UNIVERSITÉ LIBRE DE BRUXELLES

Evaluating Impact of Reconfiguration Speed on FPGA Performance

Author:
Chris LAMPLE

Supervisors:
Prof. Joël GOOSSENS
Prof. Dragomir MILOJEVIC

Additional jury member:
Prof. Bernard FORTZ

August 14, 2021

ACKNOWLEDGEMENTS

The completion of this thesis was a challenging process for me, and it was only with a large amount of support from others that I was able to complete this work.

Thank you Professor Goossens and Professor Milojevic for introducing me to such an exciting idea for a thesis and for sharing your expertise. It was a real pleasure to be able to work with both of you.

Thank you also to my family - I really appreciate your love and support over all of these years.

Thank you Mario for spending your day off work reading my thesis and giving me comments.

A very special thank you is needed for Marie. For every hour I spent working on this thesis, I must have spent two hours procrastinating or worrying about it. I appreciate you always being willing to talk about it. Thank you for your boundless patience and support in helping me get through this.

LIST OF FIGURES

2.1 FPGA logic cell 7
2.2 Loop unrolling optimization 10
2.3 FPGA resources vs. performance for hash functions 10
2.4 Improvements in FPGA reconfiguration speed 14
2.5 Wire lengths in 3D packaging 15
2.6 Foveros 3D packaging 15
2.7 Optimizing execution time with prefetching 17

3.1 Gap in makespan between simplified and original model 31

4.1 Example schedule with slow reconfiguration 34
4.2 Example schedule with medium reconfiguration speed 35
4.3 Example schedule with fast reconfiguration speed 36
4.4 Example schedule, makespan vs. reconfiguration speed 37
4.5 Average makespan with respect to reconfiguration time per resource unit . 40
4.6 Average number of SW implementations used vs. reconfiguration speed . . 41
4.7 Average FPGA implementation size vs. reconfiguration speed 42
4.8 Average number of FPGA partitions vs. reconfiguration speed 43
4.9 Effect of improving FPGA implementation execution time 44
4.10 Effect of reducing FPGA implementation performance 45
4.11 Average effect of splitting tasks on schedule makespan 48

LIST OF TABLES

3.1 Model parameters 23

4.1 Scheduler runtime 38
4.2 MAPE values for the iterative scheduler and simplified MILP formulation . 39
4.3 Example of task splitting 46

CONTENTS

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Research Questions . . . 3
- 1.3 Research Method . . . 3
- 1.4 Structure of Thesis . . . 4

2 Background **6**
- 2.1 FPGA's . . . 6
  - 2.1.1 FPGA's In Comparison With Software . . . 7
  - 2.1.2 FPGA Resource vs. Performance Trade-Offs . . . 9
  - 2.1.3 Dynamic Partial Reconfiguration . . . 11
- 2.2 Improving Reconfiguration Speed . . . 13
- 2.3 3D Packaging . . . 14
- 2.4 DPR Schedulers . . . 16
  - 2.4.1 Deiana et. al Scheduler . . . 16
  - 2.4.2 FRED Real Time Framework . . . 18

3 Proposed Solution **20**
- 3.1 Definition of the Model . . . 20
  - 3.1.1 Relevant Aspects To Model . . . 20
  - 3.1.2 Model Definition . . . 22
- 3.2 Applying the Deiana et. al Scheduler . . . 24
- 3.3 Determining Schedule Lower Bounds . . . 26
  - 3.3.1 Simplifying The Scheduling Problem . . . 27
  - 3.3.2 MILP Formulation . . . 27
  - 3.3.3 Verifying That A Lower Bound Is Provided . . . 29
  - 3.3.4 Remarks on the Simplified MILP Formulation . . . 30
- 3.4 Generation of Workloads and Model Parameters . . . 31

4 Results **33**
4.1 Implementation and Setup 33
4.2 Example Schedules 33
4.3 Scheduler Evaluation 37
4.4 Overall Impact of Reconfiguration Speed 39
4.5 Effects of Adjusting Resource vs. Performance Trade-Off 43
4.6 Effects of Restructuring Applications 45

5 Towards a Lagrangian Based Scheduler **49**
5.1 Introduction 49
5.2 Connection with $R||C_{max}$ 51
5.3 Lagrangian Relaxation of the Simplified Scheduling Problem 52
5.4 Simplification for Solving the Lagrangian Relaxation 58
5.5 Results 60

6 Conclusion **62**
6.1 Contributions 62
6.2 Limitations 64
6.3 Future Work 64

1 INTRODUCTION

1.1 Background

For decades, Moore's Law has had impressive predictive power and integrated circuits have continued to improve at an exponential rate [17]. This has delivered a reliable cadence of faster and cheaper computers. A key technological driver in this exponential improvement has been shrinking the size of transistors. The benefits of this approach, however, are beginning to reach physical limitations [22, 17].

While the era of more performant computers through smaller transistors may be coming to an end, new applications of computing still demand more powerful devices. In their article "The End of Moore's Law: A New Beginning for Information Technology", Theis and Wong claim that "the gradual end of Moore's law will open a new era in information technology as the focus of research and development shifts from miniaturization of long-established technologies to the coordinated introduction of new devices, new integration technologies, and new architectures for computing" [22]. They also go on to identify technologies which they think can satisfy this demand: of which, *field-programmable gate arrays* (FPGA) and *3D microelectronics packaging* are particularly relevant for this thesis. With respect to FPGA's, Theis and Wong state that they can "easily envision future energy-efficient systems consisting of a great many accelerators executing specific operations or algorithms, their interactions orchestrated to perform larger tasks, and turned on and off as needed". With respect to 3D packaging, they also state that "the realization of monolithically integrated multi-layer

logic and memory would be a revolution, and that revolution could already be brewing".

As mentioned by Theis and Wong, FPGA's could be increasingly important for improving computing power as Moore's Law comes to an end. In contrast with typical software, which expresses programs in terms of CPU instructions, FPGA's allow programs to be expressed as specialized logic circuits. By expressing programs in this way, applications targeting FPGA's can be made much more performant than software implementations. FPGA's also offer the benefit of being re-programmable. Unlike with application specific integrated circuits (ASIC's), which need to be manufactured for a specific program, with FPGA's it's possible to simply use commercial off-the-shelf hardware.

Due to their performance advantages over software and their flexibility in comparison to ASIC's, FPGA's have already received significant attention. To name just a few applications, case studies have used FPGA's to accelerate image processing, encryption, as well as machine learning algorithms [5, 13, 27]. In the case of a particular feature detection algorithm, an FPGA implementation offers a 12.89 times speedup compared to a CPU implementation and a 1.47 times speedup compared to using the GPU [13].

This thesis focuses on one particular feature of FPGA's: *dynamic partial reconfiguration* (DPR). This feature, which is already widely available in modern FPGA's, allows for a portion of an FPGA to be reconfigured while the rest of the FPGA continues computation. In general, applications implemented on an FPGA face a trade-off between the available resources and the application performance and functionality. By using DPR, applications can achieve more performance or offer more functionality using only the same amount of FPGA resources.

Currently the time needed for reconfiguration with DPR can be on the order of milliseconds, making it a relatively expensive operation and limiting the usefulness of this feature. One technology which could potentially improve the reconfiguration speed, though, is 3D packaging. By packaging the FPGA logic along with the memory used for storing configurations, the reconfiguration process could conceivably be made significantly faster. In turn, this improved reconfiguration speed could allow applications to make more extensive use of DPR and become much more efficient as a result.

1.2 Research Questions

Developing such a novel FPGA using 3D packaging would be a large undertaking. Before pursuing this technology further, it's interesting have a tentative answer to just how useful such an improvement in the reconfiguration speed would be. Clearly improving the reconfiguration speed will offer at least *some* improvement for some applications. In order to warrant the large amount of work that would be required in hardware development, though, it should be clear that a large improvement is available as a result.

An important challenge in answering this question is that different applications will likely benefit to different extents from improved reconfiguration speed. Applications with very limited functionality or where FPGA's offer only a modest performance improvement over software will possibly benefit less, for example. It may also be necessary to adapt how applications are implemented to fully take advantage of the new reconfiguration capabilities. For instance, when the reconfiguration speed is slow it may be best to limit the use of DPR: dividing the application into fewer components and reconfiguring between them infrequently. When the reconfiguration speed is faster, however, a more efficient implementation might make more use of DPR: dividing the application into many components and reconfiguring between them frequently.

With this in mind, it's possible to state the following research questions:

- To what extent can improving reconfiguration speed improve the performance of applications?
- How should applications be implemented to best take advantage of improved reconfiguration speed?

1.3 Research Method

Much work has already investigated DPR, but, to the best of my knowledge, existing literature doesn't explicitly study the impact of dramatically improving reconfiguration speed. To begin investigating these research questions, there are two approaches which seem possible.

The first possible approach is to implement real-world applications and run them on an

emulated version of the new FPGA architecture. While this approach would give a realistic evaluation of the new FPGA architecture, it would likely be very challenging to implement. In addition, it's not yet clear which applications stand to benefit the most from the new architecture and how they should be implemented to best take advantage of the improved reconfiguration speed.

A second possible approach is to model applications as abstract sets of tasks and model the FPGA only at a high level. Using this high level model, it would then be possible to evaluate the performance of different abstract workloads on the FPGA with different reconfiguration speeds. This approach of modeling applications of abstract sets of tasks has already been applied to real-time FPGA scheduling [2].

This thesis investigates the research questions using the second approach. A limitation is that the results are only as reliable insofar as the high level FPGA model is consistent with the actual hardware and actual applications. Since this is an early step in investigating the subject, the synthetic workloads and parameters of the FPGA model haven't been made to exactly match real world hardware and applications. As a result, this work falls short of delivering a precise answer to the research questions such as "a 50% speedup is possible for real world applications". Still, the experiments with synthetic workloads reveal some tendencies that will hopefully be a useful starting point for future work.

1.4 Structure of Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 introduces background information on FPGA's and DPR which is broadly relevant to the research questions.
- Chapter 3 defines a model of the FPGA and applications which will be used to investigate the research questions.
- Chapter 4 uses the approach proposed in the previous chapter to investigate the impact of improving reconfiguration speeds. Some particular effects of improving reconfiguration speeds are identified.
- Chapter 5 attempts to apply the Lagrangian relaxation technique to the FPGA schedul-

ing problem. The proposed approach proves to be unsuccessful, but the possibility of applying the Lagrangian relaxation technique may still have some potential.

- Chapter 6 summarizes the results of this thesis and outlines potential approaches for future research.

2 BACKGROUND

2.1 FPGA's

To understand the appeal of FPGA's, it helps to consider them in contrast with two other platforms: CPU's and ASIC's. In software, an application is expressed in terms of the CPU instruction set and these instructions are then executed by the CPU. This has the advantage of flexibility: the same CPU can be used to run any application, provided that it's compiled for the right instruction set. In contrast, FPGA's and ASIC's express applications as logic circuits.

In the case of ASIC's, a custom chip is manufactured which encodes the logic circuit. The need for manufacturing a custom chip has many downsides: it's difficult to adjust the application after manufacturing, it's only economically feasible at large production volumes, and the design process is more complex than implementing an application as software. At the same time, this allows for ASIC's to be highly efficient [14].

Rather than encoding the logic circuit of an application directly in the physical chip as ASIC's do, FPGA's load the logic circuit specification into a memory. This logic circuit specification is commonly referred to as a *bitstream*. Once a bitstream is loaded, the FPGA emulates the specified logic circuit. Since the logic circuit is simply loaded into memory, rather than physically manufactured into the chip, it's possible for applications targeting FPGA's to use commercial off the shelf hardware. This sidesteps many of the complications involved in producing ASIC's.

More specifically, FPGA's consist of thousands of *logic cells* (LC) which are each used to emulate a different portion of the logic circuit. A diagram of an LC is shown in figure 2.1. Each logic cell contains a *look up table* (LUT), a small memory whose value encodes a particular boolean logic function for the logic cell. The LUT can be seen as corresponding to a combinatorial circuit, since there is a direct correspondence between the input and the output. In addition, each logic cell contains a flip flop (FF). The output from the LUT may either be registered in the FF or not, depending on the logic circuit which is loaded. The output being registered by a FF or unregistered is controlled by the multiplexer, shown as the rightmost component in figure 2.1. By including the ability to register outputs, the FPGA is able to encode sequential circuits, circuits where the output depends not only on the current input but also on the circuit's state.

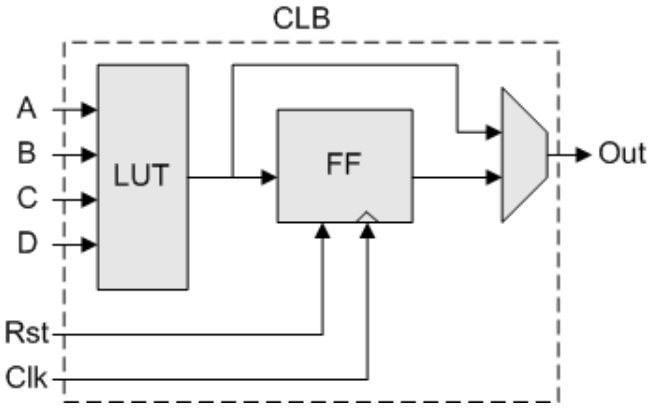


Figure 2.1: FPGA Logic Cell [15]

While logic cells are a fundamental component of FPGA's, modern FPGA's contain some other noteworthy components as well. Many FPGA's contain *digital signal processing* (DSP) slices, which implement some common operations such as multiplication more efficiently than LC's [31]. FPGA's can also include embedded memory. In the Xilinx device family, this embedded memory is referred to as *block RAM* (BRAM) [32].

2.1.1 FPGA's In Comparison With Software

As seen in the previous subsection, FPGA's have a completely different computational model than the CPU. This difference between the two allows for FPGA's to often be more efficient, but at the same time applications targeting FPGA's face a different set of constraints than when targeting CPU's. In particular, FPGA's are constrained by the number of resources

(LC, BRAM, and DSP) available. The distinctions between the two platforms will be important in evaluating the research questions of this thesis and are worth considering in detail. In particular, significantly improving DPR reconfiguration speed has the possibility of relaxing the resource constraints faced by FPGA's and possibly making new classes of applications worth implementing on FPGA's.

A key difference between the two platforms is that applications implemented on FPGA's are generally more efficient than software implementations. One source of inefficiency for the CPU is that in many cases no instruction computes precisely what's needed for an application. An example of this can be seen in an early encryption scheme DES. DES involves specific bit permutations which can be implemented very efficiently as a logic circuit, but are very inefficient when implemented in software [19]. Since there is no specific CPU instruction for the required bit permutations, software implementations will need to compose multiple instructions for what should otherwise be a simple computation. This limitation of CPU's can be mitigated by modifying the application. For example, AES, a more modern encryption scheme than DES, was intentionally designed to allow for a more efficient software implementation [19]. CPU's may also add application specific instructions (indeed, there's a specific instruction set for AES on Intel processors [9]). In general, though, it's impossible for a CPU instruction set to anticipate all of the computations needed for applications to be efficient.

The efficiency difference between FPGA's and CPU's has additional causes as well. These are well illustrated in a work by Sirowy and Forin, who ran three programs with various CPU architectures and FPGA implementations [21]. One of their findings is that instruction fetching adds a significant performance overhead for the software implementations. In FPGA's, nothing analogous to instruction fetching is necessary since the logic circuit implementing the program remains loaded on the FPGA. Similarly, processors can be limited in terms parallelism. Since FPGA's and CPU's are structured so differently, it's challenging to make a direct comparison in terms of the amount of parallelism they support. CPU's may offer parallelism at the level of having multiple cores or by supporting SIMD. With enough FPGA resources, however, it's possible to introduce a large degree of parallelism and apply optimizations such as pipelining. Sirowy and Forin found that a basic software implementation was typically much less performant than optimized FPGA implementations. To narrow this gap, they found that the software implementation could be run on a CPU supporting

additional parallelism. In particular, their study investigated using a super-scalar CPU.

While applications can be implemented very efficiently on FPGA's, these implementations are constrained by the amount of available resources in a way that software is not. When implementing an application in software, additional features and functionality can be added at the expense of a larger executable. The CPU only needs to fetch the instructions being executed at any particular time from memory, limiting the extent to which executable size is an important constraint. In FPGA implementations, on the other hand, the full logic circuit bitstream must be loaded onto the FPGA for the application to execute. [1] Because of this, the number of LC, DSP and BRAM units on an FPGA becomes an important constraint.

2.1.2 FPGA Resource vs. Performance Trade-Offs

Increasing the importance of the FPGA resources is the fact that there is typically a trade-off between the number of FPGA resources used in an application's implementation and the resulting performance. One particular optimization enabling a trade-off between FPGA resources and performance is loop unrolling [11]. When a particular computational step is executed in a loop, this step can be instantiated multiple times in the FPGA, reducing the required number of loop iterations. This can be seen in figure 2.2a. In this figure the original loop body, corresponding to the circuit R, is instantiated twice. Unrolling the loop by a factor of two then reduces the number of loop iterations required by a factor of two. At the same time, though, this may result in a reduced clock frequency since the logic in the loop body increases. The reduced clock speed reduces the performance improvements from unrolling the loop. This reduced clock speed may be remedied, though, by introducing a register within the loop body, an optimization known as pipelining [11]. This additional pipelining step can be seen in figure 2.2b.

The exact FPGA resource and performance trade-off available depends on the details of the application being implemented. A comprehensive example of this is given by Homsirikamol, Rogawski, and Gaj, who investigated the throughput vs. resource trade-off available for different cryptographic hash functions [11]. Their results can be seen in figure 2.3. From the figure, it's possible to see that some of the hash functions can be optimized for throughput more effectively than others. In the case of SHA-2 no optimizations were pos-

[1] If DPR is being used, only the active bitstreams must be loaded.

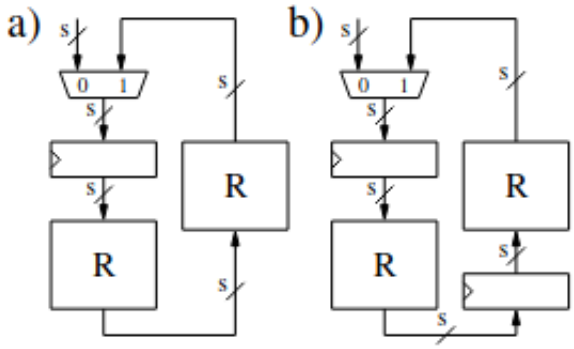


Figure 2.2: Loop unrolling. a) The loop body R is instantiated twice. b) An additional register is introduced in the loop body to improve the clock frequency. Reproduced with permission from [11].

sible other than simply instantiating another SHA-2 instance to compute additional hashes in parallel. In the case of Groestl, on the other hand, the additional throughput gained from optimizations is greater than what would be possible in simply instantiating another instance.

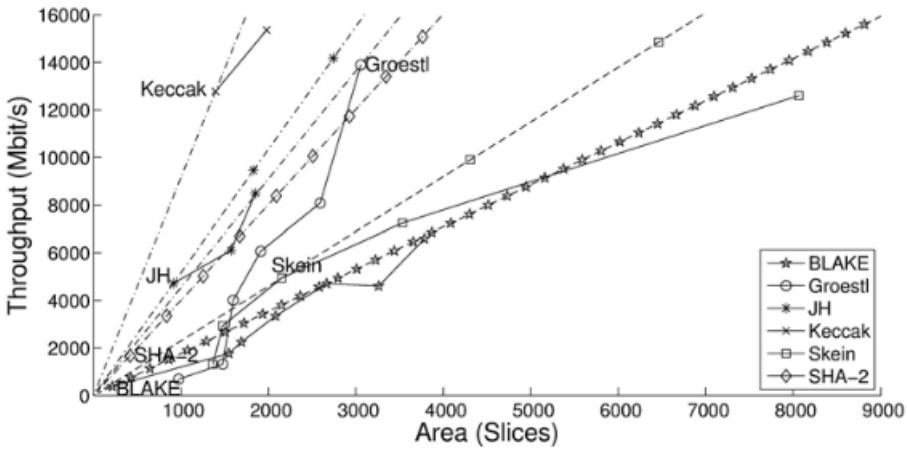


Figure 2.3: Throughput vs FPGA resources of cryptographic hash functions. Reproduced with permission from [11].

While increasing the FPGA resources may have substantial benefits for performance, it does come at a cost. FPGA's containing more resources draw more power compared to smaller FPGA's in the same device family [29]. Additionally, larger FPGA's within a particular device family will be more expensive.

2.1.3 Dynamic Partial Reconfiguration

As mentioned in the introduction, this thesis focuses in particular on dynamic partial reconfiguration. This feature allows for portions of the FPGA to reconfigured with a new bitstream while the rest of the FPGA continues computation. The benefits of this feature are well summarized in a survey paper from Vipin and Fahmy: "the effective logic density of the chip can be increased by time-multiplexing hardware resources between mutually exclusive computations, thereby allowing a larger application to be contained on a smaller chip." [25]. In many cases, applications don't require all functionality or computations to take place all of the time. By dividing an application into finer grain components and reconfiguring between them, each component has access to a greater number of resources than it would without DPR. This can allow for performance optimizations as discussed in section 2.1.2 or simply including the same amount of application functionality on a smaller device.

Among commercial FPGA vendors, Xilinx and Altera are the main vendors supporting DPR [25]. The exact terminology used by the two vendors varies, however, the overall process for DPR is the same. Instead of producing one single bitstream to load onto the FPGA, applications using DPR are synthesized to also produce partial bitstreams. These partial bitstreams each encode only a specific module of the overall application which is intended to be reconfigured. During the runtime of the application, the application can execute a partial reconfiguration by loading the partial bitstream from memory and forwarding it to a special reconfiguration port. In Xilinx FPGA's, this port is referred to as the internal configuration access port (ICAP) [35].

DPR also introduces new constraints which must be taken into account, however. During the implementation of an application, it's necessary to decide how to divide the application into different modules which will be reconfigured at runtime. It's not possible to determine at runtime which portions of the application to reconfigure. This is partly because the partial bitstreams must be synthesized beforehand.

Similarly, the exact number of reconfigurable regions of the FPGA must also be decided before runtime. In order for these reconfigurable regions to run a particular module, it must have sufficient resources allocated to it. It's therefore also necessary to decide which modules will run on which reconfigurable regions. This problem of deciding the number of

reconfigurable regions and how to group modules is known as *partitioning* [25]. Based on this specification, the reconfigurable regions are assigned physical locations on the FPGA during design time in a process known as *floorplanning* [25].

Since reconfigurable regions are mapped to fixed regions of the FPGA, the partitioning process has the potential to introduce inefficient use of FPGA resources. For instance, if a module requiring 3,000 LC's and a module requiring 1,000 LC's are assigned to the same reconfigurable region, the region will need to be allocated at least 3,000 LC's during floorplanning. When the smaller module is executing, the remaining 2000 LC's in the reconfigurable region will be unused.

An additional constraint on DPR is that it's not possible to reconfigure an arbitrary number of reconfigurable regions in parallel. The number reconfigurations is limited by the number of reconfiguration ports, and even the high end Xilinx UltraScale devices only allow one active ICAP [30]. This limits the application to only reconfigure one reconfigurable region at a time.

Because the reconfiguration process requires forwarding the partial bitstream to the ICAP, the time needed to reconfigure a reconfigurable region using DPR is proportional to the size of the partial bitstream. The size of the partial bitstream is, in turn, proportional to the number of FPGA resources it uses.

As a reference point for the exact amount of time needed for reconfiguration, it's interesting to consider the current Xilinx Virtex UltraScale+ models. These devices contain a 32-bit ICAP and support a maximum dynamic reconfiguration port clock frequency (F_{DRP_CLK}) of 250MHz [33, 30]. This gives a theoretical bandwidth of 1 GB/s. There aren't many papers reporting typical bitstream sizes, but one paper does report that they obtained bitstreams of 338 KB when synthesizing simple image convolution filters for another Xilinx device [2]. We would then expect reconfiguring these bitstreams to take approximately 0.338 ms on a high end device, with application modules which require more resources taking longer. This back of the envelope calculation leaves out some factors which may reduce reconfiguration time [2] and lacks more detailed information about typical bitstream sizes. Nevertheless, it's useful to have at least some reference point for reconfiguration speeds.

[2]These factors will be addressed in the following the following section.

2.2 Improving Reconfiguration Speed

Because of both the potential that DPR offers for better utilizing FPGA resources and the bottleneck that slow reconfiguration poses, much prior work has investigated improving the reconfiguration process. As a result, reconfiguration speeds of FPGA's have been steadily improving, and some techniques such bitstream compression have further improved reconfiguration speeds. Nevertheless, the reconfiguration process of current FPGA's still requires writing the bitstream to a single reconfiguration port. This makes the opportunity of improving reconfiguration speeds by orders of magnitude with 3D packaging particularly interesting.

One technology which already offered extremely fast reconfiguration speeds, switching between configurations in nanoseconds, was the Multi-Context FPGA. These devices were mainly developed 1990's and are not commonplace today [25]. To support such fast reconfigurations, the devices maintained the various configurations on the FPGA itself, rather than reading them from an external memory. One such device from Trimberger, Carberry, Johnson and Wong stored eight configurations on on-chip memory and could switch between configurations in 30ns [24]. Multi-Context FPGA's tended to require a high amount of power, though, limiting their usefulness [25]. Much has changed in the field of FPGA's since Multi-Context FPGA's were actively researched. Still, given that the reconfiguration speeds were so fast, closer examination of the literature on Multi-Context FPGA's could reveal some relevant information: for example, how to best structure applications to take advantage of such fast reconfiguration times.

Another approach which has been used to improve reconfiguration speed is bitstream compression. The idea behind this is that compressing the bitstream reduces the amount of data which must be read from memory. Many techniques have been developed in academic work, and Xilinx has adopted it's own support for bitstream compression [25]. In their survey paper on DPR, Vipin and Fahmy explain that "bitstream compression is useful when bitstream transfer time from external memory to the FPGA is considerably higher than the time taken to send the bitstream to configuration memory" [25]. While this approach makes more efficient use of bandwidth for accessing external memory, the ICAP still remains a bottleneck as it needs to process the incoming bitstream and load it into configuration memory. Support for bitstream compression in the ICAP itself could then

make bitstream compression even more effective. Xilinx has some support for this with the "multiple frame write register" [25].

FPGA vendors have also been steadily improving reconfiguration speeds. Figure 2.4 shows the improvements in theoretical throughput of Xilinx devices. The figure stops in 2015, but the theoretical throughput of the current Xilinx Virtex UltraScale+ is 1 GB/s [3]. With these improvements, the current FPGA devices still require the bitstream to be passed to a single ICAP. The premise of this thesis is that configuration memory and FPGA logic could be combined using 3D packaging and that by doing so, the reconfiguration process may also be parallelized. Assuming that such technology is feasible, it would offer an even more substantial improvement than what is shown in figure 2.4.

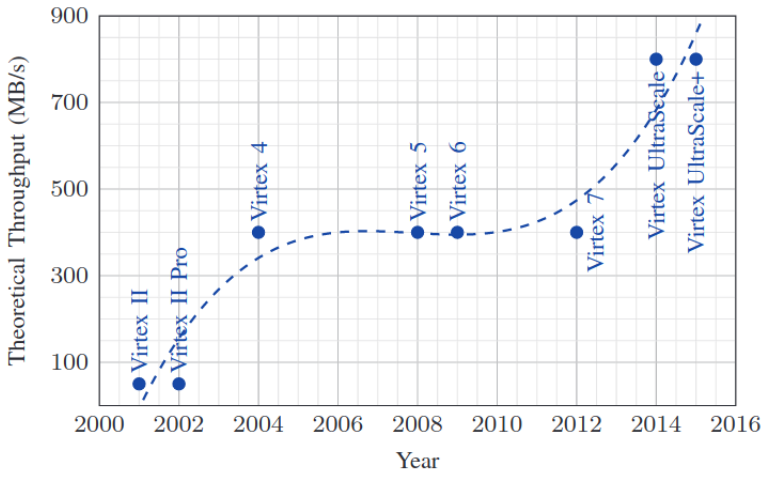


Figure 2.4: Improvements in Xilinx FPGA reconfiguration throughput. ©2016 IEEE [2].

2.3 3D Packaging

3D packaging, the technology which this thesis supposes could provide dramatic improvement in reconfiguration speed, has seen significant recent progress. This technology is already used in commercial products, with one of the most advanced examples being the Intel Lakefield CPU which uses their 3D packaging technology Foveros [16].

A key benefit of 3D packaging is that it can allow for both shorter and a higher number of interconnects between components. This is illustrated in figure 2.5. Shortening the interconnect length is important since it reduces both the latency and the power usage. In

[3] This is based on the 32-bit ICAP and 250MHz maximum reconfiguration clock frequency [33, 30].

particular, “as the parasitic capacitance in microelectronic packages is proportional to the interconnection length, the total power consumption in 3D packages is also reduced because of the reduced parasitic capacitance” [16].

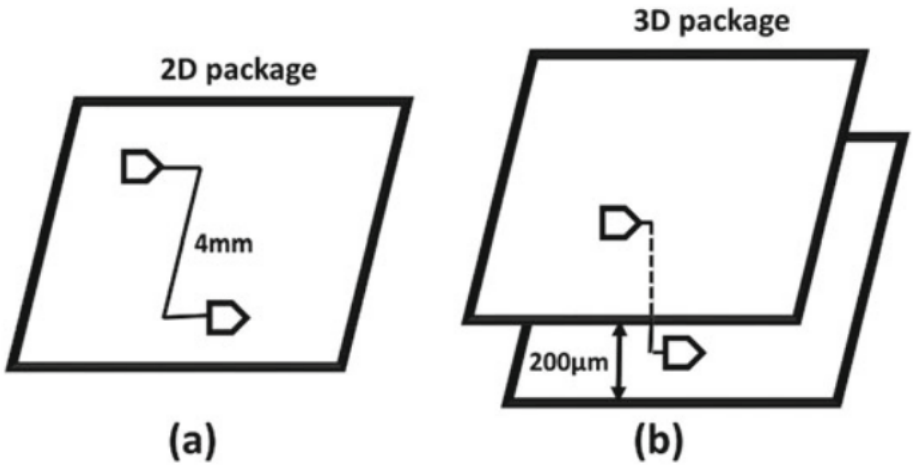


Figure 2.5: Comparison of wire lengths in 2D and 3D packages [16]

For many applications, a primary bottleneck in performance is accessing memory. The case of Foveros, shown in figure 2.6, is then particularly interesting since it combines both a CPU and memory. The 3D packaging has the advantage of increasing the available bandwidth of accessing memory and reducing the power necessary. There are different varieties of 3D packaging which exist, and Foveros in particular is an example of *Heterogeneous 3D integration*. A benefit of this approach is that it “allows the use of the best available technology node for each chiplet with different function to maintain maximum performance” [16]. In other words, the DRAM and the CPU are able to be developed with the most appropriate respective technology and then packaged together.

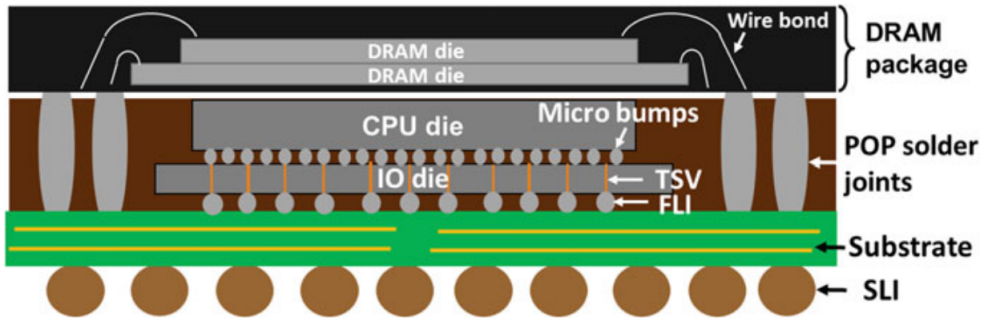


Figure 2.6: Foveros 3D packaging [16]

A recent device from Xilinx, the Virtex UltraScale+ high bandwidth memory FPGA, pack-

ages an FPGA along with memory [34]. Whereas an off-chip DDR4 DIMM memory may allow for a bandwidth of 21.3 GB/s, the high bandwidth memory FPGA has a bandwidth of 460GB/s and a size of 16GB [34]. This memory is intended for use by applications, however, rather than for improving reconfiguration speeds. Without other changes in the FPGA, the reconfiguration port remains a bottleneck in the reconfiguration process.

3D packaging technology is progressing rapidly and CPU's and FPGA's are already being packaged together with memory to allow for better memory bandwidth. It's reasonable to believe that this same technology could be applied used to improve the reconfiguration process, potentially quite soon.

2.4 DPR Schedulers

DPR involves running various components of an application over time while respecting the constraints of the FPGA. This introduces a unique scheduling problem which has already been addressed by several works. This prior work on scheduling is especially relevant to the research question at hand, since the schedulers introduce models of applications using DPR and determine what performance applications are able to obtain.

2.4.1 Deiana et. al Scheduler

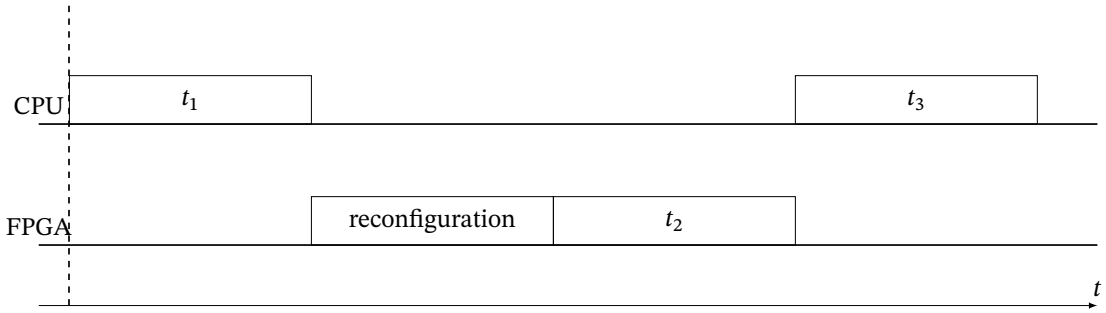
One scheduler, which will be used extensively in this thesis, was developed by Deiana, Rabozzi, Cattaneo, and Santambrogio [7]. This scheduler models the problem as a *mixed integer linear program* (MILP). This scheduler is unique in that it has an accurate model of the FPGA constraints, incorporates some optimizations which can reduce the schedule makespan, and allows for optimizing not only the schedule makespan, but also the peak power and energy consumption.

Existing schedulers may be classified as being either heuristic algorithms or exact algorithms: the exact algorithms producing optimal schedules but at the cost of typically being more time consuming. The scheduler from Deiana et. al, consists of both a heuristic and exact approach. First, the authors propose an MILP formulation of the scheduling problem which may be solved to obtain optimal schedules. Solving the MILP formulation to optimality can become time consuming, however. To work around this, the authors also propose

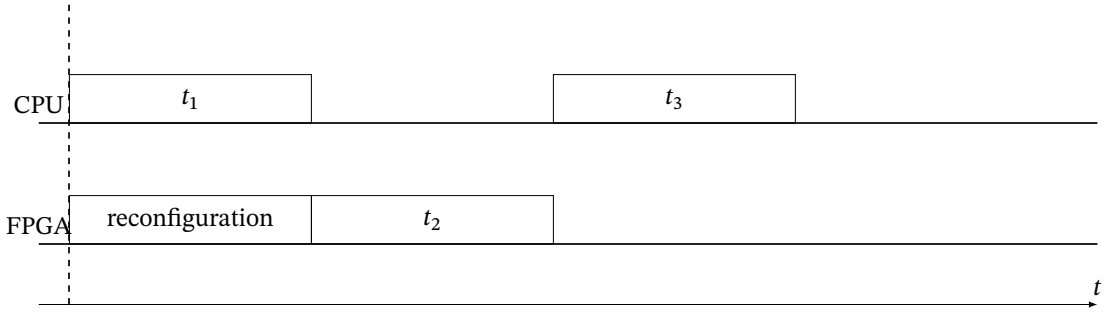
an iterative scheduler which is based on the MILP formulation. The iterative scheduler requires less time to obtain a schedule, but may produce non-optimal schedules as a result.

In the scheduling model, applications are modeled as a directed acyclic graph of tasks. Each task must be scheduled exactly once, with edges between tasks representing dependencies. Each task may have multiple implementations for the scheduler to choose from, including software implementations and hardware implementations. Because of its support for both software and hardware implementations, this scheduler is usable for heterogeneous architectures containing a processor and FPGA. The task implementations may also have different runtimes and resource requirements, giving the scheduler flexibility to make performance vs. area trade-offs where necessary.

To avoid unnecessary delays due to reconfigurations, the scheduler allows for two optimizations: module reuse and configuration prefetching. In module reuse, some tasks may share common FPGA implementations. If such tasks are scheduled consecutively on the same FPGA partition and using the same implementation, then no reconfiguration is necessary between them. In configuration prefetching, a configuration may begin for a task before the dependencies of the task have completed execution. An example of this can be seen in figure 2.7. In the figure, task t_2 depends on task t_1. Waiting for task t_1 to complete before beginning the configuration for task t_2 unnecessarily increases the schedule makespan.



(a) Without prefetching



(b) With prefetching

Figure 2.7: Optimizing execution time with prefetching

2.4.2 FRED Real Time Framework

As a first step in developing *FRED*, a set of tools for using DPR, Biondi et al. developed a model of the FPGA for real-time tasks [2]. This model is interesting as it accurately accounts for reconfiguration time and supports heterogeneous architectures. The authors then use this model to develop a schedulability analysis for real time task sets using an MILP formulation. While this thesis doesn't investigate real-time applications, this model is quite detailed and worth explaining here.

In their model, Biondi et al. account for tasks which are themselves composed of hardware and software subtasks. The hardware subtasks are always preceded and followed by software subtasks, which prepare and retrieve data. The subtasks are characterized by their worst case execution time. As the model focuses on real-time tasks, tasks also have deadlines.

For the reconfiguration process, the FRED model accounts for reconfiguration time depending on the size of the hardware task. As there are limited FPGA resources, hardware subtasks need to wait for space on their partition to become free. To manage this process, the model schedules the hardware tasks with a first-in-first-out queue for each partition. Using a FIFO policy ensures hardware subtasks eventually progress. Additionally, the reconfiguration interface can only execute one reconfiguration at a time. To manage contention for the reconfiguration interface, the model includes a ticket based scheduling policy. This policy serves the earliest requested reconfiguration first. It can be adapted for both preemptive and non-preemptive reconfiguration interfaces.

Special consideration needs to be given to how software subtasks retrieve data from a preceding hardware subtask. If the software subtask and hardware subtask need to be active at the same time to transfer data, this can cause undesirable delays for other hardware tasks. The software subtask may not be able to execute immediately, and the hardware task would continue to prevent other hardware tasks from using its resources. Even if this is a rare occurrence and doesn't have a large effect on performance, this additional delay is problematic for real-time systems. To remove the possibility of such delays, the model proposes that hardware and software subtasks communicate through a shared memory. After writing the results to memory, the partition of the hardware subtask can be reconfigured.

While the model gives a detailed description of the reconfiguration process and communi-

cation between tasks, it doesn't propose a way to partition the hardware subtasks. A partitioning must already be developed before performing the schedulability analysis. Even if a task set with a given partitioning is found to not be schedulable, it may still be the case that it's schedulable with another partitioning. The model also doesn't take into account configuration prefetching. Adding consideration of prefetching could potentially allow for more task sets to be schedulable.

3 PROPOSED SOLUTION

3.1 Definition of the Model

3.1.1 Relevant Aspects To Model

As outlined in section 1.3, this work studies the impact of reconfiguration speed only at the level of a modeled workload and FPGA. In order for this approach to yield plausible results, features of actual FPGA's and applications should be taken into account to the greatest extent possible. The following aspects of real world FPGA's appear to be particularly important to model:

Resource Constraints of the FPGA The resources available on an FPGA, the number of LC, DSP, and BRAM units, constrains how much functionality an application can implement and how performant it may be. To avoid overestimating the performance of a given application on a particular FPGA, the application's implementation should not be allowed to exceed the available resources.

Resource vs. Performance Trade-off As seen in section 2.1.2, applications targeting the FPGA can typically increase performance at the expense of resource usage. The exact trade-off available varies by application. A key benefit of DPR is that different modules in a program may be time-multiplexed, giving the modules access to additional resources and allowing them to be implemented in a more performant way. To accurately model the

potential benefits of improving reconfiguration speed, the model should then account for the ability to trade-off performance and FPGA resources.

Reconfiguration Time In current FPGA's, the time needed to reconfigure an FPGA region is proportional to the size of the reconfigurable region. This constraint potentially limits the benefits of DPR, though. While DPR will allow application modules to be implemented with more resources and therefore be more efficient, these larger implementations will require a longer time to be reconfigured.

Partitioning of the FPGA As described in section 2.1.3, the number of reconfigurable regions and the amount of FPGA resources to allocate to them must be decided at design time. This inability to map application modules to any arbitrary region of the FPGA at runtime can result in FPGA resources remaining idle at times. It's not immediately clear whether improving reconfiguration speed would improve or worsen this phenomenon, and it may be the case that the effect of this constraint tends to be small. Nevertheless, the need for partitioning the FPGA into reconfigurable regions is interesting to include in the model to remain as accurate as possible.

Ability to Run As Software It could potentially be the case that applications which stand to benefit from improved reconfiguration speed are currently run as software rather than targeting an FPGA. For instance, such applications might contain too much functionality to easily fit on an FPGA and the overhead of using DPR could be too high at current reconfiguration speeds. To avoid giving a pessimistic estimate of the current performance of these applications, which would overestimate the usefulness of improving the reconfiguration speed, the model should take into account the ability of applications to run either as software or on an FPGA. Some real world platforms, such as the Xilinx Zynq, incorporate both a CPU and FPGA, allowing portions of an application to be implemented as software and other portions to be implemented for the FPGA. Incorporating both a CPU and FPGA into the model will allow for evaluating whether applications may transition from software implementations to FPGA implementations as reconfiguration speed is improved.

3.1.2 Model Definition

With the above considerations in mind, it's possible to define a model of the applications and the hardware which they will be run on. The applications, which will also be referred to as workloads, consist of a set of tasks which must be run. Each task will have multiple possible implementations which may be used to run it. These implementations are either software implementations or FPGA implementations. The only parameter used in describing software implementations is their runtime. The FPGA implementations are characterized by their runtime as well as the amount of FPGA resources which they require.

Each task must be run exactly once, meaning tasks are not recurring. In addition, the tasks are considered to be non-preemptive. While the scheduler of Deiana et. al. considered the possibility of implementations supporting multiple tasks, this work will assume that each implementation can only be used to execute a single task. This assumption removes the possibility of the "implementation reuse" optimization accounted for by Deiana et. al. It would certainly be interesting to consider the case of recurring tasks or other more complex types of workloads. For this work, however, the modeled workloads are kept quite simple.

The platform used for running workloads consists of a CPU as well as an FPGA. The CPU is parameterized by the number of CPU cores, and it's assumed that all software implementations will require only one CPU core to run and that the CPU cores are homogeneous. The FPGA is parameterized the amount of LC, DSP, and BRAM units available.

For running a workload, the FPGA will be partitioned and each partition will be allocated an amount of resources. Each partition may be used to run one task at a time, with reconfigurations being required between tasks on a given partition. The total amount of resources occupied by the partitions must be within the limits of the FPGA, and task implementations may only be run on partitions with sufficient resources. This is a stronger condition than simply requiring the total FPGA resource limits to be respected at any given point in time and is more consistent with the actual functioning of FPGA's.

Initially, we assume that only one reconfiguration may be run by the FPGA at once, but this condition will be relaxed later in section 3.3. It is also assumed that the first task implementation run on a given FPGA partition also requires a configuration. This assumption is in contrast with the model of Deiana et al, which assumes that the first task on a partition does not require a configuration. When the FPGA first boots up, no configuration is loaded and

some configuration will therefore be necessary, so the assumption seems reasonable. In the worst case, the impact on performance will be limited to a constant factor: the time needed to configure the entire FPGA once. Assuming that initial implementations on the FPGA also require a reconfiguration becomes somewhat useful in section 3.3, since it removes the need for special handling of the initial implementations.

There are many possible benefits that could result from improved reconfiguration speed. In terms of performance, a given workload on a given FPGA may be executed within a shorter amount of time. In terms of efficiency, a smaller FPGA could potentially be used to execute a given workload within a fixed amount of time. Similarly, the size of the FPGA and the amount of execution time could be held constant, with the objective being to maximize the number of tasks executed.

This study in particular will focus on performance. For a fixed platform and workload, it will investigate how reconfiguration speed influences the *makespan*, the time needed to execute all tasks. This problem of determining the makespan for a given platform and workload is essentially a scheduling problem.

The table 3.1 introduces the actual parameters used to characterize the target platform and workload. The parameter names are chosen to remain mostly consistent with the model of Deiana et al. discussed in section 2.4.1.

Target Architecture	
P_{SW}	Set of all available CPU cores
P_{HW}	Set of FPGA reconfigurable regions (partitions)
P	Set of all processing units: $P_{SW} \cup P_{HW}$
R	Set of different FPGA resources (LC, DSP, BRAM)
$maxRes_r$	Amount of resource $r \in R$ available on the FPGA
s_r	Amount of time needed to reconfigure one unit of resource $r \in R$
Workload	
T	Set of tasks
I_{SW}	Set of software implementations of tasks
I_{HW}	Set of FPGA implementations of tasks
I	Set of all implementations, $I_{SW} \cup I_{HW}$
TPI	Set of compatible tasks, implementations, and processing units: $TPI \subseteq T \times P \times I$
l_i	Amount of time needed to complete running implementation $i \in I$
$oc_{i,r}$	Amount of resource $r \in R$ needed by implementation $i \in I_{HW}$

Table 3.1: Model parameters

While most of the parameters are straightforward, P_{HW} warrants some further discussion. The resources allocated to all FPGA partitions should be within the limits of the available

FPGA resources. Aside from that, however, there is no limit on the number of partitions that there may be. Setting P_{HW} such that $|P_{HW}| = |T|$ is sufficient, since it ensures that every task may potentially be run in parallel by the FPGA. Since a large number of FPGA partitions may make the model more complicated, though, it may make sense to try to limit the size of P_{HW} further. A solution from Deiana is to solve a variant of the bin packing problem to determine the maximum number of tasks which may be run in parallel while remaining within the FPGA resource limits [6]. While this requires additional computation, it makes later use of the model simpler. This bin packing approach is used in this work.

It's also worth clarifying the parameter s_r, the amount of time needed to reconfigure one unit of resource. This work often discuses reconfiguration speed, however, the model itself is parameterized by the inverse of the reconfiguration speed. This may lead to some confusion in terminology, especially between "reconfiguration time" referring to the total time spent by the FPGA performing reconfiguration and referring to the reconfiguration time per unit of bitstream. One advantage, though, is that representing the extreme case of $s_r = 0$ is arguably easier than considering the limit of reconfiguration speed approaching infinity.

3.2 Applying the Deiana et. al Scheduler

The model detailed above is essentially a more simple version of the scheduling problem considered by Deiana et. al (see Section 2.4.1). The only inconsistency with their scheduler is that it doesn't require configurations for the first task executed on each FPGA partition. This inconsistency can be patched easily, though, by adding an additional constraint to their MILP formulation. With the models consistent, the Deiana et. al scheduler can then be applied to the model considered here.

In particular, their formulation includes a binary variable $cft_{t,c}$ which is set to 1 if task t is the first task scheduled on the FPGA partition c. Their model also defines a binary variable $rtt_{rt,t}$ which is set to 1 if task t requires a reconfiguration rt. Finally, the set RT specifies all of the reconfigurations which may be necessary. A reconfiguration can then be required for the first task on each partition with the constraint: [1]

[1]There are minor additional details which are worth mentioning to be precise. An additional reconfiguration should be added to RT so that $|RT| = |T|$. Also, the notation used here is slightly inconsistent with Deiana et al. as they use C^h instead of P^{HW}.

$$\sum_{rt \in RT} rtt_{rt,t} \geq \sum_{c \in P^{HW}} cft_{t,c} \qquad \forall t \in T \tag{3.1}$$

To determine the makespan of a given workload run on a given platform, a possible approach is to then simply run the scheduler from Deiana et al. In particular, their work proposes two schedulers: an MILP formulation of the problem, which produces an optimal schedule for the given workload and platform, and an iterative scheduler, which will generally not produce an optimal schedule, but which is much faster than solving the MILP formulation.

Since the ultimate goal of this research is to draw conclusions based on the results of the model, the selection between applying the MILP and iterative schedulers has important implications. In using the MILP formulation, it will be possible to have more confidence when drawing conclusions from results of the model. As the schedules that the MILP formulation produces will be optimal, any differences resulting from changing reconfiguration speed will likely indicate some actual phenomenon. In using the iterative scheduler, on the other hand, there is the risk that the scheduling algorithm produces better results at slow reconfiguration speeds and worse results at faster reconfiguration speeds (or vice versa). Regardless of the results of iterative scheduler, it will be difficult to tell if any patterns are actual effects of changing the reconfiguration speed, or simply side-effects of this particular scheduling algorithm.

There is no particular aspect of the iterative scheduler which gives reason to believe that it has any behaviour like what is mentioned above. This is simply a possibility that should be taken into account since its resulting schedules are not generally optimal. In addition, any results from the model about the impact of reconfiguration speed should be held with some skepticism anyway, since the modeled workloads possibly differ significantly from real world workloads. Still, based on this analysis the MILP scheduler is preferable.

Solving the MILP formulation to optimality proved to be very computationally expensive, though, making this approach infeasible. A full discussion of the time needed to solve the MILP formulation is given in Section 4.3. In summary, solving the MILP formulation with Gurobi, a powerful commercial solver, took 9 hours for a workload of 6 tasks when run on a modest laptop computer.

The paper from Deiana et. al doesn't report the absolute amount of time needed to solve

the MILP formulation, but they do report that it is generally long. One contributing factor which they report is that the number of variables in the formulation grows quadratically with the number of tasks. In particular, for all pairs of tasks which could possibly be run at the same time, binary variables are added to encode which task occurs before the other. This requires a quadratic number of variables and may make the formulation more challenging to solve.

The large amount of time needed to solve the MILP formulation to optimality makes it less suitable for investigating the research questions of this thesis. In studying the research question, it would be particularly interesting to consider workloads with many tasks. It could potentially be the case, for example, that workloads composed of more tasks experience a more significant performance improvement. To detect patterns such as this, it's useful to be able to study workloads with more tasks than the amount which can be used with the MILP formulation. The computer used for testing the MILP formulation is admittedly quite old, and it may be possible to obtain some results by using a more powerful server. Nevertheless, it's worthwhile to evaluate other possible techniques.

3.3 Determining Schedule Lower Bounds

To be able to draw stronger conclusions about the possible performance improvement from improved reconfiguration speed, this work introduces a method for calculating *lower bounds* on the schedule makespan. The iterative scheduling algorithm described in the previous section already gives a way of determining an upper bound on the schedule makespan. By having both upper and lower bounds on the schedule makespan, it will be possible to draw stronger conclusions on the impact of improving reconfiguration speed. In particular, the lower bound will reveal any bias causing the iterative scheduler to perform significantly better at faster or slower reconfiguration speeds. Calculating the upper and lower bounds on the schedule makespan will prove to be much faster than calculating the optimal schedule, making it possible to analyze larger workloads.

3.3.1 Simplifying The Scheduling Problem

A key source of complexity in the MILP formulation of Deiana et. al is that the start and end times of tasks are explicitly modeled. This is necessary since their model takes into account the following features:

1. Tasks may be dependent on each other. If one task depends on another, it may not begin until the other has finished.

2. Some tasks may have FPGA implementations in common. By running consecutively on the same FPGA partition and with the same implementation, these tasks may avoid the need for a reconfiguration between them.

3. Only one reconfiguration may take place on the FPGA at a time.

The first two considerations are already excluded from the model proposed in section 3.1. By removing the third constraint as well, allowing multiple reconfigurations to take place at a time, the MILP formulation can simplified. It is no longer necessary to model the start and end times of the tasks, and it suffices to track only which task is executed with which implementation and with which FPGA partition or CPU core.

The Deiana et. al formulation also has an accurate, but as a result more complicated, formulation of reconfigurations, which can be simplified. In particular, their MILP formulation models reconfigurations as special tasks and enforces a correspondence between normal tasks and reconfiguration tasks. Including these reconfiguration tasks allows for modeling the fact that the reconfiguration time depends on the amount of resources allocated to a particular FPGA partition, as opposed to the amount of resources required by the implementation being configured. By making the reconfiguration time only depend on the resources required for the implementation being configured, it's possible to remove the need for reconfiguration tasks in the MILP model.

3.3.2 MILP Formulation

With these simplifications, it's possible to model the scheduling problem using a straightforward MILP formulation. In addition to the parameters described in section 3.1, it's necessary to introduce some variables. The binary variable $x_{t,p,i} \in \{0, 1\}$ indicates if the schedule

processes task $t \in T$ on region $p \in P$ with implementation $i \in I$: 1 indicating that the task-region-implementation combination is scheduled and 0 that it is not. The variable is defined for all $(t, p, i) \in TPI$. The variable $u_{r,p}$ represents the amount of FPGA resources of type $r \in R$ allocated to FPGA region $p \in P_{HW}$ in the schedule. Finally, a variable z is also introduced to represent the makespan of the schedule. The simplified scheduling problem can then be formulated as the following MILP:

$$\text{minimize} \quad z \tag{3.2a}$$

subject to

$$\sum_{(t,p,i)\in TPI|t=t'} x_{t,p,i} = 1 \qquad \forall t' \in T, \tag{3.2b}$$

$$\sum_{(t,p,i)\in TPI|p=p'} (l_i + \sum_{r\in R} s_r \cdot oc_{i,r}) x_{t,p,i} \leq z \qquad \forall p' \in P_{HW}, \tag{3.2c}$$

$$\sum_{(t,p,i)\in TPI|p=p'} l_i \cdot x_{tpi} \leq z \qquad \forall p' \in P_{SW}, \tag{3.2d}$$

$$\sum_{(t,p,i)\in TPI|p=p',t=t'} oc_{i,r} \cdot x_{t,p,i} \leq u_{r,p} \qquad \forall p' \in P_{HW}, t' \in T, r' \in R, \tag{3.2e}$$

$$\sum_{p\in P_{HW}} u_{r,p} \leq maxRes_r \qquad \forall r' \in R, \tag{3.2f}$$

$$x_{t,p,i} \in \{0, 1\} \qquad \forall (t, p, i) \in TPI \tag{3.2g}$$

The constraint 3.2b ensures that each task is processed using exactly one implementation on exactly one component. The constraints 3.2c and 3.2d ensure that the makespan z has the correct semantics. For each component $p \in P$, the makespan z should be long enough for the component to process all tasks which are assigned to it. On FPGA components, treated with constraint 3.2c, the additional term $\sum_{r\in R} s_r \cdot oc_{i,r}$ is added to model the reconfiguration time needed.

In constraint 3.2c, it's possible to see how making reconfiguration time depend directly on the implementation size helps simplify the model. If the reconfiguration time were to instead depend on the resources allocated to the partition, $u_{r,p}$, the constraint could be expressed as shown in equation 3.3. This constraint is not linear, however, since variables $u_{r,p}$ and $x_{t,p,i}$ are multiplied. To keep the formulation linear, it would likely be necessary to introduce special reconfiguration tasks in a similar way to the Deiana et al formulation.

$$\sum_{(t,p,i)\in TPI|p=p'} (l_i + \sum_{r\in R} s_r \cdot u_{r,p}) x_{t,p,i} \qquad \forall p' \in P_{HW} \tag{3.3}$$

The constraint 3.2e ensures the correct semantics for $u_{r,p}$, the resources used by FPGA component p. Each FPGA component must have enough resources allocated to it to run the tasks which are assigned to it. An alternative way of formulating this this would be adding a constraint $oc_{i,r}x_{t,p,i} \leq u_{r,p}$ for every possible combination of FPGA component $p \in P_{HW}$, task $t \in T$, resource type $r \in R$, and implementation $i \in I_t$.

Finally, constraint 3.2f ensures that the resources allocated to the FPGA components don't exceed the total resources of the FPGA.

3.3.3 Verifying That A Lower Bound Is Provided

A well known result in integer linear programming is that a *relaxation* can be used to find dual bounds. In the case of a minimization problem, such as the scheduling problem at hand, a dual bound is a lower bound. [2] For clarity, definition 1 restates the definition of a relaxation. The definition is taken from Definition 2.1 of Wolsey's book Integer Programming [28], with the definition adjusted for the minimization case.

Definition 1. *A problem (RP)* $z^R = \min\{f(x) : x \in T \subseteq R^n\}$ *is a relaxation of (IP)* $z = \min\{c(x) : x \in X \subseteq R^n\}$ *if*

1. $X \subseteq T$
2. $f(x) \leq c(x)$ *for all* $x \in X$

To show that the simplified MILP formulation given in equation 3.2 provides a lower bound on the schedule makespan in the original model, it suffices to show that the simplified model is a relaxation. Instead of a full, rigorous proof of this, a general outline will be given of why this is the case. As a starting point, the differences between the two models are quite small:

1. The original model only allows for a single FPGA partition to be reconfigured at a time, whereas the simplified model allows for an arbitrary number of partitions to be reconfigured at a time.

[2] When the problem is a maximization problem, the dual bound is a upper bound.

2. In the original model, the reconfiguration time depends on the size of the partition. In the simplified model, the reconfiguration time depends on the size of the implementation.

The first adjustment to the original model strictly increases the solution space of possible schedules, since it simply removes the constraint that there is only one reconfiguration at a time. This is therefore consistent with the first requirement of definition 1.

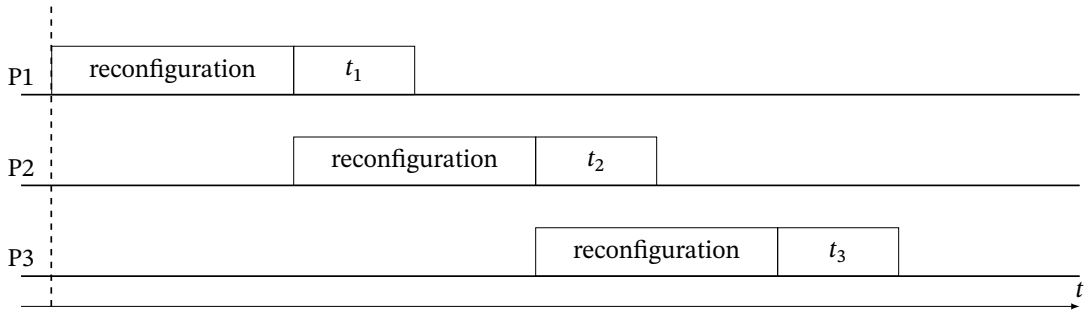
With respect to the second adjustment: the amount of resources used by an FPGA implementation is always less than or equal to the resources allocated to the FPGA partition it is assigned. As a result, the reconfiguration time needed for a task in the original model is always greater than or equal to the reconfiguration time needed for the task in the simplified model. This adjustment again increases the solution space of possible schedules, since the simplified model allows for schedules with less reconfiguration time allocated.

3.3.4 Remarks on the Simplified MILP Formulation

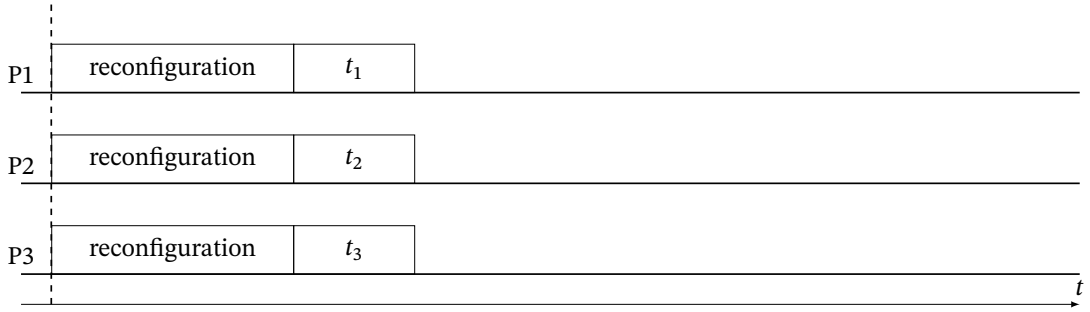
While it's not realistic to assume the computational complexity of the MILP formulation based on the number of variables alone, it's promising that the simplified MILP formulation doesn't have many variables. In particular, there are only $|TPI| + |P_{HW}| + 1$ variables.

Unlike the original MILP formulation from Deiana et. al, however, the simplified MILP formulation isn't immediately useful for scheduling real-world applications. This is because the model makes unrealistic assumptions about the FPGA behaviour. Still, the main motivation in creating this model is that it provides lower bounds on the schedule makespan.

The closer the lower bound provided by the simplified MILP formulation is to the actual optimal schedule makespan the better. This gap between the makespan in the simplified model and the original model can be made arbitrarily large, however. A simple way to see this is to consider a set of tasks which each consist of exactly one FPGA implementation. Even if the FPGA has enough resources to support all tasks executing in parallel, the limitation that one reconfiguration may run at a time will increase the makespan in the original model. In the simplified model, though, the tasks may all execute in parallel as long as the FPGA has sufficient resources. Adding additional tasks in the same way continues to increase the gap in makespan between the two schedules. This can be seen in figure 3.1.



(a) Schedule in original model



(b) Schedule in simplified model

Figure 3.1: Gap in makespan between simplified and original model

3.4 Generation of Workloads and Model Parameters

To evaluate the potential impact of reconfiguration speeds, it is necessary to generate sample workloads and specify the FPGA and CPU parameters. In generating these values, there was a trade-off available between remaining simple and corresponding with actual real-world applications and FPGA's.

For the set of FPGA resources R: LC, DSP and BRAM resources were considered. In analyzing the schedules, though, it's useful to be able to have a full ordering on the resources needed by an implementation $i \in I$. If one implementation uses more LC resources than another implementation but requires fewer BRAM resources, it's more challenging to state which is larger overall. To make the analysis simpler, the model parameters regarding resources - $maxRes_r$, s_r, $oc_{i,r}$ - are kept identical across the different resources. For instance, in all analysis in this work, $s_{LC} = s_{DSP} = s_{BRAM}$. An alternative solution would have simply been to only consider one resource in the model.

The FPGA considered in the analysis is given 10,000 units of each resource: $maxRes_r = 10,000$ for $r \in \{LC, DSP, BRAM\}$. In generating the applications, each task is given a software implementation with a random execution time l_i between 5,000 and 10,000 units of time. Similarly, each task is given FPGA implementations. The exact number of FPGA

implementations is randomly chosen and is between 2 and 10. The FPGA implementations were given increasingly higher resource requirements and increasingly lower execution times. Every task is also given an FPGA implementation requiring fewer than 3,000 resource units. This mimics, at least crudely, the resource vs performance trade-off discussed in section 2.1.2. Pseudo-code for the generation of implementations for each task is given in algorithm 1.

Algorithm 1 Generation of implementations

```
1:  function GenerateImplementationsForTask
2:      implementations ← []
3:      swImplTime ← Random(5000,10000)
4:      implementations.append(SwImpl(swImplTime))
5:      numFpgaImpls ← Random(2,10)
6:      fpgaImplSize ← Random(1, 3000)
7:      fpgaImplTime ← Random(500, swImplTime)
8:      implementations.append(FpgaImpl(fpgaImplSize, fpgaImplTime))
9:      for all 1..numFpgaImpls do
10:         fpgaImplSize ← Random(fpgaImplSize, maxFpgaResources)
11:         fpgaImplTime ← Random(500, fpgaImplTime)
12:         implementations.append(FpgaImpl(fpgaImplSize, fpgaImplTime))
13:     end for
14:     return implementations
15: end function
```

4 RESULTS

4.1 Implementation and Setup

In order to evaluation the approach proposed in chapter 3, an implementation was made in Python. The source code is publicly available and hosted at `https://gitlab.com/clample/fpga-scheduler`. The MILP formulations are implemented using the Python library Pyomo [4, 10]. Gurobi 7.5.2 [1] is used as an underlying MILP solver.

In particular, the implementation contains the optimal MILP based scheduler and the iterative scheduler of Deiana et al, along with the simplified MILP based scheduler described in section 3.3. The implementation of the schedulers using Pyomo was relatively straightforward and doesn't warrant much further explanation here. To improve reproducibility, all data which was used in producing the results of this chapter is also included in the repository. Further explanation of which which data corresponds to which figure is included in the repository's file `README.md`.

4.2 Example Schedules

In order to illustrate how the model introduced in the previous chapter works, this section includes schedules produced for a single particular workload. In particular, figures 4.1, 4.2, and 4.3 depict the optimal schedule of the same workload on the same FPGA, when scheduled with increasingly fast reconfiguration speeds. The horizontal axis represents time, and

simple boxes are included to indicate a task being run on a particular partition at a particular time. Reconfigurations are indicated using crossed lines. For tasks scheduled with an FPGA implementation, the number of LC's used by the implementation is indicated below the task name. The optimal schedules were generated by solving the Deiana et. al MILP formulation.

Figure 4.1 shows a schedule at a slow reconfiguration speed. In the schedule, it's interesting to note that all tasks except for one are executed by the CPU. The reconfiguration speed is exceptionally slow in comparison to the time needed to execute the tasks on the CPU. Even though there exist performant FPGA implementations, the overhead needed to configure them is too large.

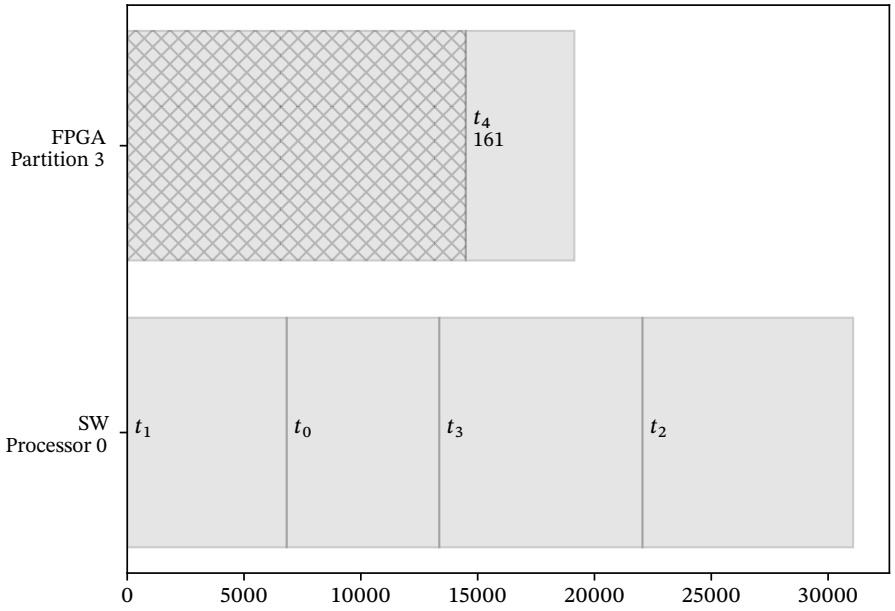


Figure 4.1: Optimal schedule with reconfiguration time per resource unit $s_r = 30$

In figure 4.2, the reconfiguration speed is faster than in the previous example. As a result, all tasks except for one are executed on the FPGA using more performant implementations. By better utilizing the FPGA, the makespan improves by a factor of two in comparison with the previous example. Still, the FPGA implementations which are used require relatively few resources. This is possibly due to the large overhead of reconfiguration.

In figure 4.2, it can also be seen that tasks don't necessarily begin execution at the earliest time possible. Task t_4, for instance, could be executed at an earlier time since there are no

other tasks executed on that partition and it is configured immediately. The reason for this slightly unintuitive behaviour is that the MILP formulation is only minimizing the schedule makespan. There is no constraint requiring that tasks begin as early as possible, so the MILP solver may produce schedules where tasks start later than expected. Still, the MILP solver will not begin tasks later than possible in cases where this actually increases the schedule makespan. As the research questions focus primarily on the schedule makespan, this effect isn't really a concern.

Another interesting feature of figure 4.2 is that the FPGA partitions and CPU are mainly idle. When viewing FPGA partition 4, it's tempting to think that the schedule makespan could be reduced by configuring task t_2 earlier. The critical path in this schedule, however, involves the FPGA reconfigurations, which are constrained to allow only one reconfiguration at a time. If a scheduler were to be produced for the same workload but using the simplified scheduling model, which allows for parallel reconfigurations, the resulting schedule makespan would be shorter.

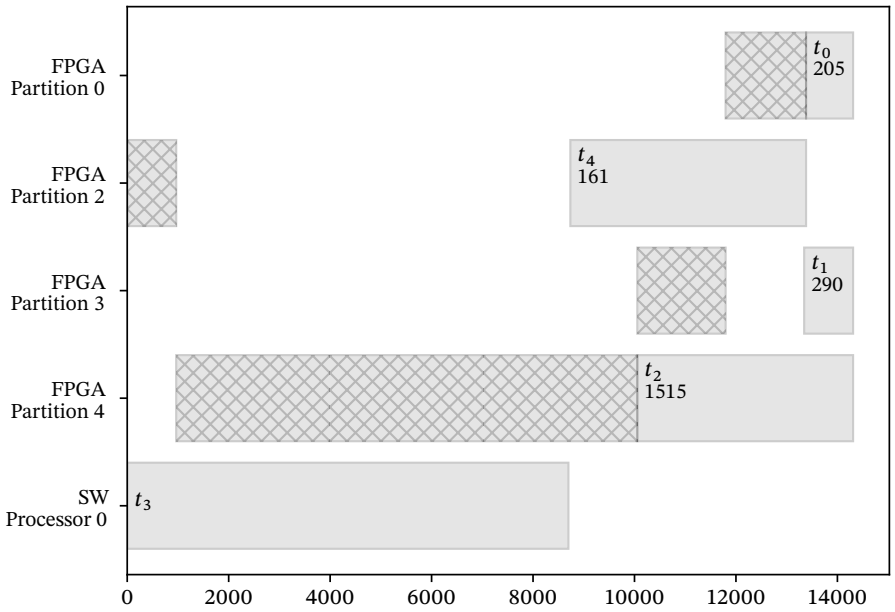


Figure 4.2: Optimal schedule with reconfiguration time per resource unit $s_r = 2$

In figure 4.3, the reconfiguration speed is again improved by a large factor. With the overhead of reconfiguration being very low, all tasks are executed on a single partition using their most performant implementations. The schedule makespan is reduced in compari-

son with figure 4.2 by over a factor of three.

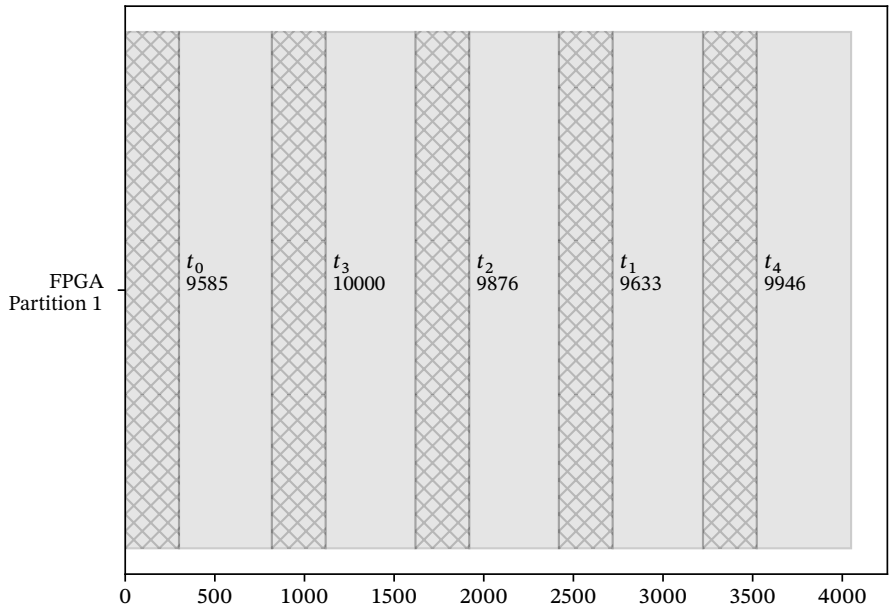


Figure 4.3: Optimal schedule with reconfiguration time per resource unit $s_r = 0.01$

The overall improvement of the makespan with respect to the reconfiguration speed is shown for this same example workload in figure 4.4. In the figure, it can be seen that the rate of improvement in makespan isn't consistent. At some points, the schedule reaches a plateau. At other points, the schedule makespan improves significantly with respect to improvements in reconfiguration speed.

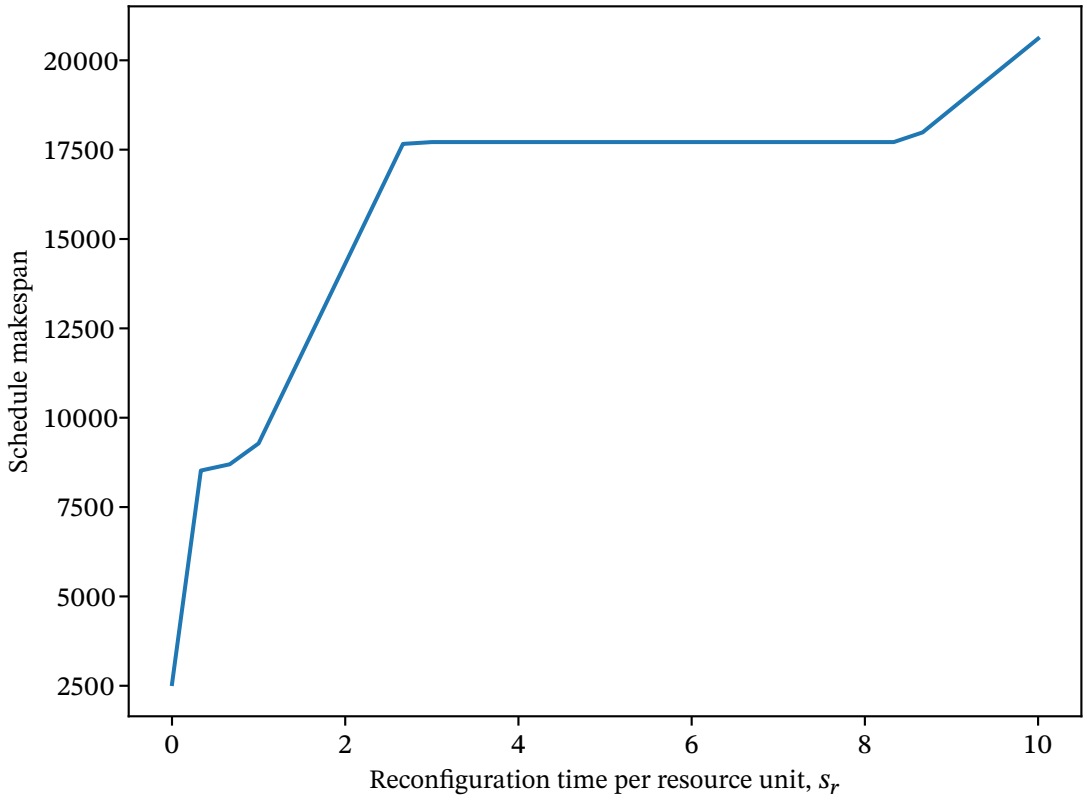


Figure 4.4: Optimal schedule makespan in comparison with reconfiguration speed

4.3 Scheduler Evaluation

Chapter 3 introduced a simplified MILP based scheduler with the motivation being that it would be faster than solving the original MILP formulation to optimality. The simplified MILP based scheduler can then be used to obtain lower bounds schedule makespan while the iterative scheduler may be used to obtain upper bounds. To evaluate the effectiveness of this approach, this section compares results of the original MILP formulation with the iterative scheduler and simplified MILP based scheduler.

This evaluation was performed by running all three schedulers on 10 randomly generated workloads, generated as described in section 3.4, and averaging the results. The implementation was run on a basic laptop computer. To be precise, it was run on Linux with an Intel i5-2520M CPU and 16GB RAM. The iterative scheduler can be made to produce more optimal schedules at the cost of more computation time by adjusting a parameter k, the number of tasks which are added to the scheduler with each iteration. The implementation used

throughout this chapter has a value of $k = 2$.

Table 4.1 lists the average time needed to produce a schedule. To be clear, this isn't the makespan of the schedule, but rather the time needed for the algorithm to generate the schedule. The amount of time needed to solve the full MILP formulation to optimality increases quite quickly. The iterative scheduler and the simplified MILP formulation produce results within a short amount of time, however. Among the samples with 6 tasks, the full MILP formulation completed within 471s for the first sample but took 9 hours to reach a 10% optimality gap for the second sample. Due to the long runtime, the process was terminated. The hardware used in generating table 4.1 is not particularly powerful, which contributes to the longer solve times, and one cutting from the full MILP formulation wasn't used. Nevertheless, the long solve times are generally consistent with what was reported in the paper of Deiana et. al.

Number of tasks	Full MILP formulation	Simplified MILP formulation	Iterative scheduler
1	0.39	0.39	0.09
2	0.48	0.39	0.20
3	1.24	0.40	0.55
4	5.49	0.41	0.75
5	56.17	0.43	1.72
6	N/A	0.43	2.28

Table 4.1: Average time needed to produce a schedule, measured in seconds

For investigating the research question of this thesis, it's important that the iterative scheduler and simplified MILP formulation produce schedules which are relatively close to optimal. To evaluate the accuracy of these two schedulers it's possible to use the mean absolute percentage error (MAPE), given in equation 4.1. In this equation A_w corresponds to the optimal schedule makespan for workload w, as determined by the full MILP formulation, and F_w corresponds to the makespan as determined by the iterative scheduler or simplified MILP formulation. The goal of applying MAPE is simply to evaluate how close the two schedulers are to being optimal. It should be stressed that MAPE should not be used to try and make a direct comparison between the two schedulers and find which is "better". Not only are they solving different problems, one computing an upper bound and the other a lower bound, but also MAPE is biased to report better results for the lower bound produced by the simplified MILP formulation [23].

$$MAPE = \frac{100}{n} \sum_{w=0}^{n} \left| \frac{A_w - F_w}{A_w} \right| \tag{4.1}$$

The resulting MAPE values for the two schedulers on the same workloads are included in table 4.2. As can be seen in figure 4.5, the difference in makespan between the simplified MILP formulation and iterative scheduler depends on the reconfiguration speed. The results given in table 4.2 are based on a reconfiguration time per unit of FPGA resources of $s_r = 2$.

Unfortunately, the inaccuracy of both the simplified MILP formulation and the iterative scheduler appears to increase with the number of tasks. The two schedulers also diverge significantly from the actual optimal schedule. This will limit the usefulness of the approach. The iterative scheduler can be made more accurate at the cost of computation time by adjusting its parameter k, but such a trade-off isn't possible with the simplified MILP formulation.

Number of tasks	Simplified MILP formulation	Iterative scheduler
1	0.0%	0.0%
2	1.5%	0.0%
3	7.0%	7.7%
4	16.5%	5.1%
5	16.9%	18.1%

Table 4.2: MAPE values for the iterative scheduler and simplified MILP formulation

4.4 Overall Impact of Reconfiguration Speed

With the iterative scheduler and simplified MILP formulation implemented, it's possible to run the schedulers on synthetic workloads to gain some insight into the effects of improving reconfiguration speed. To begin, 10 workloads each consisting of 8 tasks were randomly generated using the process described in section 3.4. The schedules were then produced using the iterative scheduler and simplified MILP formulation: the iterative scheduler giving an upper bound on the actual optimal schedule makespan and the simplified MILP formulation giving a lower bound.

Figure 4.5 gives the average schedule makespan with respect to the reconfiguration time per resource unit. The results in this figure appear consistent with what was obtained for the

single example workload in figure 4.4. The makespan of the schedule appears to improve significantly as the reconfiguration overhead approaches zero.

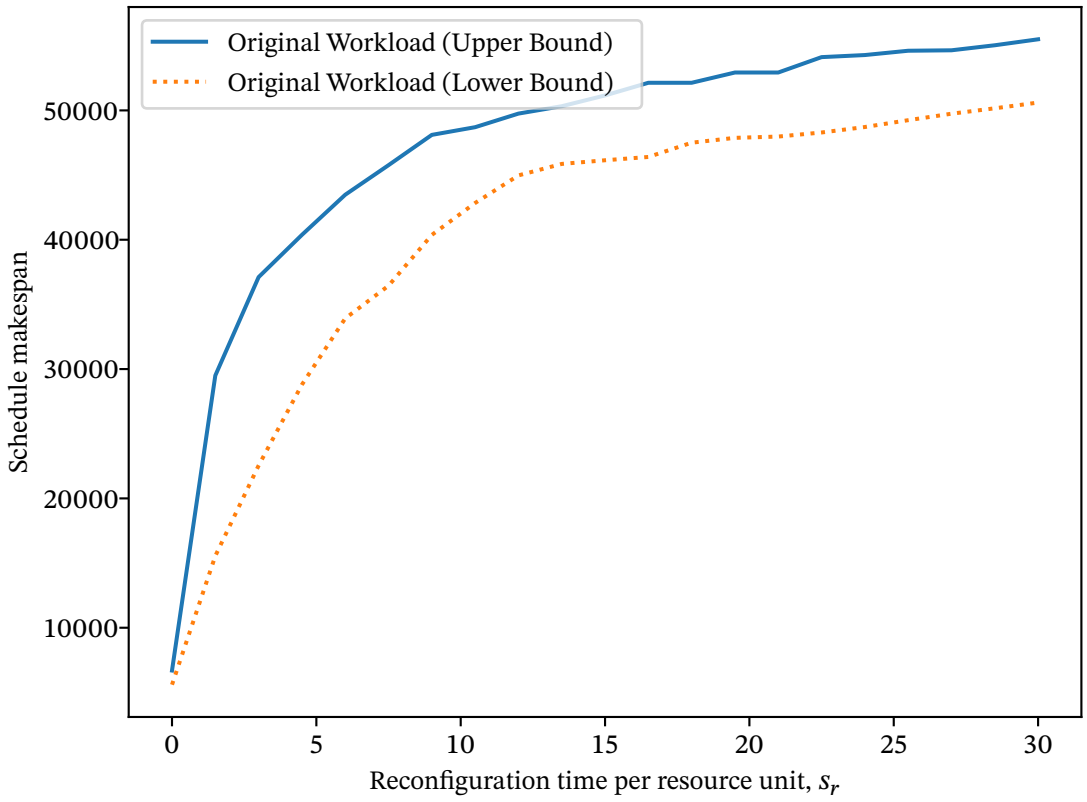


Figure 4.5: Average makespan with respect to reconfiguration time per resource unit

In viewing the example schedules in section 4.2, the tasks transitioned from being scheduled on the CPU to being scheduled on the FPGA as the reconfiguration speed improves. The example schedules also showed that the tasks tended to be scheduled with larger, more performant FPGA implementations as the reconfiguration speed improved. Figures 4.6, 4.7, and 4.8 are useful in checking if this pattern holds more generally. These figures are based on the results of the iterative scheduler and use the same 10 workloads used for figure 4.5.

Figure 4.6 shows the average number of software implementations used as s_r varies. The figure shows that number of software implementations used decreases as the reconfiguration speed improves. With no reconfiguration overhead, $s_r = 0$, very few software implementations are used at all.

Figure 4.7 shows the average FPGA implementation size, the number of LC's, as s_r varies.

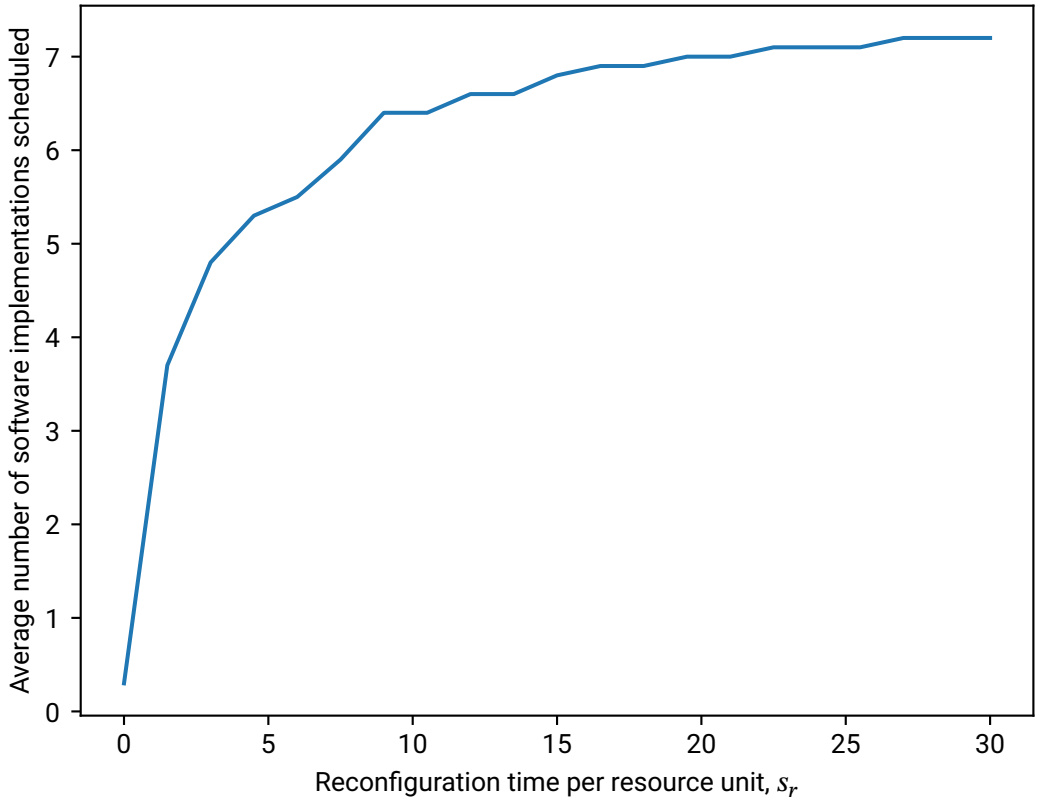


Figure 4.6: Use of software implementations with respect to reconfiguration time per resource unit

To obtain more clear results, the figure is based on a smaller range of s_r: from 0 to 3. Based on figure 4.6, this is the range where FPGA implementations actually begin being used in the schedule. Similar to what was seen in the example schedules in section 4.2, the schedules change to using larger FPGA implementations as the reconfiguration speed improves. This is interesting to note since it could have implications for how applications are developed for a device with extremely low reconfiguration overhead. To best take advantage of such a device when optimizing for performance, it may make sense for application developers to focus on developing larger and more performant FPGA implementations than would typically make sense on devices with slower reconfiguration speeds.

Figure 4.8 shows the average number of FPGA partitions which are used in the schedules. With no reconfiguration overhead, $s_r = 0$, the optimal schedules tend to use fewer FPGA partitions. When considered along with the tendency to use larger FPGA implementations when there is low reconfiguration overhead, it appears that these schedules tend to par-

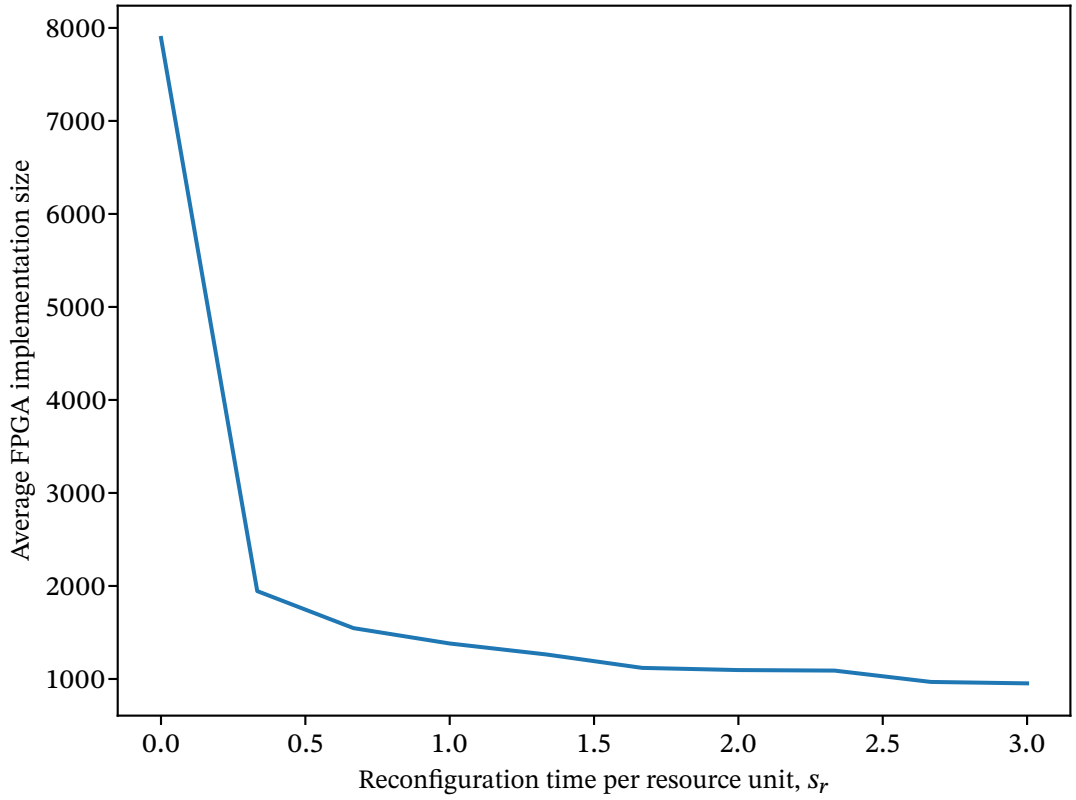


Figure 4.7: Average FPGA implementation size with respect to reconfiguration time per resource unit

tition the FPGA into fewer, larger regions. Assuming that this pattern holds for more realistic workloads, it could have interesting implications for designing an FPGA with fast reconfigurations. When there is almost no reconfiguration overhead, it might not be necessary to support dynamic *partial* reconfiguration, where various portions of the FPGA can be reconfigured at runtime. It might be sufficient to only support dynamic reconfiguration: reconfiguring the entire FPGA at once. Not needing to support partial reconfiguration may make such a device simpler to produce and it might not significantly impact its usefulness.

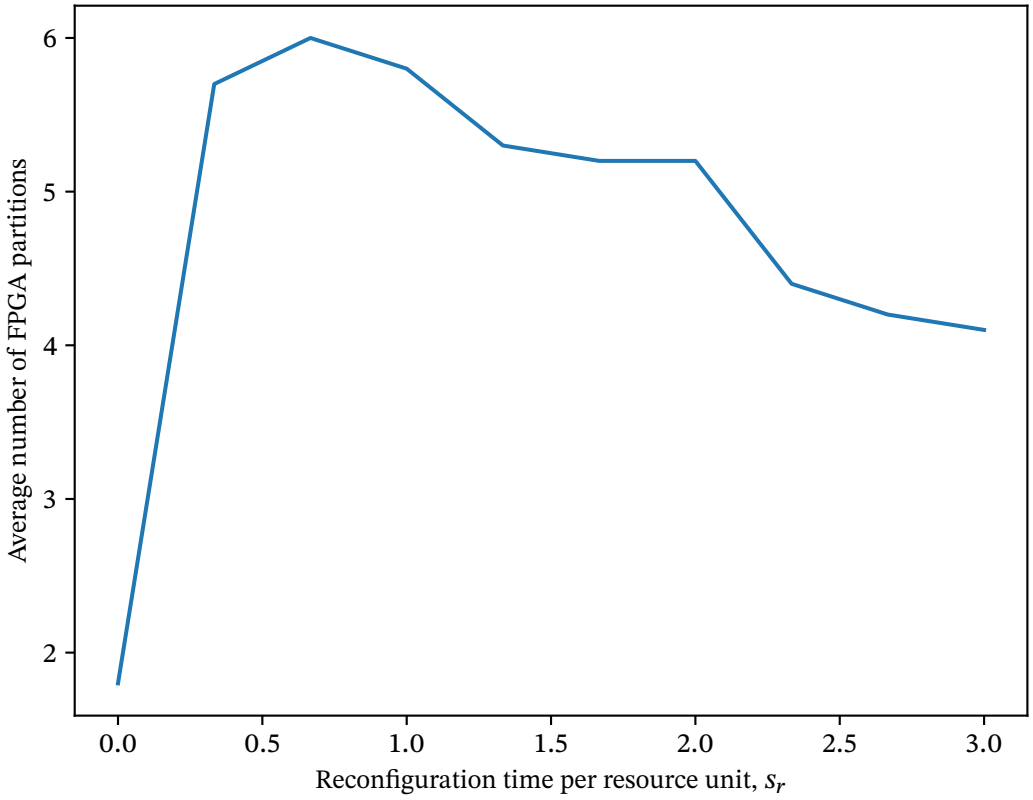


Figure 4.8: Average number of FPGA partitions with respect to reconfiguration time per resource unit

4.5 Effects of Adjusting Resource vs. Performance Trade-Off

As seen in section 2.1.2, the exact FPGA resource vs. performance trade-off available will depend on the application. In the case of cryptographic hash functions, some were able to significantly improve performance by increasing the number of FPGA resources used and some were not. To examine the impact of changing the exact resource vs. performance trade-off, this section applied transformations to the 10 workloads examined in the previous section.

For figure 4.9, the execution times of all FPGA implementations were reduced by 70%. For such a dramatic modification to the FPGA execution times, the figure shows a very limited change. The workloads with improved FPGA implementations clearly perform better at $s_r = 0$, but the difference is less clear at slower reconfiguration speeds.

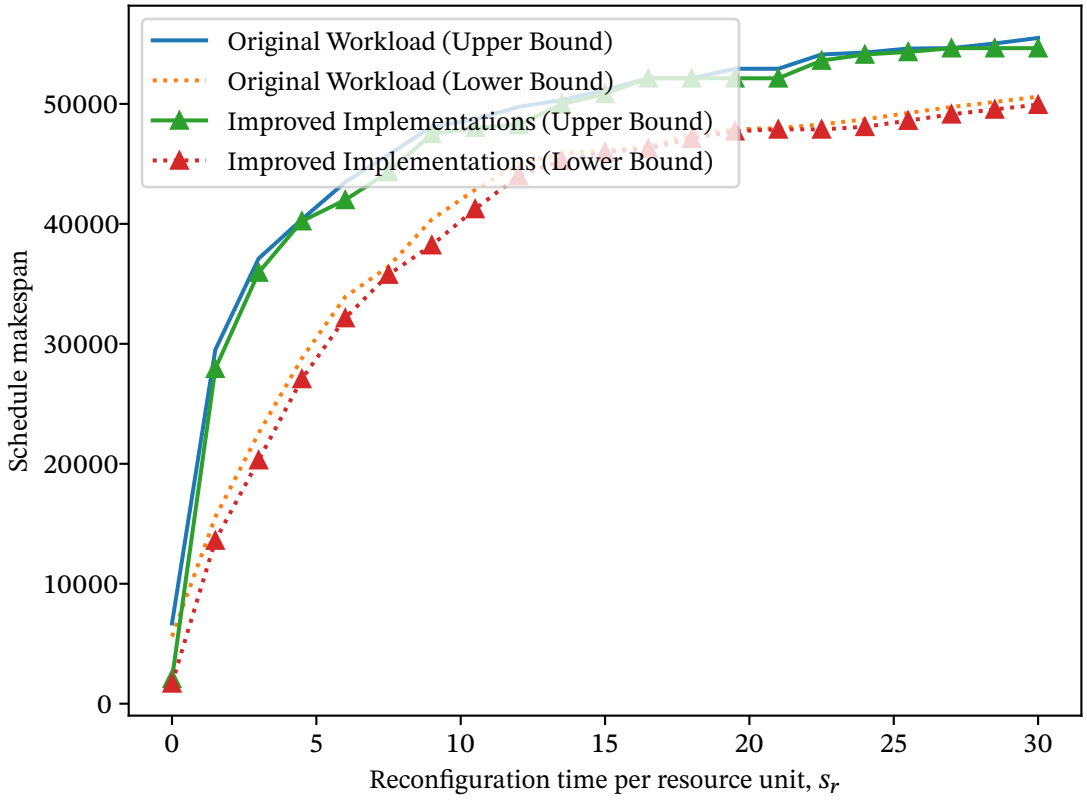


Figure 4.9: Effect of improving FPGA implementation performance on makespan. "Improved Implementations" corresponds to reducing the runtime of FPGA implementations in the original workloads by 70%.

For figure 4.10, the "Shifted Implementations" workloads were obtained by increasing the execution time of all FPGA implementations by a factor of 4, except for the fastest FPGA implementation of each task. The idea behind this is to model applications where larger FPGA implementations are significantly faster than smaller FPGA implementations. Since the original workloads and the "shifted" workloads share the same performant FPGA implementations, they have the same average makespan at $s_r = 0$. This is because when there is no reconfiguration overhead, the fastest FPGA implementations tend to be used. Since the other FPGA implementations are made slower for the "shifted" workloads, the average makespan of these workloads is higher at slower reconfiguration speeds.

Based on these results, it appears that the exact FPGA resource vs. performance trade-off available is an important factor in determining the effect of reconfiguration speed on application performance. In further studies evaluating the usefulness of improving reconfiguration speeds, it may make sense to determine exactly what resource vs. performance

trade-offs are possible for different classes of applications.

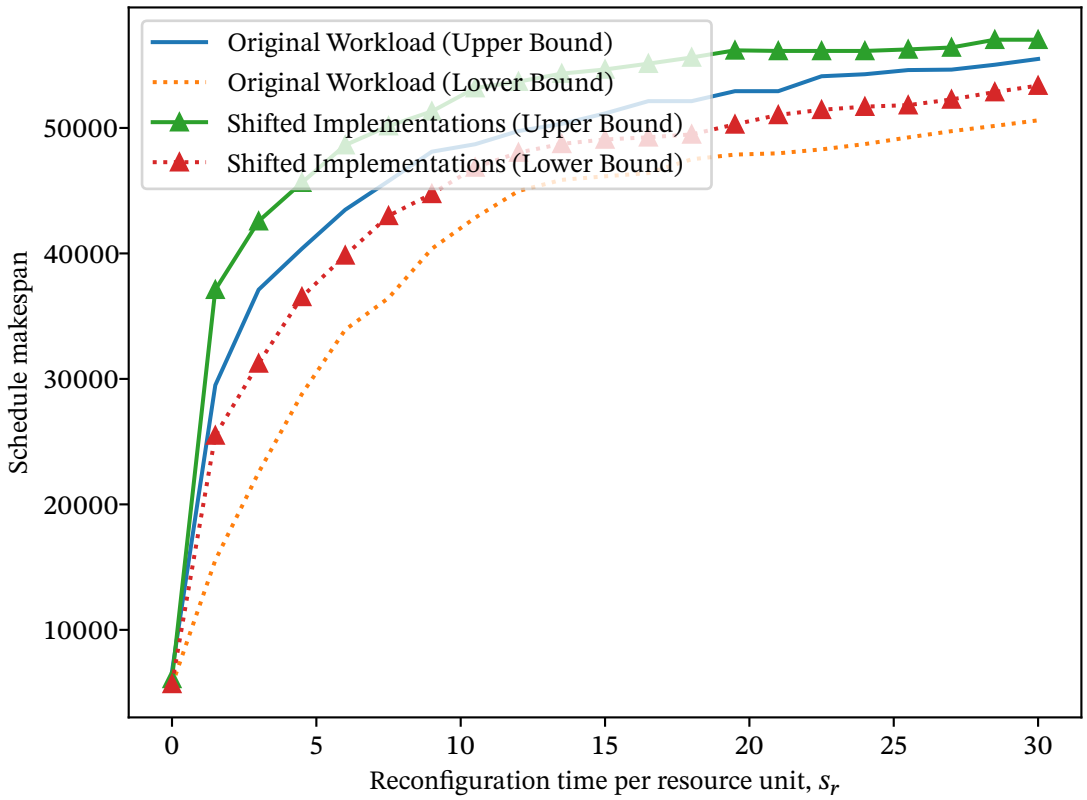


Figure 4.10: Effect of worsening FPGA implementation performance on makespan

4.6 Effects of Restructuring Applications

The previous sections indicate that larger, more performant FPGA implementations tend to be used when reconfiguration speed is fast. It's then interesting to see if applications can be restructured to better take advantage of this tendency. One possibility is to split an application into finer grained tasks.

It's conceivable that some applications have steps which may either be combined into a single task or separated into different tasks. As an example, an application may need to periodically compress and encrypt images that it receives. The compression and encryption phases could either be modeled as a single task comprising both steps, or as two separate tasks. An advantage of combining the two computations into a single task could be that they are able to share some FPGA resources to be more efficient. A potential advantage

of splitting the two computations, on the other hand, is that the resulting tasks could be run in serial by using DPR, with each computation using a larger portion of the total FPGA resources.

A hypothetical example of this using imaginary values is given in table 4.3. For this hypothetical example, we may assume that an FPGA has 1000 logic cells and that both encryption and compression allow for significant performance optimizations. Because the FPGA does not contain enough resources, it's necessary to split them into separate tasks before applying the possible performance optimizations. The separate encryption and compression tasks require in total 0.6ms, compared to 1ms for the combined encryption and compression task.

	Encryption and Compression (Combined)	Encryption	Compression
Resources	1,000 LC	1,000 LC	1,000 LC
Execution Time	1ms	0.3ms	0.3ms

Table 4.3: Example of task splitting

Whether or not splitting a task may result in better performance depends on a few factors. First, there should be a low amount of communication between the different steps and it should be possible to run them one after the other. If the steps stream a significant amount of data between each other, it may be infeasible to store this in memory to run the tasks one after the other. Secondly, the computational steps should have a good performance vs. resource trade-off. If the finer grained tasks are only slightly more performant when using additional FPGA resources, it's unlikely that splitting the task will offer much improvement. With the tasks split, there will also be an additional reconfiguration which is required. Whether or not this optimization improves performance therefore depends on the reconfiguration speed.

Figure 4.11 shows the effects of applying this task splitting optimization to randomly generated workloads. In particular, 10 workloads each containing 4 tasks were generated. These workloads correspond to the "original workload" series in the figure. Based on these workloads, new workloads were generated by splitting each task t_i into two tasks: t_i^a and t_i^b. The software implementations of t_i^a and t_i^b both were given half the execution time of the corresponding software implementation of t_i. The hardware implementations of t_i^a and t_i^b were given the same execution time as the corresponding implementation of t_i, but each required 60% of the resources of t_i. Since the resource requirements of the split tasks is greater than

50%, this accounts for some resource overhead in splitting the tasks. In addition, the split tasks t_i^a and t_i^b were each given an additional hardware implementation. This hardware implementation requires approximately the same amount of resources as the largest hardware implementation of t_i, but requires only one third of the execution time. Adding these additional implementations results in a similar situation to that depicted in table 4.3

In figure 4.11, task splitting appears to result in a longer makespan at slower reconfiguration speeds. This is likely due to the split FPGA implementations being larger than the combined FPGA implementation, resulting in a longer time needed for reconfiguration. At $s_r = 0$, however, task splitting does result in a slightly lower makespan.

The improvement from task splitting shown in 4.11 is quite modest compared to the overall improvement in makespan from improving reconfiguration speeds. This restructuring of applications may result in even larger improvements, though, if tasks are split into even finer grain tasks or if the possible performance vs. resource trade-off is better than what was used in this model. One factor that may make this optimization unrealistic, though, is that it assumes there is little communication between the split tasks and that the split tasks may be run in serial. Many efficient FPGA implementations may stream large amounts of data from one component to another. To split such applications into finer grained tasks and run them in serial may require more data to be transferred to and from memory, adding a significant overhead which outweighs any potential benefits.

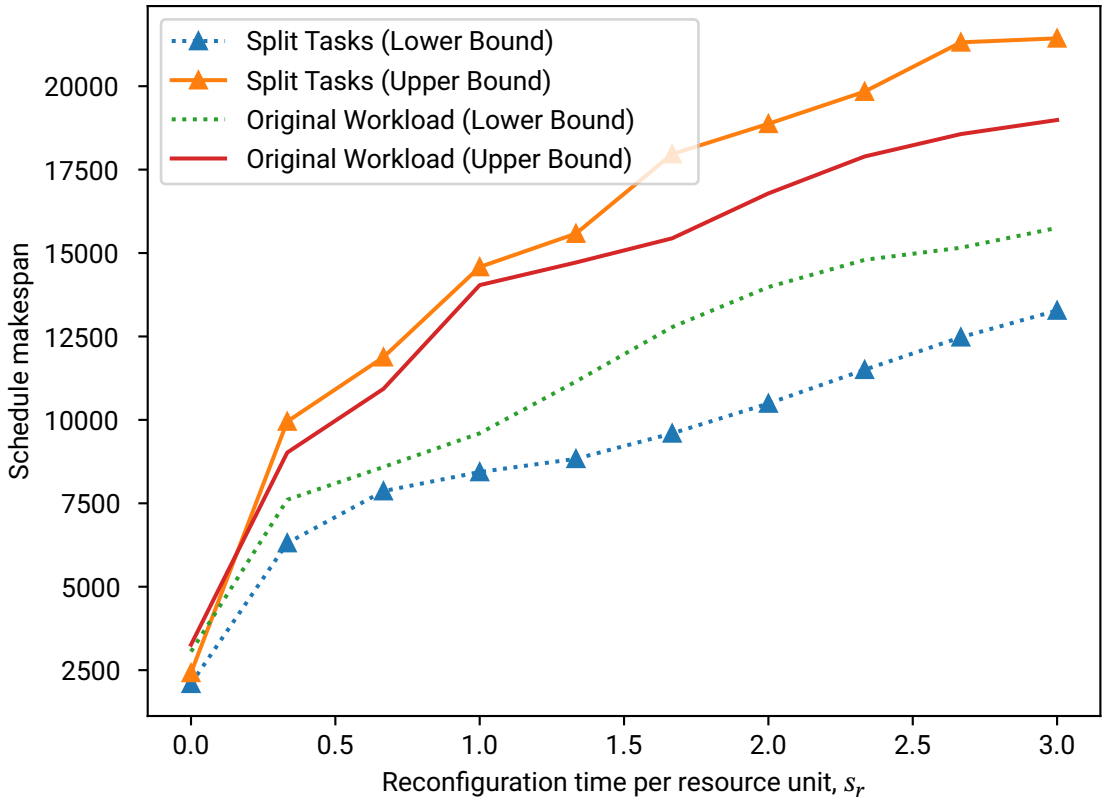


Figure 4.11: Average effect of splitting tasks on schedule makespan

5 TOWARDS A LAGRANGIAN BASED SCHEDULER

5.1 Introduction

A limitation of the analysis in chapter 4 is that the gap between upper and lower bounds on the schedule makespan was quite large. It would be interesting to obtain a smaller gap between the upper and lower bounds on schedule makespan and also to make the iterative scheduler require less time to find a solution. This would allow for obtaining more clear results on the impact of reconfiguration times than what was obtained in the previous chapter. A more performant scheduler would also be useful for scheduling existing real-world applications which use DPR.

An approach which is commonly used in optimization to accomplish these objectives is the Lagrangian relaxation. Section 3.3.3 already gave the definition of a relaxation and showed that the simplified scheduling problem was a relaxation of the MILP formulation from Deiana et. al. Rather than simply ignoring constraints of the original problem, as was done for obtaining the simplified scheduling problem, the Lagrangian relaxation involves dualizing some constraints: removing them as actual constraints and incorporating them in the objective function.

A more thorough discussion of the Lagrangian can be found in [28], but some definitions will be included here. In particular, consider the following integer program *IP*:

$$\begin{aligned} z = \min \quad & cx \\ \text{s.t.} \quad & Dx \leq d, \\ & x \in X \end{aligned} \tag{5.1}$$

For any value of $u = (u_1, \ldots, u_m) \geq 0$, we may then define the problem $IP(u)$ as shown in equation 5.2. This is done by removing the constraint $Dx \leq d$ and incorporating it into the objective function.

$$\begin{aligned} z(u) = \min \quad & cx - u(d - Dx) \\ \text{s.t.} \quad & x \in X \end{aligned} \tag{5.2}$$

Problem $IP(u)$ is known as a *Lagrangian relaxation* of IP and it can be shown that $IP(u)$ is a relaxation of IP meeting the definition given much earlier in definition 1. In particular, the domain of $IP(u)$ is a super-set of the domain of IP, since we simply remove the constraint $Dx \leq d$. Also, for all $x \in X$ satisfying $Dx \leq d$, we have that the objective function of $IP(u)$ is less than that of IP. As a relaxation of IP, $IP(u)$ provides a dual bound, in this case a lower bound, for any value of u. The vector u is known as the *Lagrangian multipliers*. To find the best possible dual bound, it's necessary to then solve the *Lagrangian Dual Problem*:

$$w_{LD} = \max_u z(u) \tag{5.3}$$

Not only does the Lagrangian method provide dual bounds, but it may also be used in algorithms for solving the original problem. One possibility is to develop heuristic algorithms based on the Lagrangian relaxation [28]. The solutions to the Lagrangian relaxation $IP(u)$ may become quite close to being primal feasible solutions. In this case, it can be possible to devise a heuristic algorithm which converts a solution of the Lagrangian relaxation to a solution of the original problem. The bounds from the Lagrangian relaxation may also be used in developing a branch and bound algorithm for solving the original problem [8].

This chapter presents an attempt at applying the Lagrangian relaxation to the FPGA scheduling problem, but falls short of obtaining a strong lower bound. This chapter will outline the approach that was taken, clarify why it may have yielded poor results, and provide ideas for how the Lagrangian relaxation could be better applied.

5.2 Connection with $R||C_{max}$

The FPGA scheduling problem is related to a classic scheduling problem $R||C_{\max}$. This is the problem of minimizing a schedule makespan on unrelated parallel machines [18].

More formally, let there be n jobs $j = 1, \dots, n$ and m machines $i = 1, \dots, m$. Also let the processing time required by job j on machine i be given by the parameter e_{ij}. The scheduling problem $R||C_{\max}$ then consists of assigning each job to a machine, such that the makespan for processing all jobs is minimized.

Martello et. al give a straightforward ILP formulation of this problem by introducing binary variables x_{ij} [18]. x_{ij} is 1 if job i is to be processed on machine m and is 0 otherwise. The formulation also introduces a variable z which gives the makespan of the schedule. The ILP formulation of $R||C_{\max}$ can then be defined as:

$$\begin{aligned} \min \quad & z \\ \text{s.t.} \quad & \sum_{i=1}^{m} x_{ij} = 1 && (j = 1, \dots, n), \\ & \sum_{j=1}^{n} e_{ij} x_{ij} \leq z && (i = 1, \dots, m), \\ & x_{ij} \in \{0, 1\} \end{aligned} \tag{5.4}$$

The simplified scheduling problem reduces to an $R||C_{\max}$ problem when FPGA partition sizes $u_{r,p}$ are fixed. The FPGA partitions and CPU cores correspond to machines, and each task must still be scheduled exactly once with the objective of minimizing makespan. One difference is that the simplified scheduling problem allows for the scheduler to select from multiple implementations and contains FPGA resource constraints. With the FPGA partition sizes $u_{r,p}$ fixed, however, a task scheduled to an FPGA partition will simply use the fastest possible implementation which fits.

In particular, for all tasks t and all FPGA partitions $p \in P_{HW}$ we may define the execution time $e_{t,p}$ as the shortest amount of time needed to reconfigure and execute task t using an implementation which fits on FPGA partition p. For all tasks t and all CPU cores $p \in P_{SW}$, we may similarly define the execution time $e_{t,p}$ as the shortest time needed to execute task t on component p.

An interesting result regarding $R||C_{\max}$ from Martello, Soumis, and Toth applies the La-

grangian relaxation to the problem [18]. Given the similarity between $R||C_{\max}$ and FPGA scheduling, it appears plausible to try and reuse their same techniques in solving the FPGA scheduling problem.

5.3 Lagrangian Relaxation of the Simplified Scheduling Problem

A natural choice would be to try and apply the Lagrangian relaxation to the full model of the FPGA scheduling problem from Deiana et. al. This could potentially improve the performance of the scheduler and be used to obtain stronger lower bounds than the bounds provided by the simplified scheduling problem. This proved to be very complicated, however. In order to model all of the details of FPGA reconfigurations, the full scheduling model contains over twenty different types of constraints. Instead, this chapter will propose an application of the Lagrangian relaxation to the simplified scheduling problem introduced in section 3.3. This simplified scheduling problem is much simpler, and applying the Lagrangian relaxation will be more straightforward as a result.

The decision to apply the Lagrangian relaxation to the simplified FPGA scheduling problem is questionable, though. The simplified scheduling problem is already a relaxation of the model of Deiana et. al, so relaxing the simplified scheduling problem further isn't so beneficial. The simplified scheduling problem is already solvable in a reasonable amount of time, so it isn't so valuable to improve its performance with algorithms based on the Lagrangian relaxation. Nevertheless, one potential benefit of doing this is to gain some insight into how the Lagrangian relaxation could be applied to the full scheduling problem.

To make the presentation more clear, the MILP formulation of the simplified scheduling problem originally shown in equation3.2 is copied here:

$$\text{minimize} \quad z \tag{5.5a}$$

subject to

$$\sum_{(t,p,i)\in TPI|t=t'} x_{t,p,i} = 1 \qquad \forall t' \in T, \tag{5.5b}$$

$$\sum_{(t,p,i)\in TPI|p=p'} (l_i + \sum_{r\in R} s_r \cdot oc_{i,r}) x_{t,p,i} \leq z \qquad \forall p' \in P_{HW}, \tag{5.5c}$$

$$\sum_{(t,p,i)\in TPI|p=p'} l_i \cdot x_{tpi} \leq z \qquad \forall p' \in P_{SW}, \tag{5.5d}$$

$$\sum_{(t,p,i)\in TPI|p=p',t=t'} oc_{i,r} \cdot x_{t,p,i} \leq u_{r,p} \qquad \forall p' \in P_{HW}, t' \in T, r' \in R, \tag{5.5e}$$

$$\sum_{p\in P_{HW}} u_{r,p} \leq maxRes_r \qquad \forall r' \in R, \tag{5.5f}$$

$$x_{t,p,i} \in \{0,1\} \qquad \forall (t,p,i) \in TPI \tag{5.5g}$$

For the Lagrangian relaxation, it's necessary to select which constraints to dualize. This selection has implications for how challenging the Lagrangian relaxation is to solve and how tight the resulting bound is. As will be shown later on, dualizing constraints 5.5b and 5.5e allows for a relatively performant algorithm for solving the Lagrangian relaxation. By removing these particular constraints and incorporating them in the objective function, valid schedules no longer need to run tasks exactly once or respect the resource limitations of each component.

While the relaxation allows for tasks to be scheduled multiple times, it may make sense to limit this. Adding the cutting planes given in equation 5.6 restricts each task to being scheduled once per component. This prevents the same task from being scheduled multiple times on a given component using different implementations. Adding this cutting plane doesn't significantly complicate solving the Lagrangian relaxation.

$$\sum_{(t,p,i)\in TPI|t=t',p=p'} x_{t,p,i} \leq 1 \qquad \forall t' \in T, p' \in P \tag{5.6}$$

Additionally, it helps to add cutting planes restricting the possible values of z. It will be shown later on that this simplifies the algorithm for solving the Lagrangian relaxation. Given a lower bound z_L and upper bound z_U on the optimal schedule makespan from the

primal problem, the MILP formulation 5.5, constraints $z_L \leq z$ and $z \leq z_U$ are added to the Lagrangian relaxation.

In dualizing constraints 5.5b, it's necessary to introduce Lagrangian multipliers λ_t for all tasks $t \in T$. Similarly, dualizing constraints 5.5e introduces Lagrangian multipliers $\pi_{p,t,r}$ for all combinations of FPGA components $p \in P_{HW}$, tasks $t \in T$, and resource types $r \in R$. Let λ be a vector of the multipliers λ_t and π a vector of the multipliers $\pi_{p,t,r}$. Since constraint 5.5b is an equality, the multipliers λ_t may be positive or negative. The multipliers $\pi_{p,t,r}$ must be positive. The resulting Lagrangian relaxation is then subject to constraints 5.5c, 5.5d, 5.5f, 5.5g, and 5.6 and optimizes the following objective function:

$$\begin{aligned} L(\lambda,\pi) = \min \Bigg[& z - \sum_{t' \in T} \lambda_{t'} \Big(\sum_{(t,p,i) \in TPI | t' = t} x_{t,p,i} - 1 \Big) \\ & - \sum_{p' \in P_{HW}} \sum_{t' \in T} \sum_{r \in R} \pi_{p',t',r} \Big(u_{r,p'} - \sum_{(t,p,i) \in TPI | t' = t, p' = p} oc_{i,r} \cdot x_{t,p,i} \Big) \Bigg] \end{aligned} \tag{5.7}$$

This objective function is obtained by dualizing constraints 5.5b and 5.5e with the standard application of the Lagrangian relaxation. In this form, though, it's not immediately clear how to solve the Lagrangian relaxation problem. To make the problem more clear, it helps to group variables so that they each appear in exactly one term of the objective function. The objective function can then be rewritten into the following form:

$$\begin{aligned} L(\lambda,\pi) = \min \Bigg[& z + \sum_{t \in T} \lambda_t \\ & - \sum_{p' \in P_{SW}} \sum_{(t,p,i) \in TPI | p' = p} \lambda_t x_{t,p,i} \\ & - \sum_{p' \in P_{HW}} \sum_{(t,p,i) \in TPI | p' = p} (\lambda_t - \sum_{r \in R} \pi_{p,t,r} \cdot oc_{i,r}) x_{t,p,i} \\ & - \sum_{r \in R} \sum_{p \in P_{HW}} (u_{r,p} \cdot \sum_{t \in T} \pi_{p,t,r}) \Bigg] \end{aligned} \tag{5.8}$$

This Lagrangian relaxation problem is similar to the one obtained by Martello et. al in their work on $R||C_{\max}$. As they observed for $R||C_{\max}$, when the variable z is fixed to any positive integer value, this problem can be decomposed into independent 0-1 knapsack problems. In particular, for all $p' \in P_{SW}$ the sub-problem $LL^{SW}_{p'}(\lambda, z)$ may be defined as:

$$
\begin{aligned}
LL^{SW}_{p'}(\lambda, z) = \max \quad & \sum_{(t,p,i)\in TPI|p=p'} \lambda_t x_{t,p,i} \\
\text{s.t.} \quad & \sum_{(t,p,i)\in TPI|p=p'} l_i \cdot x_{t,p,i} \leq z, \\
& \sum_{(t,p,i)\in TPI|p=p',t=t'} x_{t,p,i} \leq 1 \qquad \forall t \in T, \\
& x_{t,p,i} \in \{0,1\} \qquad \forall (t,p,i) \in TPI
\end{aligned}
\tag{5.9}
$$

Similarly, for all $p' \in P_{HW}$ the sub-problem $LL^{HW}_{p'}(\lambda, \pi, z)$ may be defined as:

$$
\begin{aligned}
LL^{HW}_{p'}(\lambda, \pi, z) = \max \quad & \sum_{(t,p,i)\in TPI|p=p'} (\lambda_t - \sum_{r\in R} \pi_{p,t,r} oc_{i,r}) \cdot x_{t,p,i} \\
\text{s.t.} \quad & \sum_{(t,p,i)\in TPI|p=p'} (l_i + \sum_{r\in R} s_r \cdot oc_{i,r}) \cdot x_{t,p,i} \leq z, \\
& \sum_{(t,p,i)\in TPI|p=p',t=t'} x_{t,p,i} \leq 1 \qquad \forall t \in T, \\
& x_{t,p,i} \in \{0,1\} \qquad \forall (t,p,i) \in TPI
\end{aligned}
\tag{5.10}
$$

Sub-problems LL^{HW}_p and LL^{SW}_p can be thought of as a scheduling problem for a single FPGA reconfigurable region or software processor, respectively. Since the constraint 5.5b from the original MILP formulation was relaxed, the same task may be scheduled multiple times across the different sub-problems. Since z is fixed for these sub-problems, they are essentially instances of knapsack problems. In the case of LL^{SW}_p, the parameter z corresponds to the size of the knapsack, λ_t corresponds to the value of items, and l_i corresponds to the weight of items. The cutting planes which limit one implementation being selected per task make this problem a variant of the knapsack problem, rather than corresponding exactly to the knapsack problem. It's worth pointing out that LL^{HW}_p and LL^{SW}_p are maximization problems whereas the Lagrangian relaxation is a minimization problem. These sub-problems contribute negatively to the overall objective function of the Lagrangian relaxation, so that is why they are maximizing.

The final sub-problem, $LL^{Resource}_r(\pi)$, may be defined for all resources $r \in R$ as:

$$\begin{aligned} LL_r^{Resource}(\pi) = \max \quad & \sum_{p \in P_{HW}} u_{r,p} \cdot \sum_{t \in T} \pi_{p,t,r} \\ \text{s.t.} \quad & \sum_{p \in P_{HW}} u_{r,p} \leq maxRes_r \end{aligned} \tag{5.11}$$

This sub-problem may be solved by determining $p_{\max} = \text{argmax}_{p \in P_{HW}} \sum_{t \in T} \pi_{p,t,r}$. The optimal solution is obtained by setting $u_{r,p_{\max}} = maxRes_r$ and setting $u_{r,p} = 0$ for $p \neq p_{\max}$. If there are multiple partitions $p_1, \ldots, p_n$ which maximize the argmax value, they may each be allocated the same amount of resources. In particular, an optimal solution is obtained by setting $u_{r,p_i} = maxRes_r/n$ for the partitions p_i where $i = 1, \ldots, n$ and otherwise $u_{r,p} = 0$. Keeping the partitions symmetrical will make the Lagrangian relaxation easier to solve, as will be explained in section 5.4.

Solving the collection of newly defined sub-problems for a given value of z results in a feasible solution to the Lagrangian relaxation problem. The resulting value of the objective function 5.8 is then given by:

$$L(\lambda, \pi, z) = z + \sum_{t \in T} \lambda_t - \sum_{p \in P_{SW}} LL_p^{SW}(\lambda, z) - \sum_{p \in P_{HW}} LL_p^{HW}(\lambda, \pi, z) - \sum_{r \in R} LL_r^{Resource}(\pi) \tag{5.12}$$

An *optimal* solution to the Lagrangian relaxation problem, as opposed to simply any feasible solution, is needed in order to obtain a valid lower bound, though. When defining the Lagrangian relaxation, the valid values of z were restricted to the range $z_L \leq z \leq z_U$. An optimal solution to the Lagrangian relaxation problem can then be determined by minimizing $L(\lambda, \pi, z)$ with respect to z:

$$L(\lambda, \pi) = \min_{z_L \leq z \leq z_U} L(\lambda, \pi, z) \tag{5.13}$$

Martello et. al arrived at the same form in their application of the Lagrangian relaxation to $R||C_{\max}$. They proposed a general way of improving lower bounds of this form by adding a cutting constraint. While Martello et. al arrived at the following theorem for $R||C_{\max}$, their proof holds for the simplified FPGA scheduling problem as well.

Theorem 1. *Given any lower bound L computed as* $L = \min_{z_L \leq z \leq z_U} L(z)$*, where* $L(z)$ *is a valid lower bound when the optimal solution has value z, then*

$$\overline{L} = \min\{z : z_L \leq z \leq z_U \wedge L(z) \leq z\} \tag{5.14}$$

is a valid lower bound dominating L, i.e., $\overline{L} \geq L$.

Since equation 5.13 meets the criteria required by theorem 1, it's possible to calculate an even stronger lower bound $\overline{L}(\lambda, \pi)$. Not only does theorem 1 result in a better lower bound, but Martello et. al observe that the value of $\overline{L}$ can be determined through binary search over z. The only requirement is that $L(\lambda, \pi, z) - z$ is monotonic step-wise decreasing, which is shown in lemma 1. Being able to apply binary search is particularly useful since each calculation of $L(\lambda, \pi, z)$ requires solving knapsack problems and may be computationally expensive. Determining $L(\lambda, \pi)$ by evaluating $L(\lambda, \pi, z)$ for all z from z_L to z_U would be less efficient.

Because we perform binary search over z, it's useful to assume that z is an integer. For simplicity, this analysis will simply assume that all parameters and variables of the scheduling problem are integer. It may be possible to loosen this requirement with further work.

Lemma 1. *$L(\lambda, \pi, z) - z$ is monotonic step-wise decreasing.*

Proof. From equation 5.12 we have that:

$$\begin{aligned} L(\lambda, \pi, z+1) =& (z+1) + \sum_{t \in T} \lambda_t - \sum_{p \in P_{SW}} LL_p^{SW}(\lambda, z+1) - \\ & \sum_{p \in P_{HW}} LL_p^{HW}(\lambda, \pi, z+1) - \sum_{r \in R} LL_r^{Resource}(\pi) \end{aligned} \tag{5.15}$$

Given that sub-problems LL_p^{SW} and LL_p^{HW} are knapsack problems, reducing the size of the knapsack will reduce their value (or leave it unchanged). This gives the following inequalities:

$$\begin{aligned} LL_p^{SW}(\lambda, z) &\leq LL_p^{SW}(\lambda, z+1) \\ LL_p^{HW}(\lambda, \pi, z) &\leq LL_p^{HW}(\lambda, \pi, z+1) \end{aligned} \tag{5.16}$$

Taken together, equations 5.15 and 5.16 imply that:

$$\begin{aligned} L(\lambda, \pi, z+1) \leq (z+1) + \sum_{t \in T} \lambda_t - \sum_{p \in P_{SW}} LL_p^{SW}(\lambda, z) - \\ \sum_{p \in P_{HW}} LL_p^{HW}(\lambda, \pi, z) - \sum_{r \in R} LL_r^{Resource}(\pi) \\ = 1 + L(\lambda, \pi, z) \end{aligned} \tag{5.17}$$

Rearranging this inequality then shows that $L(\lambda, \pi, z+1) - (z+1) \leq L(\lambda, \pi, z) - z$. □

In order to apply this approach, it's necessary to obtain initial upper and lower bounds on the schedule makespan. As a lower bound, a weak bound may be obtained by finding for each task the shortest amount of time needed to execute it. The longest such time is then a valid lower bound. This lower bound is given in equation 5.18. The function $time(i)$ is the time needed to execute the implementation, with $time(i) = l_i$ for software implementations and $time(i) = l_i + \sum_{r \in R} s_r \cdot oc_{i,r}$ for FPGA implementations. I_t is the set of implementations compatible with task t.

$$z_L = \max_{t \in T} \min_{i \in I_t} time(i) \tag{5.18}$$

For the upper bound, it's possible to assume that every task runs in serial using the slowest implementations available. This is given by the equation 5.19.

$$z_U = \sum_{t \in T} \max_{i \in I_t} time(i) \tag{5.19}$$

5.4 Simplification for Solving the Lagrangian Relaxation

As described in the previous section, the solution to the Lagrangian relaxation $L(\lambda, \pi)$ can be found by a binary search over values of z. With each value of z, it will be necessary to solve the sub-problems LL_p^{SW} and LL_p^{HW} for all components $p \in P$ and additionally $LL_r^{Resource}$ for all $r \in R$. As LL_p^{SW} and LL_p^{HW} are problems are knapsack problems, though, they may still be computationally challenging to solve. Solving the Lagrangian relaxation may therefore become especially complex.

Still, it's possible to make a simplification to the problem which wasn't possible for Martello

et. al's work on $R||C_{\max}$. Unlike in $R||C_{\max}$, the different FPGA components are all homogeneous. Similarly, the CPU components are all homogeneous. By selecting the Lagrangian multipliers in a symmetric way, the sub-problems LL_p^{HW} and LL_p^{SW} will be equivalent for all $p \in P_{HW}$ and $p \in P_{SW}$ respectively.

As a reminder, it's possible to select any values for the Lagrangian multipliers and still obtain a valid dual bound. By assigning $\pi_{p_i,t,r} = \pi_{p_j,t,r}$ for all $p_i, p_j \in P_{HW}$, the resulting hardware sub-problems are all identical. Rather than needing solve the sub-problems LL_p^{HW} for all FPGA components $p \in P_{HW}$, it's possible to solve once and reuse the same solution. For any fixed value for the Lagrangian multipliers, the software sub-problems LL_p^{SW} are all equivalent as well. It's therefore possible to similarly only solve one instance of LL_p^{SW}. This significantly reduces the time needed to solve the Lagrangian Relaxation $L(\lambda, \pi)$.

While setting the Lagrangian multipliers in such a way simplifies the computation of $L(\lambda, \pi)$, it's not immediately clear whether such multipliers can result in a good lower bound. Fortunately, the highest possible lower bound is obtainable with π which are symmetric over the hardware partitions. This is proved in the following Theorem.

Theorem 2. *Suppose that the Lagrangian dual problem is bounded: there exists some particular Lagrangian multipliers which maximize the Lagrangian relaxation. Then there exist multipliers λ^*, π^* such that:*

1. *The π^* multipliers are symmetric over p: $\pi^*_{p_i,t,r} = \pi^*_{p_j,t,r}$ for all $p_i, p_j \in P_{HW}$, $t \in T$, $r \in R$.*

2. *The multipliers λ^* and π^* result in an optimal bound. For all other λ and π, we have that $L(\lambda^*, \pi^*) \geq L(\lambda, \pi)$.*

Proof. The proof follows from the fact that $L(\lambda, \pi)$ is subdifferantiable everywhere and that the FPGA partitions are essentially identical with each other.

Suppose that there exist optimal multipliers λ' and π' such that $\pi'_{p_i,t,r} \neq \pi'_{p_j,t,r}$ for some $p_i, p_j \in P_{HW}$, $t \in T$, $r \in R$. Consider the set of permutations of permutations of P_{HW} given by $Sym(P_{HW}) = \{\alpha : P_{HW} \mapsto P_{HW} \mid \alpha \text{ is a permutation of } P_{HW}\}$. For each permutation $\alpha_i \in Sym(G)$, let A_i be a corresponding function on the Lagrangian multipliers π which sets $\mathrm{A}_i(\pi_{p,t,r}) = \pi_{\alpha_i(p),t,r}$. Since A is essentially just renaming the FPGA partitions, the value of the Lagrangian relaxation is unchanged. In particular this applies to the optimal multipliers λ' and π': $L(\lambda', \mathrm{A}_i(\pi')) = L(\lambda', \pi')$ for all permutations A_i.

Based on A_i, it's possible to define a new multiplier π^* by averaging the different permutations of π'. This is given in equation 5.20. It can be shown that the resulting π^* is symmetric over $p \in P_{HW}$ as defined in part 1 of the theorem. For defining λ^*, we may simply set $\lambda^* = \lambda'$.

$$\pi^* = \frac{1}{|Sym(P_{HW})|} \cdot \sum_{\alpha_i \in Sym(P_{HW})} A_i(\pi') \tag{5.20}$$

It then remains to show that $L(\lambda^*, \pi^*) = L(\lambda', \pi')$. In general, the Lagrangian relaxation is concave (or convex depending on whether the problem is minimizing or maximizing). Given that λ^*, π^* are obtained from a convex combination of the λ' and $A_i(\pi')$ multipliers, $L(\lambda, \pi)$ is concave, and $L(\lambda', A_i(\pi'))$ are all equal, we have that $L(\lambda^*, \pi^*) \geq L(\lambda', \pi')$. Since $L(\lambda', \pi')$ is optimal, condition 2 of the theorem is satisfied. □

5.5 Results

In order to evaluate this Lagrangian relaxation, an implementation was done in python. For optimizing the Lagrangian dual problem, both the subgradient and bundle methods were tested. The subgradient and bundle methods were not implemented from scratch, and a library nsopy [26] was used instead.

To evaluate the bounds from the Lagrangian relaxation, the bounds were compared against the result of the simplified scheduling problem MILP formulation and the naive lower bound z_L. The Lagrangian relaxation occasionally produced a slightly better lower bound than z_L, but the lower bound was always much less than the result of the simplified scheduling problem. Overall, the resulting bounds from the Lagrangian relaxation showed almost no improvement over z_L.

Given that this application of the Lagrangian relaxation is so similar to that of Martello et. al for $R||C_{\max}$, it's interesting that it produced much worse results. One possible explanation is that the solutions to the relaxation of the simplified FPGA scheduling problem tends to diverge more from actual solutions to the primal problem. Since an identical sub-problem LL_p^{HW} is used for all FPGA components, either a given task will be scheduled $|P_{HW}|$ times or it will not be scheduled at all. In $R||C_{\max}$, the machines are heterogeneous so, depending on the values of the Lagrangian multipliers, it's possible that tasks are scheduled more closely to one time, as required in the primal problem. More investigation would be needed,

though, to determine exactly why the bounds from the Lagrangian relaxation proposed here are so weak.

Rather than applying the Lagrangian relaxation to the simplified scheduling problem, which can already be solved by an MILP solver quite quickly, a better approach could be to apply the technique to the full FPGA scheduling problem. Rather than relaxing the constraints that each task is run once, as was done in this chapter, it may make sense to relax different constraints to obtain a better lower bound. One option, for example, may be to dualize the constraint that only one reconfiguration may take place at a time. This constraint was simply removed in section 3.3.1, but dualizing the constraint instead may result in better bounds.

The Lagrangian relaxation method has a been applied to many different optimization problems with great success. Even though this attempt at applying it to the FPGA scheduling problem was unsuccessful, there is some potential that it's possible to achieve good results with more investigation.

6 CONCLUSION

The ability to build faster and cheaper computers by shrinking the size of transistors is potentially approaching physical limitations. Given this development, now may be a particularly interesting time for the creation of novel approaches to build more efficient computers. FPGA's are already demonstrated to be significantly more efficient than the CPU for many different classes of applications. Similarly, dynamic partial reconfiguration is a well researched technology which is already used to better utilize FPGA resources. Further developing this technology is then a promising way of having more efficient computation.

One bottleneck in using DPR is that the overhead time of reconfiguration is still high. A breakthrough in improving reconfiguration speeds could come from 3D microelectronic packaging. This technology is already being used in Xilinx high bandwidth memory devices to package an FPGA and memory, allowing for a very high bandwidth of memory access. It's very plausible that this same technology could be adapted to reduce the overhead of reconfiguration.

6.1 Contributions

The objective of this thesis was to evaluate the potential performance improvement that could result from significantly reducing the reconfiguration overhead. To this end, a model of FPGA scheduling was proposed which is a specialized case of the model from Deiana et. al [7].

While it is possible to simply apply the exact or heuristic schedulers from Deiana et. al, both approaches have downsides. The exact scheduler has a very long runtime, which makes it unsuitable for evaluating more complex applications consisting of many tasks. The heuristic scheduler has a lower runtime, but this comes at the cost of precision. It may be the case that the heuristic scheduler functions better at faster or slower reconfiguration speeds, and this could bias the results of evaluating the impact of reconfiguration speed.

To work around this limitation, a relaxed version of the scheduling problem was introduced which can be used to obtain a lower bound schedule makespan. This lower bound can then be used to evaluate how precise the the results of the heuristic scheduler are.

Using the heuristic scheduler from Deiana et. al in conjunction with the relaxed scheduling problem to obtain lower bounds, the impact of reconfiguration speed was evaluated for a sample of random applications. Given that the randomly generated applications may not be consistent with real-world applications, it's not possible to draw any conclusions yet on what performance improvements can be obtained by improving reconfiguration speeds. Instead, some overall trends were studied.

As reconfiguration speed improves, the sample applications tended to use the FPGA more frequently instead of the CPU. With very fast reconfiguration speeds, the sample applications also tended to transition to using larger FPGA implementations which trade-off FPGA resources for performance. The exact resource vs. performance trade-off available for an application appears to have an effect on how large of a performance improvement is obtainable by increasing reconfiguration speeds.

Based on this tendency for applications to trade-off resources for performance as reconfiguration speed improves, a potential optimization was also studied. In this optimization, tasks of an application are divided and run in serial, allowing the divided tasks to each use more FPGA resources. This optimization had little impact compared to the overall benefits of improving reconfiguration speed and it is questionable to what extent it could apply to real-world applications. Still, it's possible that structuring applications in this way may allow them to better take advantage of improved reconfiguration speeds.

The analysis of the sample workloads was quite limited. The heuristic scheduler of Deiana et. al was much faster than the exact scheduler, but it nevertheless was time consuming for scheduling many sample workloads, each with many tasks. The relaxed scheduling problem also offered quite weak lower bounds, making it challenging to evaluate the po-

tential impact of improved reconfiguration speeds. In an attempt to resolve both of these limitations, the Lagrangian relaxation method was evaluated. The attempt to apply the Lagrangian relaxation was unsuccessful, but the Lagrangian relaxation still appears to be a potentially useful technique to apply to FPGA scheduling.

6.2 Limitations

A fundamental limitation in this work is the use of unrealistic parameters to model the FPGA and applications. The reconfiguration speeds, the total FPGA resources, and the specification of applications used in this work weren't specifically created based on real-world values. The results obtained here are still useful for observing general patterns and tendencies, which may be a useful starting point for future work. Nevertheless, given the unrealistic parameters it isn't possible to answer the question of exactly what performance improvement could result from significantly improving reconfiguration speeds.

A clear and straightforward way to address this limitation would be to bring the parameters of the model into alignment with real-world values. For the reconfiguration speeds and total FPGA resources, this would be relatively straightforward and only require investigating the data-sheets of current FPGA models. Finding realistic parameters to specify applications - the number of resources used in FPGA based implementations and the corresponding runtimes - will be more challenging, though. Since a benefit of faster reconfiguration speeds is that applications may trade-off higher resource usage for improved performance, it's necessary to evaluate not only one FPGA based implementation of a given application, but rather several, with each using a different resource vs. performance trade-off.

6.3 Future Work

While the need for realistic model parameters as mentioned in the previous section is certainly important, it may already be practical to begin other approaches for evaluating the research questions. Even with realistic parameters, models of the FPGA will always be somewhat unrealistic. In the existing model considered in this thesis, some important factors aren't modeled such as how much data must be communicated between components of an application. Streaming data between application components, for example, may re-

sult in completely different performance characteristics than storing data in an external memory.

Rather than try to model applications as abstract sets of tasks, an interesting approach would be to select an actual application which could benefit from improved reconfiguration speeds and implement it to try and take full advantage of DPR. Since the FPGA's under investigation don't actually exist yet, it would be necessary to somehow simulate the application running on an FPGA with fast reconfiguration. Actually implementing a non-trivial application to take advantage of fast reconfiguration speeds would require a significant amount of work, so it would be infeasible to do this for many applications. At best this approach would offer a case study of what fast reconfiguration speeds could offer. Still, the results may offer much more insight than what it is possible to obtain from an abstract model.

Assuming an FPGA with extremely low reconfiguration overhead is feasible, an immediate challenge would be how to best utilize it. Currently, many developers are acclimated to producing software, and developing applications for the FPGA using a hardware description language represents a large shift. Even for FPGA developers, making more extensive use of DPR could be challenging. Introducing new abstractions in programming languages or frameworks may lower the barrier for running performance critical portions of an application on an FPGA. Similarly, new abstractions in hardware description languages may make DPR more straightforward to use. Interesting work in this direction is already underway [12, 3].

BIBLIOGRAPHY

[1] Gurobi optimization. https://www.gurobi.com/.

[2] Biondi, A., Balsini, A., Pagani, M., Rossi, E., Marinoni, M., and Buttazzo, G. A framework for supporting real-time applications on dynamic reconfigurable fpgas. In *2016 IEEE Real-Time Systems Symposium (RTSS)* (2016), pp. 1–12.

[3] Bosch, J., Vidal, M., Filgueras, A., Jiménez-González, D., Álvarez, C., Martorell, X., and Ayguadé, E. Task-based programming models for heterogeneous recurrent workloads. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications* (Cham, 2021), S. Derrien, F. Hannig, P. C. Diniz, and D. Chillet, Eds., Springer International Publishing, pp. 108–122.

[4] Bynum, M. L., Hackebeil, G. A., Hart, W. E., Laird, C. D., Nicholson, B. L., Siirola, J. D., Watson, J.-P., and Woodruff, D. L. *Pyomo–optimization modeling in python*, third ed., vol. 67. Springer Science & Business Media, 2021.

[5] Chodowiec, P., and Gaj, K. Very compact fpga implementation of the aes algorithm. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2003), Springer, pp. 319–333.

[6] DEIANA, E. A. Multiobjective reconfiguration-aware scheduling on fpga-based heterogeneous architectures. Master's thesis, University of Illinois at Chicago, 2015. https://indigo.uic.edu/articles/thesis/Multiobjective_Reconfiguration-Aware_Scheduling_on_FPGA-Based_Heterogeneous_Architectures/10819241.

[7] DEIANA, E. A., RABOZZI, M., CATTANEO, R., AND SANTAMBROGIO, M. D. A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (2015), IEEE, pp. 1–6.

[8] FISHER, M. L. The lagrangian relaxation method for solving integer programming problems. *Management science 27*, 1 (1981), 1–18.

[9] GUERON, S. Intel advanced encryption standard (aes) new instructions set. `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf`, 2010.

[10] HART, W. E., WATSON, J.-P., AND WOODRUFF, D. L. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation 3*, 3 (2011), 219–260.

[11] HOMSIRIKAMOL, E., ROGAWSKI, M., AND GAJ, K. Throughput vs. area trade-offs in high-speed architectures of five round 3 sha-3 candidates implemented using xilinx and altera fpgas. In *Cryptographic Hardware and Embedded Systems – CHES 2011* (Berlin, Heidelberg, 2011), B. Preneel and T. Takagi, Eds., Springer Berlin Heidelberg, pp. 491–506.

[12] JUNGBLUT, P., AND KRANZLMÜLLER, D. Dynamic spatial multiplexing on fpgas with opencl. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications* (Cham, 2021), S. Derrien, F. Hannig, P. C. Diniz, and D. Chillet, Eds., Springer International Publishing, pp. 265–274.

[13] KALMS, L., AND GOHRINGER, D. Exploration of OpenCL for FPGAs using SDAccel and comparison to GPUs and multicore CPUs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (Sept. 2017), IEEE.

[14] KUON, I., AND ROSE, J. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems 26*, 2 (2007), 203–215.

[15] LDVBIN. Clb block diagram. `https://commons.wikimedia.org/wiki/File:CLB_Block_Diagram.png`. Accessed 05-08-21, distributed under CC-BY 3.0 license.

[16] LI, Y., AND GOYAL, D. *Introduction to 3D Microelectronic Packaging*. Springer Singapore, Singapore, 2021, pp. 1–16.

[17] MACK, C. A. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing 24*, 2 (2011), 202–207.

[18] MARTELLO, S., SOUMIS, F., AND TOTH, P. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete applied mathematics 75*, 2 (1997), 169–188.

[19] PAAR, C., AND PELZL, J. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[20] RIPOLL, I., CRESPO, A., AND MOK, A. K. Improvement in feasibility testing for real-time tasks. *Real-Time Systems 11*, 1 (1996), 19–39.

[21] SIROWY, S., AND FORIN, A. Where's the beef? why fpgas are so fast. *MS Research* (2008).

[22] THEIS, T. N., AND WONG, H.-S. P. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering 19*, 2 (2017), 41–50.

[23] TOFALLIS, C. A better measure of relative prediction accuracy for model selection and model estimation. *Journal of the Operational Research Society 66*, 8 (2015), 1352–1362.

[24] TRIMBERGER, S., CARBERRY, D., JOHNSON, A., AND WONG, J. A time-multiplexed fpga. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)* (1997), IEEE, pp. 22–28.

[25] VIPIN, K., AND FAHMY, S. A. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv. 51*, 4 (July 2018).

[26] VUJANIC, R., AND ESFAHANI, P. M. Nsopy. `https://github.com/robin-vjc/nsopy`. Accessed 13-08-21.

[27] WANG, C., GONG, L., YU, Q., LI, X., XIE, Y., AND ZHOU, X. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36*, 3 (2016), 513–517.

[28] WOLSEY, L. A. *Integer Programming*, first ed. Wiley-Interscience, 1998.

[29] XILINX. Power methodology guide ug786. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug786_PowerMethodology.pdf. Accessed 06-08-21.

[30] XILINX. Ultrascale architecture configuration ug-570. https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf. Accessed 09-08-21.

[31] XILINX. Ultrascale architecture dsp slice ug-579. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf. Accessed 06-08-21.

[32] XILINX. Ultrascale architecture memory resources ug-573. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf. Accessed 06-08-21.

[33] XILINX. Virtex ultrascale+ fpga data sheet ds-923. https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf. Accessed 09-08-21.

[34] XILINX. Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance. https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf. Accessed 13-08-21.

[35] XILINX. Vivado design suite user guide: Partial reconfiguration ug-909. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf. Accessed 06-08-21.