{8}$$

$$\begin{aligned} new_read_addr &= read_addr + 2, \\ &\quad if\ A.(notB).C(where, \mathbf{ABCD = 101x}) \end{aligned} \tag{9}$$

$$new_read_addr = read_addr + 3, \quad if\ A.(notB).(notC).(D)(where, \mathbf{ABCD = 1001}) \tag{10}$$

$$new_read_addr = read_addr + 4, \quad if\ A.(notB).(notC).(notD)(where, \mathbf{ABCD = 1000}) \tag{11}$$

The encoding for the new set of sector bits A, B, C and D respectively for the next path matching are given by (12) to (15).

$$A = cam[new_read_addr][0] \tag{12}$$

$$B = cam[new_read_addr + 1][0] \tag{13}$$

$$C = cam[new_read_addr + 2][0] \tag{14}$$

$$D = cam[new_read_addr + 3][0] \tag{15}$$

4) PATH SPECIFIC PROCESS INVOCATION COUNT

Referring to the objectives laid out in Section- III-A, the above subsections dived deeper into the path identification process. The goal of this module is to further gain an understanding on the contribution of each process towards the paths that it is a part of. The hardware-based profilers [11], [12], [15] discuss on calculating the total invocation count of various processes executing in a dynamic context. The objective of this module is to gather data on the number of times a process is respectively invoked in the active paths of execution. The block diagram for the same is shown in Fig. 8 and described as follows. The details are made in reference to Section- III-B1. Corresponding to the event of path traversal being recorded in the Path Tracking Register labelled as Register-1, the invocation counts for each specific process in the active path are recorded in a parallel register labelled as Register-2. Once the path is identified with the help of Path memory logic, the data in this path specific process invocation count (PSPIC) register is forwarded to the PSPIC memory. The number of rows in the PSPIC memory are made equal to the maximum number of unique process tags that a profiler architecture can support (32 in this case). Each row is further sectioned into maximum number of paths that can be uniquely identified by the profiler (32 in this case). Upon receiving the information from the Register-1 and Register-2, the invocation count for the processes of the path are respectively accumulated in the corresponding path tag section.

The following design considerations ensure correctness of the information being stored in PSPIC memory hardware:-

- *PSPIC memory configuration–* Every PSPIC row is divided into 32 sections which is equal to the total number of paths supported by the architecture. This is because one process could be a part of multiple paths. Though not all processes will be part of multiple paths, this is done to ensure that a static hardware resource allocation considers this possibility for PSPIC memory, unlike a dynamic memory allocation that can be done in software-based implementation.
- *Invocation count for common parent processes-* The above-described implementation could lead to a false increment in the parent process invocation count if a sub-path (and hence the entire path) is invoked multiple times (eg. AES benchmark, where the sub-paths to common parent path are invoked in an iterative sequence). This can also lead to loss of parent invocation attribution in PSPIC memory in case of branching sub-paths. To avoid this, Branching Pointer as mentioned in Fig. 8 is introduced. This stores the index from which the current sub-path branched out or if a new path originated. This ensures that the PSPIC is attributed to all the paths that a parent process is part of. It also helps avoid false attribution to parent counts, when a path is repeatedly invoked due to an iterative branching node in it.

Design for PSPIC as shown in Fig.8 ensures least latency for a dynamic context. Every cell of PSPIC Memory contains a separate cell selection logic based on path number and process number, a register for invocation count accumulation, an adder and a comparator, and no resource sharing or Block RAMs. This ensures 1-clock cycle latency for every module in the profiler. This results in high utilization of the CLB LUTs. There is a trade-off between lower clock cycle latency and resource utilization which is discussed in detail in Section-IV. An alternate implementation as shown in Fig.9, implements resource sharing with a net PSPIC clock latency of 3 clock cycles. This is higher as compared with the above implementation resulting in 1 clock cycle latency. However, it is still within acceptable limits for our current path profiling implementation and is seen to result in lower resource utilization.

IV. RESULTS

A. PATH IDENTIFICATION, PATH INVOCATION COUNT AND PATH SPECIFIC PROCESS INVOCATION COUNT

Paths identified in real time for the different examples listed in Table-6 using the proposed approach are in conformance with the description of the calling hierarchy for a process and their process invocation counts determined and found offline with help of DynRP and GProf. The function description in the actual benchmark codes (CRC, BLIT, BCNT and AES-Encrypt) corresponding to the function number, can be found in Table-7, which is same as mentioned in DynRP [15]. The process invocation counts and DynPath based path structures follow the results as shown in Table-1 of the DynRP paper. Here *_modsi3* and *_divsi3* are Microblaze specific functions used to perform certain floating point mathematical operations as present in the AES-Encrypt code. Such platform specific functions executing in a run-time context are also detected and identified by our profiler. Loops however have no identifiers in C semantics, hence the description is not mentioned in the table for the same. For the sample exhaustive code structure given in Fig.3, the intention was more on

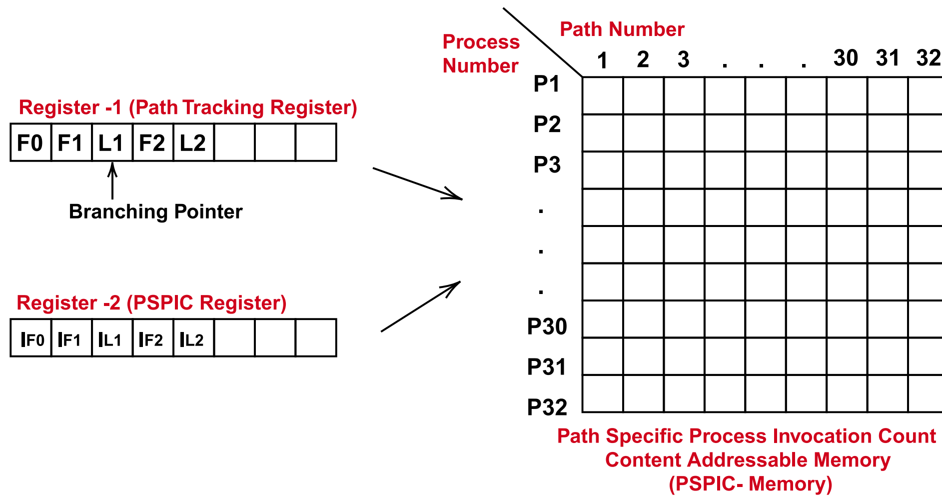


FIGURE 8. Flow diagram- path specific process invocation count.

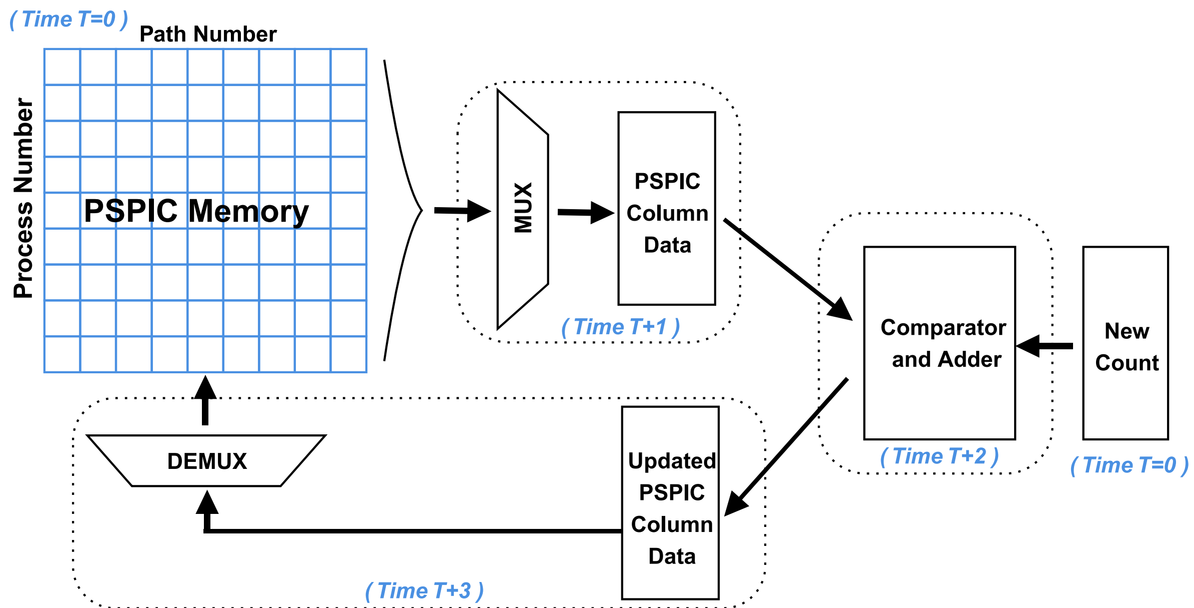


FIGURE 9. PSPIC implementation with resource sharing.

creating exhaustive path structures (Section-III-A) to validate the path profiler behaviour, than achieving a specific code functionality. Therefore function name/description is not applicable.

Below, we discuss the path profiling results obtained for the sample exhaustive case Fig.3 using the proposed approach and in context with other benchmark codes. Section IV-B analysis the comparison with GProf.

From the abstract representation of code structure for the exhaustive sample code depicted using a directed graph and from the results shown in Table-6, it is clear that F1 is the primary parent node inside the main function/program, within which the entire application code is described. This is similar to the benchmark code AES-Encrypt, where the entire code is inside the function named *aes_main* denoted as F1. For the sample code, F1 calls a nested function F2, which then branches into several sub-paths inside it due to branching conditions. These sub-paths are described below.

- If the branching condition in F2 is satisfied, the execution goes through the first nested sub-path Path-1 comprising of processes L1, F3, and L2 respectively; else it executes a portion of application code inside F2 which does not have any process (i.e. either a function or a loop) inside it excepting for some sequential program statements. Based on the branching parameters the first nested sub-path executes. On completion of execution of Path-1, execution continues in the merger node L3 inside F2. Path-1 created by the Path Tracker Unit is then written into Path CAM with its invocation count set to 1. The Path Tracker Unit then detects the second path, i.e. Path-2 comprising of the processes F1, F2 and L3, respectively. Subsequent to its completion it gets written

TABLE 6. Results for path identification and path invocation count.

Code	Total number of Paths executing in dynamic context	Paths Detected and Identified		Path Invocation Count
Sample code (Fig. 3)	12	Path -1	F1, F2, L1, F3, L2	1
		Path -2	F1, F2, L3	1
		Path -3	F1, F4	1
		Path -4	F1, L4, F5	1
		Path -5	F1, L4, F6	1
		Path -6	F1, L5, F8	20
		Path -7	F1, L5, F9	20
		Path -8	F1, F10	1
		Path -9	F1, F11	1
		Path -10	F1, L6, F11	1
		Path -11	F1, F12, F10	1
		Path -12	F13	1
CRC	2	Path -1	F1, L1, F2, L2	1
		Path -2	F1, L3	2
BLIT	2	Path -1	F1, L1	1
		Path -2	F1, L2	1
BCNT	1	Path -1	L1	1
AES-Encrypt	11	Path -1	F1, F2, F3, L1, L2	1
		Path -2	F1, F2, F3, L3, F4	50
		Path -3	F1, F2, F3, L3, F5	20
		Path -4	F1, F2, F3, L3, F6	10
		Path -5	F1, F2, F3, L3, L5	40
		Path -6	F1, F2, F7, L6	2
		Path -7	F1, F2, L7, F8	9
		Path -8	F1, F2, L7, F9, L8	9
		Path -9	F1, F2, L7, F9, L9	9
		Path -10	F1, F2, F8	1
		Path -11	F1, F2, L10	1

into Path CAM with an invocation count of 1 after being ascertained that it is not present in the Path CAM. The above two path structures seen in the sample code is similar to the two paths, Path-1 and Path-2, respectively, seen in the benchmark code, CRC, excepting that Path-2 is invoked twice due to F1 being invoked twice by the main function and the "else" condition being satisfied in the second invocation. Continuing with the description of path formation in the sample code, it can be observed that F4 is invoked inside F1 as a level node to F2 (refer to terminology, Section III-A). In a similar manner L4 is a level node to F4. L4 has branching sub-paths F5, F6 and F7 within it. With the dynamic inputs deciding on the branching conditions, first Path-4 gets invoked which is then followed by invocation of Path 5 for the remaining iterations of L4. Process F7 however never gets invoked for the given input values. This case is similar to AES-Encrypt where one of the processes i.e. L4 of AES-Encrypt never gets invoked [15] and is pointed out while explaining GProf results.

- Next, we have Path-6 and Path-7 formed due to iterations of L5. It is interesting to note that the invocations counts of these paths are equal to iteration count of L5 as shown in Table-6, hence greater than 1, which is unlike invocation counts for Path-4 and Path-5, respectively, as described above. This observation is due to F8 and F9 being *level* nodes within L5. Hence both the paths get invoked multiple times depending on iterations of L5.
- The processes F10 and F11 in the Path-8 and 9 respectively are found invoked in multiple paths (i.e. path 10 and 11). This is the case as referred to in Section III. Their path specific process invocation counts are shown in Table-7. This case is similar to AES-Encrypt, which includes a child process, viz., function F8, which is called in two different paths, once due to being nested inside L7 (Path-7) and another due to F2 (Path-10) as shown in Table-6. DynPath here addresses the ambiguity of contribution of F8 to each of the paths and respective parent process execution count, through the PSPIC feature. In reference to the total process execution count profiled by DynRP for function F8(=5210) and other parent processes (refer results section from [15] for counts), the static analysis if done shows that F8 (inside L7) should be = 5361(L7) - 366(L1 only) - 306(F9) = 4689. Here, 4689 is also 9/10th of total 5210 which implies F8 must be invoked nine times in L7 and once inside F2. This is exactly as found out through DynPath's PSPIC feature in Table-7, thereby providing further granularity and resolving ambiguity.
- F10 in the sample code is a recursive function. The recursive nature of F10 can be identified using *Gprof* [21]. However, this feature is also inferred from its higher invocation count (as shown in Table-7), despite its not being present inside an iterating parent loop (the path invocation count is seen to be equal to 1 in Table-6).

Once the execution of the application code in Path-11 completes, it continues to execute by invoking the neighboring level process node F13, which is in the second level of the process hierarchy. The application code execution finally completes after F13 completes its execution.

TABLE 7. Path specific process invocation count v/s total process invocation count.

Code	Process	Path	Path Specific Process Invocation Count	Total Process Invocation Count
Sample code (Fig. 3)	F1	Path 1 to 11	Invoked once being the parent node	1
	F2	Path 1 and 2	Invoked once being the parent node	1
	L1	Path 1	1	1
	F3	Path 1	5 (=iteration count of L1)	5
	L2	Path 1	5 (=iteration count of L1)	5
	L3	Path 2	1	1
	F4	Path 3	1	1
	L4	Path 4 and 5	Invoked once being the parent node	1
	F5	Path 4	6 (= no. of times condition met in iterations of L4)	6
	F6	Path 5	6 (= no. of times condition met in iterations of L4)	6
	F7	Not executed in dynamic context		
	L5	Path 6 and 7	Invoked once being the parent node	1
	F8	Path 6	20 (=iteration count of L5)	20
	F9	Path 7	20 (=iteration count of L5)	20
	F10	Path 8	9	18
		Path 11	9	
	L6	Path 10	1	1
	F12	Path 11	1	1
	F11	Path 9	1	17
		Path 10	16 (=iteration count of L6)	
	F13	Path 12	1	1
CRC	F1 (*icrc*)	Path 1 and 2	Invoked twice from int main	2
	L1	Path 1	1	1
	F2 (*icrc1*)	Path 1	256 (=iteration count of L1)	256
	L2	Path 1	256	256
	L3	Path 2	2	2
BLIT	F1 (*blit*)	Path 1 and 2	2 (invoked twice from int main)	2
	L1	Path 1	1 (invoked in first invocation of F1)	1
	L2	Path 2	1 (invoked in second invocation of F1)	1
BCNT	L1	Path 1	1 (int main contains only 1 primary loop L1 which is invoked once)	1
AES-Encrypt	F1 (*aes_main*)	All paths	1 (invoked once being the parent node)	1
	F2 (*encrypt*)	All paths	1 (invoked once being the parent node)	1
	F3 (*KeySchedule*)	Path 1 to 5	1 (invoked once being the parent node)	1
	L1	Path 1	1	1
	L2	Path 1	2	2
	L3	Path 2, 3, 4 and 5	1 (invoked once being the parent node)	1
	F4 (*_modsi3*)	Path 2	80	80
	F5 (*SubByte*)	Path 3	40	40
	F6 (*_divsi3*)	Path 4	10	10
	L4	Not executed in dynamic context		
	L5	Path 5	40	40
	F7 (*AddRoundKey*)	Path 6	2	2
	L6	Path 6	2	2
	L7	Path 7, 8 and 9	1	1
	F8 (*ByteSubShiftRow*)	Path 7	9	10
		Path 10	1	
	F9 (*MixedColumn_AddRoundKey*)	Path 8 and 9	9 (Child of an iterating parent- L7; Contains *level* child nodes L8 and L9 resulting in two paths- Path 8 and 9)	9
	L8	Path 8	9 (= invocation count of F9)	9
	L9	Path 9	9 (= invocation count of F9)	9
	L10	Path 11	1	1

B. COMPARISON WITH CALL GRAPH RESULTS FROM Gprof

Table 8 shares the call graph results generated by GProf [21] for the exhaustive sample case given in Fig.-3. As mentioned, GProf gives the calling hierarchy for every function that is executed based on the code logic and dataset. Column-I shows the count of 12 though the total number of functions is 13. This is because function F7 is not executed for the given dataset. Thus, F7 too is not detected by DynPath which is intended for hardware-based profiling in a dynamic context. Column-III gives the call hierarchy with respect to every executed function. The function in consideration is shown with a left alignment, whereas its parent and child nodes which are shown with an indentation in GProf's call graph, are shown in the table with a right alignment. Column-II gives the ratio of the number of times the parent invokes the function under consideration or the function under consideration as a parent invokes the child function. In contrast to GProf, DynPath includes loops as well. It generates output in terms of Path Detection and Identification, Total Number of Paths,

TABLE 8. GProf call graph output - sample code (Fig.3).

Code	Sr. No.	Called	Function name	Function Name (Invocation Count)
Sample code (Fig. 3)	**1**	20/20	F1	F8 (20)
		20	F8	
	2	20/20	F1	F9 (20)
		20	F9	
	3	6/6	F1	F5 (6)
		6	F5	
	4	6/6	F1	F6 (6)
		6	F6	
	5	5/5	F2	F3 (5)
		5	F3	
	6	16	F10	F10 (1)
		1/2	F12	
		1/2	F1	
		2+16	F10	
		16	F10	
	7	1/1	main	F1 (1)
		1	F1	
		20/20	F8	
		20/20	F9	
		6/6	F5	
		6/6	F6	
		1/1	F2	
		1/1	F4	
		1/2	F10	
		1/1	F11	
		1/1	F12	
	8	1/1	F1	F11 (17)
		1	F11	
	9	1/1	F1	F12 (1)
		1	F12	
		1/2	F10	
	10	1/1	main	F13 (1)
		1	F13	
	11	1/1	F1	F2 (1)
		1	F2	
		5/5	F3	
	12	1/1	F1	F4 (1)
		1	F4	

Invocation Count of each Path and Path Specific Process Invocation count in a dynamic context, without the associated run-time overhead of code instrumentation. These can be deduced or inferred offline by analysing the outputs generated by GProf. However, these are not explicitly presented as results by GProf. Additionally, F11 is part of 2 paths, Path-9 and Path-10, both having F1 as the common parent node. DynPath captures this granularity and uniquely identifies each path along with calculating F11's PSPIC specific to each path. This information cannot be obtained through GProf as the Path-9 is generated through a Loop-L6 inside F1 itself. One can see that the invocation count for all the functions obtained using DynPath matches with those generated GProf, except for F10. F10 is invoked twice, once inside F1 and once inside F12. For every alternate invocation, it iteratively invokes itself 8 times. This can be deduced by studying the GProf call graph. The total invocation count for F10 = (2+16) as shown in the call graph by GProf is captured by DynPath as 18. The recursive nature and multiple parents are however captured by the dynamic paths and path-specific invocation count of 9 each. We do not store "main" as the root function in DynPath, as mentioned in some of the calling

TABLE 9. GProf call graph output - Benchmarks.

Code	Sr. No.	Called	Function name	Function Name (Invocation Count)
CRC	**1**	256/256	F1	F2 (256)
		256	F2	
	2	1/1	main	F1 (2)
		2	F1	
		256/256	F2	
BLIT	**1**	2/2	main	F1 (2)
		2	F1	
BCNT	N.A.			
AES-Encrypt	**1**	40/40	F3	F5 (40)
		40	F5	
	2	10/10	F2	F8 (10)
		10	F8	
	3	9/9	F2	F9 (9)
		9	F9	
	4	2/2	F2	F7 (2)
		2	F7	
	5	1/1	F2	F3 (1)
		1	F3	
		40/40	F5	
	6	1/1	main	F1 (1)
		1	F1	
		1/1	F2	
	7	1/1	F1	F2 (1)
		1	F2	
		10/10	F8	
		9/9	F9	
		2/2	F7	
		1/1	F3	

hierarchies in GProf. This is because in case of C/C++ semantics, every execution path is expected to launch from "main." This information, being redundant, the hardware does not store it for efficient use of memory.

Similarly, Table 9 shows results for the CRC, BLIT, BCNT and AES-Encrypt benchmarks. Only functions and not loops are detected hence the process-specific call graph varies in comparison to Table-6 and 7 accordingly. For AES-Encrypt, F4 and F6 as mentioned in Table-7 are not detected by GProf, as these are platform specific functions used for floating point operations, already discussed in Section-IV-A.

C. RESOURCE UTILIZATION RESULTS

Table- 10 (Column 1 to 3) shows the resource utilization summary of the Verilog based execution path profiler as shown in Fig. 4, for the Xilinx Kintex Embedded Kit (XC7K325T-2FFG900C) [29]. The higher resource utilization count is attributed to the CLB LUT based implementation of the design logic of the Process (Function and Loop) CAM, Path memory and the PSPIC memory and with no resource sharing (refer Section-III-B4). The Xilinx Block RAM based memory implementation is not chosen for any of the profiler memories in the context of this paper. This is due to the limitation on the number of read and write ports allowed with a Block RAM and the latency costs associated with every read and write operation [15]. This is especially true for PSPIC memory where the existing invocation counts for all the processes in an identified path are to be incremented based on new counts. With the hardware circuit fixed to cater maximum of

TABLE 10. Resource utilization summary (XC7K325T-2FFG900C).

Resources	Available	No Resource Sharing		With Resource Sharing		% reduction with resource sharing
		Path Profiling Logic	Utilization (%)	Path Profiling Logic	Utilization (%)	
Slice LUTs	203800	178276	87.48	4792	2.35	94.76
Slice Register	407600	12221	3.00	11137	2.73	9.00
F7 Mux	101900	25193	24.72	572	0.56	97.73
F8 Mux	50950	903	1.77	120	0.24	86.44

24 processes in a path as explained in Section III, for any given path identified, the run-time execution path profiler will have to accommodate a high net latency of 24*(read cycle latency + write cycle latency + registering latency for the combinational (increment) logic) as will be incurred by the PSPIC Block RAM operations. Similar is true for the Process CAM and the Path memory where the Block RAM or a resource sharing based design will comparatively incur a greater number of clock cycles in a dynamic context than a CLB LUT based. The tradeoff however is a high utilization of the CLB LUTs as in Table- 10 (Column-3).

Table-10 (Column-4 and 5) summarizes resource utilization for an implementation of our proposed design based on resource sharing. PSPIC without resource sharing consumes 90% of the total profiler area in terms of number of FPGA Logic cells (as reported by the Xilinx Vivado Tool). Hence, PSPIC implementation with resource sharing, has a significant effect on reducing the overall Profiler resource utilization. Resource sharing brings down PSPIC area by 96% and overall profiler area by 87%. Hence, Table-10 reaffirms the tradeoff between latency and resource utilization, in context from Section-III-B4. In FPGAs, the resource utilization is dependent on to the kind of logic cells generated by the Vivado synthesis tool for a chosen Xilinx FPGA family implementing the two PSPIC versions. Hence, the exact underlying resource utilization and selection of logic cells by the tool for synthesis could vary with the FPGA device family chosen.

a: NET PROFILER LATENCY

Following is the clock cycle latency for each module once a loop/function is detected. The detection flag is raised in the next clock cycle itself:- (1) Process Identification and Tracking Path formation - 3; (2) Path CAM - once path change detected - 1 or 2 - depends on whether Path CAM is empty or filling; (3) PSPIC - 1 or 3 - depends on whether resource sharing is enabled. Hence the latency of the proposed path profiler lies in range of 5 to 8 clock cycles.

V. CONCLUSION

The paper discusses design objectives and implementation details of a non-intrusive hardware-based path profiler for modern embedded systems. This though discussed in the context of FPGA, can be used in both static and dynamic profiling scenarios for any target platform as discussed in Section I and Section V-A. The implementation focuses on realizing the path profiler using well known basic digital logic blocks like multiplexers, gates and registers and considering the possibility of easy extensibility and scalability to include more features. The design of the Path CAM matching logic block focuses on the objective of maximizing the matching of a newly detected path, during execution of the application code with each path stored in the Path memory within fewer clock cycles through a concurrent approach. This enables multiple rows corresponding to a single path in the Path memory to be matched concurrently with a new path detected by the Path Tracking Unit, thereby reducing the overall path matching latency of the Path CAM. The proposed approach considers the feature of dynamic depth of a path when stored in the Path CAM to result in its optimal utilization with respect to its memory locations.

A. FUTURE SCOPE

This profiler with relevant modifications carries good scope for real-time applications. The scope and enhancements are discussed herewith.

1) MULTICORE PROCESSING

This profiler is currently implemented for single-core processing system. The profiler can be replicated, to profile individual cores in a multicore system. It can then be extended to profile for a complete heterogeneous platform containing multiple multicore or single-core processors and hardware accelerator IPs or reconfigurable arrays. The intervention of the operating system (OS) will play a crucial role in such an implementation. The profiler must be designed to communicate with the OS/Job Scheduler [18], [30] and the underlying communication protocols and interrupt mechanisms.

2) DYNAMIC RECONFIGURATION

The profiler extension to heterogeneous platform will further aid its implementation to enable actual dynamic reconfiguration, as pointed to in Section-I. Based on the literature, the components required in an actual dynamically reconfigurable system, namely,

1) Reconfigurable Architectures [30], [31], [32], [33], [34]- Availability of architectures that can dynamically reconfigure in less time.
2) Software / Operating System (OS) Support [9], [17], [18], [19], [30], [35]- To enable seamless execution

between a processor and reconfigurable architecture in a system, for dynamic resource allocation and swapping configurations between computing elements.
3) Technology Support [30], [36], [37], [38] - Enables faster dynamic reconfiguration.
4) Profiling [11], [15], [20], [21]– To provide support to the Software/OS with both offline and run-time statistics and decide whether to reconfigure dynamically and which specific process. Specifications for the same are well defined in [15].

Most of the current versions of DPR implementations are based on having static profiling-based analysis of application codes. To aid optimal hardware utilization, the configurations are dynamically loaded based on their turn of invocation. However, there is no room for further dynamic optimization based on real-time bottlenecks. Point 4, i.e. profiling process invocation frequency and inter-relationships and if done dynamically can help provide a holistic view to the Software support unit to take a decision and execute the DPR flow.

Following are a some of the use cases from the existing literature, where the proposed integration with our profiler could be worked out in future. (1) Warp processor [35] is proposed on similar lines. The profiler used in it calculates run-time bottlenecks based on invocation frequency of executing loops. Our proposed profiler can replace this to consider both functions, loops and the execution path. (2) ReConOS [17] aids dynamic reconfiguration between hardware and software on thread-level for design space exploration. This can be extended to support profiling information received from our profiler for even better decision making. (3) SysteMorph [32] is proposed as a conceptual system for dynamic adaptive optimization for application programs whose threads of behaviours change dynamically over time based on external or environmental inputs. It performs hot path profiling to find a part of the instruction sequence in an application program which is frequently executed. This is then dynamically reconfigured using a smart hardware. Beyond instruction-based branch path creation, the granularity of the scope to which the instructions belong in term the functions and loops, their hierarchy, dynamic execution path, path specific process invocation count could further help refine this proposed system. Due to current technical constraints these proposed integration methods on dynamic reconfiguration are not a part of this paper.

3) DYNAMIC VALUE PREDICTION

Apart from this, Dynamic Information Flow Analysis [4] as referred to in Section-I is the key to performing a better value prediction for the current generation microprocessors [4], [6]. The value prediction of a variable can either be based on local value locality (previous values of the same variable, e.g. $x = x+5$) or global value locality (inter-dependency on other variables e.g. $x = y + 2z + 3$) [4], [7]. This further depends on the logic statements and processes the variables are part of, the branching affecting the loop and function invocations, the number of loop iterations, and the execution path that influences its local and global value locality.

Having a view of flow of control and data through the dynamic execution paths will help understand and leverage the global locality to enable even more accurate value prediction in high-end multi core server processors. Non-intrusive monitoring will aid in accurately carrying out value prediction even if a piece of executing code gets swapped between computing elements in a DPR scenario.

Given the current granularity regarding the relationship between various loops, functions and execution paths occurring in a dynamic context, this information can prove to be extremely helpful in profiling the factors that dynamically influence the value of a variable. This can be done by tracking the functions, loops, logic statements and execution paths that a variable (and the globally influencing variables) is a part of.

4) OTHER FEATURE ENHANCEMENTS

Following feature enhancements, if done, to the current version of the profiler, will further help its run-time implementation for the above stated advanced application areas. FIFO modules can be added to the design from timing synchronization point of view. Replacement policies can be implemented for Path CAM / Count Storage RAM as implemented in [12]. There could be pointers to parent paths common to multiple sub-paths in the Path CAM, instead of storing the entire parent path to each sub-path. This will be efficient on memory. However, this can increase the complexity of the path decoding logic. The total path encoding will require separate encoding for the common parent sub-path and the different branching sub-paths in all the paths resulting from the conditional node in the parent sub-path. The trade-off hence will have to be analyzed. Another trade-off could be with sequential v/s concurrent Path CAM matching logic. The current implementation matches the paths stored in the Path CAM sequentially, as done externally. Concurrent path matching of a newly detected path with every path stored in the Path Memory could be needed to meet stringent timing constraints for some applications. While this approach can result in very efficient timing performance, it will require a relatively complex matching logic, primarily due to the feature of the path storage in the Path CAM with respect to path depths being dynamic.

APPENDIX
SOURCE CODE

More details to the Design flow, architectural details, test case generation and working with FPGA, for DynPath and Poster PDF for [5] are uploaded to GitHub (*https://github.com/manp-git/DynPath*).

REFERENCES

[1] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design-Modelling, Synthesis and Verification*. Dordrecht, The Netherlands: Springer, 2009.

[2] N. M. C. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Dynamic partial reconfiguration of customized single-row accelerators," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 1, pp. 116–125, Jan. 2019.

[3] N. Sumeet, D. Deeksha, and M. Nambiar, "HLS_Profiler: Non-intrusive profiling tool for HLS based applications," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, New York, NY, USA, Jul. 2022, pp. 187–198.
[4] W. J. Ghandour, H. Akkary, and W. Masri, "Leveraging strength-based dynamic information flow analysis to enhance data value prediction," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 1–33, Mar. 2012.
[5] M. K. Jaswal, S. K. Roy, and N. Rao, "Performance enhancement of RISC-V cores through value prediction based on dynamic data flow analysis," presented at the RISC-V Workshop, ETH Zurich, Jun. 2019. [Online]. Available: https://video.ethz.ch/events/2019/risc-v/18edc0cf-3b2a-4292-90fe-090b438f26bb.html
[6] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 428–439.
[7] S. Mittal, "A survey of value prediction techniques for leveraging value locality," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 21, p. e4250, Nov. 2017.
[8] C. Wang, W. Lou, L. Gong, L. Jin, L. Tan, Y. Hu, X. Li, and X. Zhou, "Reconfigurable hardware accelerators: Opportunities, trends, and challenges," 2017, *arXiv:1712.04771*.
[9] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proc. IEEE*, vol. 103, no. 3, pp. 332–354, Mar. 2015.
[10] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt, "Profiling tools for hardware/software partitioning of embedded applications," in *Proc. ACM SIGPLAN Conf. Lang., Compiler, Tool Embedded Syst. (LCTES)*, New York, NY, USA, 2003, pp. 189–198.
[11] N. P. Kumar, "An efficient FPGA-based profiling tool for hardware/software partitioning of embedded system applications," M.S. thesis, Int. Inst. Inf. Technol. Bangalore, India, 2016.
[12] A. Nair, K. Shankar, and R. Lysecky, "Efficient hardware-based nonintrusive dynamic application profiling," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 3, pp. 1–22, May 2011.
[13] J. G. Tong and M. A. S. Khalid, "Profiling CAD tools: A proposed classification," in *Proc. Internatonal Conf. Microelectron.*, Dec. 2007, pp. 253–256.
[14] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, Jun. 2002.
[15] M. K. Jaswal and S. K. Roy, "DynRP—Non-intrusive profiler for dynamic reconfigurability," in *Proc. 24th Int. Symp. VLSI Design Test (VDAT)*, Jul. 2020, pp. 1–6.
[16] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, R. W. Hartenstein and M. Glesner, eds. Berlin, Germany: Springer, 1996, pp. 327–336.
[17] *Architecture of Reconos*. Accessed: 2022. [Online]. Available: http://www.reconos.de/documentation/architecture/
[18] O. Knodel, P. Lehmann, and R. G. Spallek, "RC3E: Reconfigurable accelerators in data centres and their provision by adapted service models," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2016, pp. 19–26.
[19] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "Hthreads: A hardware/software co-designed multithreaded RTOS kernel," in *Proc. IEEE Conf. Emerg. Technol. Factory Autom.*, vol. 2, Sep. 2005, p. 8.
[20] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *Proc. ACM/SIGDA 12th Int. Symp. Field Program. Gate Arrays (FPGA)*, New York, NY, USA, 2004, pp. 190–199.
[21] *GNU Prof*. Accessed: 2022. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
[22] *Perf-Intel-pt(1)—Linux Manual Page*. Accessed: 2022. [Online]. Available: https://www.man7.org/linux/man-pages/man1/perf-intel-pt.1.html
[23] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, Mountain View, CA, USA, Apr. 2022.
[24] *Chstone Benchmark Suite*. Accessed: 2020. [Online]. Available: https://github.com/davestevens/LE1/tree/master/benchmarks/legup/chstone/aes/
[25] *Powerstone Benchmark Suite*. Accessed: 2020. [Online]. Available: https://github.com/li-qingan/powerstone/
[26] D. B. West, *Graph Theory*, 2nd ed. London, U.K.: Pearson Education, 2002.
[27] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical call graph generation in Python," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1646–1657.
[28] *Microblaze Soft Processor Core*. Accessed: 2016. [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html/
[29] *Xilinx Kintex-7 FPGA Embedded Kit*. Accessed: 2021. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/dk-k7-embd-g.html
[30] M. Belwal, M. Purnaprajna, and T. S. B. Sudarshan, "Enabling seamless execution on hybrid CPU/FPGA systems: Challenges directions," in *Proc. 25th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2015, pp. 1–8.
[31] L. Bozzoli and L. Sterpone, "ReM: A reconfigurable multipotent cell for new distributed reconfigurable architectures," in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, eds. Cham, Switzerland: Springer, 2019, pp. 295–304.
[32] N. Yoshimatsu, M. Yoshida, T. Soga, M. Shuto, Y. Tanoue, Y. Fujii, K. Eshima, T. Hayashida, and K. Murakami, "SysteMorph: Dynamic/online/adaptive system-level optimization for SoC," in *Proc. 7th Int. Conf. High Perform. Comput. Grid Asia Pacific Region*, 2004, pp. 442–447.
[33] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Model. Simul. (SAMOS)*, Jul. 2016, pp. 235–244.
[34] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surveys*, vol. 52, no. 6, pp. 1–39, Oct. 2019.
[35] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 659–681, Jun. 2004.
[36] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," in *Proc. 9th Annu. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 147–159.
[37] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *IEE Proc. Comput. Digit. Techn.*, vol. 153, no. 3, pp. 157–164, May 2006.
[38] N. Charaf, C. Tietz, and D. Goehringer, "MaNaBIT: A versatile tool for manipulating and analyzing FPGA bitstreams," in *Proc. IEEE 30th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2022, p. 1.

MANPREET KAUR JASWAL received the B.Tech. degree in electronics and communication engineering, in 2014, and the M.Tech. degree in VLSI design, in 2016.

She has been an Intern at the Centre for Development of Advanced Computing (2012), Indian Space Research Organization (2014), and the Semiconductor Laboratory, India (2015–2016). She is currently a Ph.D. Scholar in the domain of VLSI systems design at the International Institute of Information Technology, Bengaluru. Her research interests include hardware software co-design, non-intrusive profiling mechanisms, dynamic reconfiguration, micro-architectural techniques for hardware acceleration, and design for space applications.

SUBIR KUMAR ROY received the B.E. degree from the University of Pune, in 1982, the M.Tech. degree from IIT Madras, in 1984, and the Ph.D. degree from IIT Bombay, in 1993.

In 1993, he worked at Semiconductor Complex Ltd., Chandigarh, and the Department of Computer Science and Engineering, VLSI Design Centre, IIT Bombay. From 1993 to 2001, he was with the Faculty of Electrical Engineering, IIT Kanpur. From 2001 to 2003, he was with Synplicity Inc., Sunnyvale, USA, and Bengaluru. From April 2004 to January 2013, he was with Texas Instruments India, Bengaluru, working in the area of hardware formal verification. Since April 2013, he has been with the International Institute of Information Technology, Bengaluru. He spent two years, from 1998 to 2000, carrying out research on formal verification at Fujitsu Laboratories Ltd., Kawasaki, Japan, on a sabbatical from IIT Kanpur. His research interests include hardware formal verification, power estimation, performance analysis, CAD for VLSI and embedded systems, testability, and design for testability.