An EDF Scheduling Algorithm for Aperiodic Tasks in a Heterogenous SoC

Stephen Donchez, *Member, IEEE*

***Abstract*—Earliest Deadline First (EDF) scheduling is a mechanism by which discrete tasks in a system are scheduled based on their respective deadlines. This algorithm is not constrained solely to its traditional application in the realm of scheduling tasks on processors, but is suitable for use in any scenario wherein an ideal arrangement of tasks is needed that respects their time constraints. In this work, a modified EDF algorithm is applied to schedule aperiodic tasks among a number of Intellectual Property (IP) cores contained on a Field Programmable Gate Array (FPGA). Special consideration is given in the modification of the algorithm to the nature of tasks suited for execution on an FPGA, as well as the common properties of such IP cores as they impact the schedulability of these tasks.**

This work first outlines the algorithm described above, then implements it in C++ and demonstrates its execution on both a desktop computer and on a heterogenous System on a Chip (SoC) device. It also discusses the results observed both with regards to functionality and performance, as well as considers avenues for expansion and refinement. This work also situates this algorithm within the larger system described by this research effort, both for context and as an example of its potential benefit.

***Index Terms*—Earliest Deadline First Scheduling, Scheduling Algorithm, System-on-a-Chip, Field Programmable Gate Array, Embedded System, Heterogenous System**

I. Introduction

FIELD Programmable Gate Arrays, or FPGAs, have long been utilized in devices benefiting from hardware acceleration of processes unsuitable for execution on a traditional processor, but which don't warrant the time consuming and cost-prohibitive process of ASIC design. Accordingly, it should come as no surprise that, as much of the world pivots from on-site datacenters and computing resources to hybrid or cloud based platforms, Cloud Service Providers (CSPs) have shown an interest in providing FPGA-as-a-service resources to their tenants. Amazon, Google, and Microsoft all offer such services as part of their respective cloud platforms, as do many of the other major players in the industry.

Currently, these offerings mirror the traditional FPGA implementation - a single physical device per tenant instance. This is due to a variety of factors, mostly centered around device security for both the tenant and the CSP. However, research is underway that seeks to utilize Dynamic Partial Reconfiguration (DPR) as a means to enable CSPs to partition large FPGAs into multiple virtual devices, each of which can be allocated to a tenant. This would allow CSPs to utilize much more economical devices, driving down cost for tenants and likely spurring increased adoption. Before this technology can see widespread adoption, it must address a number of security concerns surrounding co-tenancy, as well as trust relationships between the CSP and the tenant.

This effort seeks to address the latter concern, that of CSP - tenant trust. Understandably, many tenants are hesitant to provide their Intellectual Property (IP) directly to the CSP, as there is financial incentive for a CSP to integrate some of that IP into their own accelerator offerings. Accordingly, tenants seek to upload encrypted IP to their partitions, with assurances that it is not possible for the CSP to directly access the decrypted content. This is feasible, and has already been researched in academia. However, the solution posed in the literature is inefficient in its duplication of components, and furthermore contains a flaw that entirely violates the integrity of the bitstream as it pertains to its inaccessibility in its decrypted form by the CSP. This research effort seeks to address that vulnerability, while also providing performance enhancements to the original design. Specifically, this work, as a part of the larger effort, seeks to provide an efficient way of allocating shared decryption resources in a manner that reduces overhead while still ensuring the integrity of the IP's encryption through all CSP-accessible parts of the tenancy cycle.

A. Organization of this Work

This work is organized into several sections. Section II outlines the overarching proposal for the larger research effort of which this work is a part. This discussion of the larger effort is provided as context for the specific implementation described in later sections. Section III outlines the design of the scheduling algorithm for allocating the AES Decryption cores between the multiple tenant partitions on the FPGA. Section IV describes the implementation of this algorithm, while Section V describes the findings that have resulted from this implementation process. Section VI concludes the report.

II. Proposal

A. Literature Implementation

In "Cryptographically Secure Multi-Tenant Provisioning of FPGAs"[1] Bag et al. outline a comprehensive architecture for the implementation of a cloud-compatible FPGA architecture whereby multiple discrete tenants can occupy "Virtual FPGA" (VFPGA) partitions of a larger Physical FPGA (PFPGA) device in a secured environment. In their proposal, they utilize Key Aggregation Cryptography (KAC) to facilitate the encryption of a symmetric (AES) key, which in turn can

S. Donchez is with Villanova University, Villanova, PA 19085 USA e-mail:sdonchez@villanova.edu

Manuscript received December 19, 2021

be used to decrypt tenant bitstreams within their allocated partition. In this implementation, they utilize a single KAC decryption engine at the PFPGA level, which extracts the AES key and furnishes it to individual AES engines at the VFPGA level. The authors claim that by implementing their design in this manner, the only party which the tenant is required to trust is the FPGA Vendor, who provides the public and private keys used by the KAC engine to decrypt the symmetric key.

B. Implementation Evaluation

However, this claim is not completely true. By virtue of decrypting the symmetric key at the PFPGA layer, the Cloud Service Provider (CSP) is in possession of both the encrypted bitstream and the symmetric key required to decrypt it. In essence, this is equivalent to them having access to the IP contained within the bitstream itself. Therefore, the tenant must necessarily also trust the CSP, which is not a policy compatible with the security needs of many would-be tenants.

Additionally, this implementation allocates a considerable amount of resources to the multiple identical AES decryption cores, which don't actually yield any additional security benefit since the AES key is present at the PFPGA layer alongside the bitstream it encrypts. Furthermore, this implementation requires a predefined number of VFPGAs per PFPGA, which constrains the CSP's ability to adjust the size and number (inversely) of VFPGAs per PFPGA to adapt to demand.

C. Proposed implementation

The author proposes modifying this implementation to eliminate the need for trust in the CSP, along with eliminating the duplication of the AES IP. To achieve this goal, the author proposes the implementation of an attestably secured partition within the PFPGA fabric, which contains both the KAC and AES engines. By isolating both of these components in an attestably secure partition, the CSP no longer has access to the decrypted symmetric key, preventing them from decrypting the bitstream and gaining access to the IP contained therein. Meanwhile, the presence of the single AES IP in this secured partition eliminates the need for identical engines on each partition, which increases available space and also offers flexibility with regards to VFPGA provisioning, as the only fixed limit remaining is the number of VFPGA device IDs allocated by the FPGA Vendor for the device. This number can simply be made arbitrarily large relative to the realistic number of simultaneous tenants that the device can support, such that the CSP is effectively free to divide the PFPGA as they see fit.

The presence of a secure partition for decryption operations is useless without the ability to securely transport the decrypted data from said partition to the VFPGA in question. The use of the Hard Processor System (HPS), and specifically a Trusted Execution Environment (TEE) implemented therein, could conceivably facilitate the secure transfer of this information between regions of the FPGA without transiting the unsecured outer region of the programable logic. However, implementation of direct connections between partitions and the HPS such as this necessitates will require further investigation, as the mechanics of such an operation are not currently understood by the author. Alternatively, the use of direct memory access could be a route to secure transfer, although assurance must exist to prevent malicious access to the memory regions in question.

The use of the HPS to facilitate such a transfer has additional advantages, as it is conceivable on larger devices that multiple such partitions could be desired to facilitate simultaneous tenant reprogramming. In either case, the use of the HPS to schedule access to the secured partition(s) could similarly prove to be a useful design component.

D. Research Objectives

The above implementation requires a number of discrete objectives be completed and integrated into a comprehensive design. Those objectives are described below.

- Obtain or design the AES and KAC cores for the secured partition.
  - Ideally these cores should be pipelined for efficiency
- Investigate direct communication between partition and HPS (without untrusted transit)
- Investigate use of DMA to facilitate secure communication between partitions securely
- Design and Implement scheduling algorithm to allocate decryption partition to tenants
  - With multiple AES cores, it may make sense to share a single KAC engine if secured link between them can be guaranteed (the KAC has to decrypt a single key of trivial size, whereas the AES engine is in much higher demand due to need to decrypt entire bitstream)
- Implement attestably secure partition and provide mechanism for attestation and verification
- Implement Trusted Execution Environment (if using HPS for secured data transfer)
- Implement data routing and verification logic within TEE (if using HPS for secured data transfer)

E. Focus and organization of this work

This work represents efforts addressing a specific subset of the larger research focus outlined above. Specifically, it pertains to the development and implementation of a scheduling algorithm for the utilization of a set of shared AES Decryption Cores between a larger number of VFPGAs contained on a single PFPGA. The following sections outline in detail these efforts.

III. AES Engine Scheduling

Since the AES engine is decrypting a bitstream, as opposed to a simple key (as is the case in the KAC engine), it will necessarily be occupied for a considerably longer period of time than the KAC engine for each reconfiguration cycle. Furthermore, since the trend in cloud infrastructure is to implement large devices with many partitions rather than many smaller ones, it is likely that a single AES engine per PFPGA

will be insufficient to accommodate the partial reconfiguration needs of said device's tenants.

To this end, this architecture proposes the implementation of multiple discrete AES cores, each of which may be attached to a partition for the purposes of facilitating a decryption operation as part of the reconfiguration process. This necessitates the scheduling and allocation of these resources in a fashion that takes into account the real-world constraints of the CSP. To this end, this architecture proposes an Earliest Deadline First (EDF) scheduling algorithm, to be implemented on the Hard Processor System (HPS) of the heterogenous system. This algorithm, including its assumptions, constraints, and design, is outlined in the following sections.

A. Assumptions

The following assumptions are made regarding the implementation of the AES core scheduling process for this architecture:

- The time required to decrypt a given partial bitstream increases linearly with a corresponding increase in file size.
- The time required to decrypt a given partial bitstream can be approximated into a discrete number of "time units".
- Deadlines for decryption are made available as tasks are received, and are driven by real world constraints (License Agreements, Pricing Tier, etc.).
- The CSP will not allocate more decryption jobs to the PFPGA than can be scheduled within their given deadlines (meaning that the schedulability does not need to be verified).
- The time required to switch from processing one bitstream to processing another is both non-trivial and constant.
- The act of reading (ingesting) and writing (outputting) bitstream segments to/from the AES engine requires a known, constant time.
- The time required for communication between the HPS and the AES core is assumed to be both constant and trivial.
- Due to the nature of the environment in which the algorithm is being implemented, it is important to note that tasks are random and likely aperiodic in nature.

B. Constraints

The implementation outlined in subsequent sections will be subject to the following set of design constraints, imposed by the assumptions listed in the preceding section as well as best practices:

- In the event of a tie between two task for the next time unit, where one task is the task that was operated on in the previous time unit, the task which was previously being executed shall continue to be executed, in order to eliminate overhead associated with context switching between tasks.
- As the IP core currently selected for performing the AES decryption operation is non-pipelined, and therefore not capable of transitioning seamlessly between decryption operations without first having its state machine reset, there shall be a small reset delay during context switches
- The algorithm shall be implemented for an abstract number of AES cores N, with the assumption that N shall be some small number in the real-world implementation.
- The HPS shall communicate with the AES cores via the AXI bus.
- The bitstream shall reside in the domain of the PS for purposes of this algorithm, and shall be transferred via the AXI bus.
- The atomic unit of the bitstream shall be considered 1 Kilobyte, as too small of an atomic unit will cause scheduling computations to take longer than the time unit itself.
- Tasks are placed in a queue, Q, arranged by deadline in ascending order (such that the earliest deadline is at the start of the queue).
- The deadline associated with a processor is the deadline of the task it is currently executing, or ∞ if it is idle.

C. Algorithm Design

Algorithm 1 implements the scheduling routine for the AES Cores, as governed by the constraints and assumptions outlined above. This algorithm serves as the guidance for the implementation of the scheduler as outlined in subsequent sections, subject to the constraints and assumptions outlined above.

IV. EDF Algorithm Implementation

This effort implemented the above algorithm in a test application which is representative of its implementation in the real-world use case outlined in Section II-C. However, as the scope of the research completed to date does not include the larger system outlined in that section, it is somewhat constrained with regards to its external interfaces. Namely, it does not attempt to directly affect any actual IP Cores, but rather performs theoretical scheduling of the cores and notes its results in a log file. It does, however, attempt to mimic the likely real-world method of input, in that it parses serialized data such as would be suitable for transmission by a discrete centralized controller overseeing a number of PFPGA instances.

Along with the development of the Scheduler itself, it was also necessary to thoroughly exercise the application to ensure that it functions successfully under load. As a result, it was also necessary to develop a testbench application, which is capable of generating arbitrary tasks for the AES cores to schedule. This testbench is capable of generating tasks in accordance with a given target utilization rate. A comprehensive set of interprocess communication (IPC) calls enable synchronization between the two applications. For purposes of ensuring low resource utilization, as well as promoting well organized source code, both of these applications were developed in C++, utilizing an object-oriented approach. Section IV-A, below, outlines the implementation of the scheduler itself, while Section IV-B outlines the implementation of the

Algorithm 1 Global EDF Scheduling Algorithm for AES Decryption Cores

We consider a task set τ consisting of x tasks T_1 , . . . , T_n that are sporadically released from the CSP with arbitrary deadlines
Variables:
N - the set of cores to be utilized
Q - the queue containing all tasks currently pending execution
T_{n_i} - the task assigned to a given core
D_t - the delay associated with a given task
D_{n_i} - the delay associated with the task currently executing on a core (for brevity)
D_{Q_i} - the delay associated with the task at a given position in the queue (for brevity)

Require: $n(N) > 0$ ▷ There are a nonzero number of cores
  function SWAPTASKTOQUEUE(T)
    for all $Q_i \in Q$ **do**
      if $D_T < D_{Q_I}$ **then**
        Insert T into Q at index Q_i
        return pop(Q)
      end if
    end for
    Append T to the end of Q ▷ If all currently queued tasks expire before T
  end function

  while true **do** ▷ Each Time Unit
    for all $n_i \in N$ **do**
      if $Q \neq \emptyset$ **then**
        if $D_{n_i} == \infty$ **then**
          $T_{n_i} \leftarrow T_{Q_0}$ ▷ Pop Q
        ▷ If the currently executing task has a later deadline than the first task in the queue and has the latest deadline of any currently executing task across all cores.
        else if $(D_{n_i} > D_{Q_0})$ AND $D_{n_i} == max(D_N)$ **then**
          $T_{n_i} \leftarrow$SwapTaskToQueue(T_{n_i})
        else
          $T_{n_i} \leftarrow T_{n_i}$
        end if
      end if
    end for
  end while

testbench. Section IV-C outlines the environment within which the applications run, while IV-D outlines the interconnection of these two applications.

A. Scheduler Implementation

The scheduler application consists of several interrelated threads operating in parallel. First among these is an input parser, which waits for serialized task information and, upon receipt, deserializes it into task objects. For the sake of best practices (as well as ease of parsing), this data is encapsulated in a JavaScript Object Notation (JSON) string, which, despite its name, is a language agnostic structure that is commonly used to represent entities. In its current implementation this data is simply read from a file descriptor (such as standard input), although future extensibility to a socket based architecture for network communications is possible.

Alongside the input parser is the timer manager. The timer manager is responsible for managing the discrete units of time for which the cores can be scheduled. The AES cores intended for use in the larger research effort are currently non-pipelined cores requiring a fixed number of clock cycles to process each word of data. Accordingly, the frequency with which they can be scheduled is based on a multiple of that number of clock cycles, which must be small enough to afford flexibility but high enough to minimize wasted cost due to context switching (per the design assumptions). As the HPS and the PL do not necessarily share a common clock, and as the execution of the various threads on the processor is non-deterministic in its own scheduling, it is necessary to utilize a timer to ensure scheduling operations are executed exactly once per atomic unit of IP Core schedulability. This thread provides that functionality, and sets flags for the other threads as needed to facilitate their operation.

The final (and arguably most consequential) thread of the scheduler application is the core servicer itself, which is responsible for ensuring each core is executing the ideal task per the prescribed algorithm. Once per Time Unit, the core servicer evaluates the deadlines of the tasks currently executing on the cores against those contained in the scheduler's Task Queue, and performs substitutions as necessary to ensure that the tasks with the nearest deadlines are those currently in execution.

For purposes of this effort, these services are implemented using C++'s standard threading library (as opposed to the more traditional "pthread" library). This threading implementation was chosen to promote cross-platform interoperability, as well as performance optimization where possible.

B. Testbench Implementation

The Testbench application in in many ways appreciably simpler than the Scheduler itself. It is only a single threaded application, and the task it performs is straightforward - it simply creates tasks populated with arbitrary information. However, this seemingly straightforward operation is greatly complicated by one of the assumptions outlined in Section III-A above. Namely it is the fourth assumption in that section which proves problematic: The CSP will not allocate more

decryption jobs to the PFPGA than can be scheduled within their given deadlines. This greatly eased the development of the Scheduler itself, by eliminating the need for it to perform schedulability checks as part of its task processing. However, as the Testbench is effectively serving as the CSP in these simulations, it falls to the testbench to ensure that the tasks it generates do not overload the PFPGA instance for which the Scheduler is allocating resources.

This is accomplished through introducing some element of reporting to the Scheduler itself, and making those results available to the testbench. Specifically, the scheduler maintains a table outlining how many outstanding time units need to be executed by any given point in time, as described by the sum of all of the outstanding units for each task due at that point. By combining this information with knowledge of what unit in time the PFPGA and PS are currently in, it is possible to validate in real time the schedulability of a task as it is being generated, and to make adjustments as needed.

This logic is captured in Equation 1, below. In this equation, t_c represents the current Time Unit, t_n represents the time unit being proposed as the deadline for the task, D_i represents the number of units due at time i, and UtE represents the number of Units to Execute for the task currently being evaluated. P is the parallelization factor, or the number of cores which the schedule is being distributed between, while $n - c$ yields the total number of TimeUnits between the time the equation is evaluated and when the deadline will come. The "Desired Load Percentage" is the target utilization rate for the cores.

$$\frac{(\sum_{i=t_c}^{t_n} D_i) + UtE}{P * (n - c)} < Desired\ Load\ Percentage \quad (1)$$

By verifying the validity of this equation for each generated task's proposed deadline, it is possible to ensure that target average utilization is not exceeded. Furthermore, the maintenance of this information on the part of the scheduler is non-resource intensive, requiring a single addition operation as each task is parsed, a single decrement operation as each core is serviced, and a single increment operation (for the current unit counter) on the part of timer manager per Time Unit.

Unfortunately, this does not fully resolve the issue outlined above. In addition to ensuring that the average utilization is not compromised, it is also important to verify that the instantaneous utilization never exceeds the maximum. (For example, inserting a task that has a Units to Execute value of 100 with a deadline of 500 may not compromise the average utilization, but may cause a task due at unit 510 to lack sufficient time to run). To rectify this, an additioanl check is necessary.

To perform this additional check, the generator iterates across the entire simulation window, computing a running sum of the outstanding work to be done that is due at that unit, and comparing that value plus a prorated portion of the new task's units against the theoretical limit. This check is captured by Equation 2:

$$\sum_{i=t_0}^{t_n} D_i + \max(\min(UtE, i - t_c), 0) < P * (n - i - c) \forall i > c \quad (2)$$

Provided the conditions of both Equations 1 and 2, above, are satisfied, the testbench application can then assume that the proposed deadline is valid for the generated task.

C. Operating System and Environment

By virtue of introducing multiple threads to the scheduler (and also by virtue of running two applications on a single core processor), it is necessary to implement an operating system (OS) on the processor system upon which the scheduler simulation is running. Futhermore, the larger design will add additional functionality to the processor system, further necessitating an operating system for thread management and for scheduling of the processor itself. To this end, several operating system choices were considered for implementation on the hardware target in support of this simulation. As the target in question includes a Xilinx SoC, it was decided that the ideal choice for such a system would be the Xilinx PetaLinux OS, as it includes all of the necessary drivers for interfacing with the various devices on the development platform. Additionally, the PetaLinux platform is based on the OpenEmbedded Project's toolchain, which affords tremendous flexibility and customization potential as needed.

For ease of development and testing, the two software applications were designed to run both in the native Windows environment of the development computer as well as the Linux environment of the target. This reduced cycle time between incremental iterations of the applications, as well as eliminated dependence on an extensive physical setup for testing.

D. InterProcess Communications

The inclusion of an operating system into this research effort greatly eased efforts to facilitate communication between the two discrete applications. Both Windows and Linux implement comprehensive facilities for InterProcess Communication (IPC), which enable the transfer of data between applications. Unfortunately, the two systems do not have a common set of IPC functions, requiring slightly difference implementations to satisfy the two architectures. Despite this, the overall design of the IPC structures for the two architectures is effectively the same.

1) Testbench to Scheduler Communication: Communication from the Testbench application to the Scheduler application is straightforward, as the only traffic flowing in this direction is the serialized task data. Since this data is serialized and relatively small in size, and in the interest of replicating the real-world functionality of the software, this data is simply read in from a file descriptor. In Windows, this is accomplished via a Named Pipe, which effectively enables the data to be consumed as if it were typed into the standard input of the application in the terminal window in which it is executing. Specifically, this named pipe was configured in "message" mode, such that the entire task is treated as

an atomic unit for purposes of the transfer, rather than its constituent bytes (the latter configuration can result in multiple or even partial messages being sent as a single input, which would unnecessarily complicate the parsing operation).

Linux also provides a "Named Pipe" functionality, but this functionality is strictly byte-oriented as opposed to message-based. Accordingly, it is not desirable for use in this scenario. Fortunately, the Linux system also supports System-V Message Queues, a part of the Posix standard that is functionally equivalent to Windows's Named Pipes in message mode. The only impactful difference between the Windows and Linux implementations of this facility is that the Linux implementation does not facilitate the ability to wait for a connection, thus requiring the use of a Semaphore (another POSIX IPC method) to ensure synchronization between the two applications.

2) Scheduler to Testbench Communication: The transfer of data from the Scheduler to the Testbench is appreciably more complex, as the table of Outstanding Units Due (used in Equation 1) is updated during each TimeUnit, and contains $P * N$ words of data, where N is the number of time units to simulate and P is the number of IP Cores. It would be inefficient to the point of impracticability to transfer and parse that data every time unit, as its scale for even modest simulations would likely consume a non-trivial portion of the available compute time to parse. Accordingly, the implementation instead uses a shared memory space for the purposes of facilitating this communication, accomplished in Windows by utilizing the File Mapping functionality of the Win32 API, and in Linux via the comparable Shared Memory Segment functionality. In this way, the generator simply receives a pointer to the scheduler's own copy of the table, and is always able to view the latest updates to it without any need for direct transfer between the applications.

The same functionality is utilized for the mapping of the current Time Unit between the two systems, as it eliminates any edge cases wherein the unit changes shortly after the testbench queries it. This allows for the most accurate possible scheduling of tasks.

V. Findings

The implementation of the EDF algorithm and testbench on the Windows target has revealed that it is able to successfully schedule tasks at high utilization levels with low miss rates, given that the requirements outlined in the design constraints and assumptions are met. Careful analysis of the scheduler's output reveals that it is scheduling in accordance with the algorithm, with only a single exception - at the end of the simulation, as the task queue depletes, the scheduler unnecessarily shifts tasks from higher index cores to vacant lower-indexed cores. Ultimately, this inconsistency is non-consequential, as in a real-world implementation it is unlikely that the task queue would ever be depleted (and also as it does not cause any tasks to miss their deadline).

A. Scheduler Effectiveness

As the primary measure of success for this scheduler implementation is how it performs under heavy load, a series of trials were conducted on the hardware target under varying levels of load in order to determine the average performance of the algorithm for each loading. These results are described in Table I, below. All iterations were performed for a simulation of 1000 TimeUnits. The results below are computed from the average of three trials for each utilization, each utilizing a different randomly generated set of tasks.

Target Utilization	Num. Tasks	Units Elapsed	% Units Missed
50%	38	512	0%
60%	47	604	0%
70%	53	732	0%
80%	64	825	0%
90%	74	925	0.15%
100%	82	1000	8.63%

TABLE I: Scheduler Effectiveness

B. Performance Evaluation

Although the effectiveness of the algorithm (and its implementation) are by far the most important criteria for evaluating their success, it is also important to consider the performance of the application. In the context of the larger system outlined in Section II, a number of other processes will likely have to execute on the single core available on the hardware target. Accordingly, it is crucial that the scheduler is not overtaxing the HPS, such that the other portions of the system are also able to execute as needed. Accordingly, Table II, below, reports on the total utilization time of the processor for each of the iterations above.

Utilization	Clock Time	User Time	Sys Time	RAM Used
50%	10.79s	10.34s	0.07s	10.48MB
60%	11.04s	10.35s	0.06s	10.63MB
70%	10.85s	10.35s	0.07s	10.61MB
80%	10.93s	10.33s	0.11s	10.54MB
90%	10.87s	10.39s	0.05s	10.73MB
100%	10.92s	10.39s	0.09s	10.63MB

TABLE II: Scheduler Performance

C. Analysis of Findings

These findings indicate a high level of success on the part of the scheduling algorithm, as it effectively schedules tasks even at high levels of utilization. It is worth noting that the case of 100% utilization is a special case, as a detailed analysis of individual runs reveals some discrepencies that explain the high miss rate. The current implementation of the testbench does not immediately send a task at TimeUnit = 0 on every iteration, as its operation is, by design, random. As a result, the logged allocation of cores to tasks often indicates empty cores for the first several TimeUnits. However, the testbench is still attempting to generate 100% of the TimeUnits (4000 in the case of these trials), meaning that it will occasionally overschedule the processor. It has been observed through additional experimentation that the increase in miss rate between 90% and 100% utilization more nearly approximates an exponenetial line of best fit than a linear one, further validating the performance of the algorithm.

It should also be noted that much of the ”User Time” listed in Table II above is currently ”idle” time. As the current implementation of the scheduler is not controlling real IP, most of the time in each TimeUnit is spent waiting for the unit to expire. This waiting process is not optimized to minimize processor consumption in the implementation used to generate these results, as it is a placeholder for future work.

Additionally, resource consumption is not closely correlated with target utilization, but appears roughly constant across all target utilization levels. This is highly beneficial in this scenario, as, with the larger and more complex Task data that will necessarily accompany real tasks being implemented on real IP, any utilization correlated growth in resource consumption would be greatly magnified (and as it is desireable to maximize utilization for revenue purposes on the part of the service provider).

VI. Conclusion

In this paper, a novel solution for scheduling the allocation of shared IP cores on a multi-tenant FPGA is proposed and designed utilizing an Earliest Deadline First (EDF) based scheduling algorithm executing on a general purpose processor co-located with the programmable logic as part of a heterogenous embedded system. This paper also describes the implementation of this design, along with a stimulus application designed to execute it, both authored in C++ and utilizing both Windows and Linux InterProcess Communication functionality for flexibility of evaluation. The paper then discusses the results of the implementation, both with regards to primary objectives (the effectiveness of the algorithm itself), and secondary objectives (performance, etc.)

A. Avenues for Future Work

This scheduler implementation outlines a solution for the scheduling of common IPs between multiple VFPGAs on a heterogenous system, but it is conceivable that a CSP may desire to implement VFPGAs on a purely PL device, such that no onboard processor exists. Although such an implementation would not facilitate the implementation of much of the larger system outlined in Section II above, it does pose a number of avenues for future work specific to the implementation of a scheduling algorithm itself. Similarly, the current implementation’s testbench is understandably primitive in nature compared to the larger, more complex stimulus applicationset that a CSP would utilize. Finally, it is worth noting that a logical route for expansion is the implementation of the larger system outlined in Section II above, which would necessarily depend on the integration of the components developed herein.

Acknowledgment

The author would like to thank Dr. Xiaofang Wang, Villanova University, for her support in the authoring of this paper, as well as Dr. Sarvesh Kulkarni, Villanova University, for his support in application development efforts. The author would also like to thank Dr. Hubertus Franke, IBM T.J. Watson Research Center, coauthor of [2] for his guidance and willingness to provide assistance when contacted in the course of this effort.

References

[1] A. Bag, S. Patranabis, D. B. Roy, and D. Mukhopadhyay, “Cryptographically Secure Multi-tenant Provisioning of FPGAs,” in *Security, Privacy, and Applied Cryptography Engineering*, ser. Lecture Notes in Computer Science, L. Batina, S. Picek, and M. Mondal, Eds. Cham: Springer International Publishing, 2020, pp. 208–225.

[2] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, “Enabling FPGAs in the cloud,” in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF ’14. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2597917.2597929

Stephen Donchez (M’20) received his B.S. in computer engineering from Villanova University, Villanova, PA in 2020. He has remained at Villanova in pursuit of his M.S. in computer engineering, which he anticipates receiving in May of 2022.

In the summer of 2018 he was a software engineering intern at Harris Corporation (now L3Harris Technologies, Inc.). He returned to L3Harris for the summer of 2019 as a systems engineering intern, and returned again to the same position for the summer of 2020 at their facility in Clifton, NJ, where he continues to work on a part-time basis. He anticipates joining L3Harris’s Camden facility as a Systems Engineer specializing in Cybersecurity following his Master’s Program.

His research interests include FPGAs, embedded software development, and embedded systems with a focus on System-on-a-Chip technologies and security.

Mr. Donchez is a student member of the IEEE.