2020-04-13

Distributed Memory Based FPGA Debug

Robert Benjamin Hale
Brigham Young University

Distributed Memory Based FPGA Debug

Robert Benjamin Hale

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Brad Hutchings, Chair
Brent Nelson
Michael Wirthlin
James Archibald
Jeffrey Goeders

Department of Electrical & Computer Engineering

Brigham Young University

ABSTRACT

Distributed Memory Based FPGA Debug

Robert Benjamin Hale
Department of Electrical & Computer Engineering, BYU
Doctor of Philosophy

Field-programmable gate arrays (FPGAs) are powerful integrated circuits for low-overhead custom computing needs and design prototyping. Due to the hardware nature of programming an FPGA, finding bugs in a design can be a very challenging process. Signals need to be physically probed and data recorded in real time. This is often done by dedicating some resources on the FPGA itself towards an embedded logic analyzer. This method is effective but can be time and resource consuming. Academic research projects have produced a variety of methods for reducing this difficulty.

One option that has previously been unexplored is the use of distributed LUT memory for debug trace buffers, rather than dedicated FPGA BRAM. This dissertation presents a novel, lean embedded logic analyzer that leverages leftover LUT resources on the FPGA for this purpose. Distributed Memory Debug (abbreviated as "DIME Debug") provides some amount of signal visibility into very large (90%+ LUT utilized) FPGA designs or designs where the programmer requires all available device BRAM, situations in which currently available embedded logic analyzers are likely to fail.

The ubiquitous nature of LUTs on FPGAs provides opportunities to insert debug circuitry near signals of interest without disturbing placement of the user design. Using only leftover LUTs for trace buffers allows for effectively no area overhead. The DIME Debug system typically has a critical path delay in the 7-9ns range, which can force non-ideal slower timing constraints on the user design. A simulated annealing based placement algorithm and other optimizations are shown to improve timing closure results from 20-50% depending on benchmark and probe count. DIME debug can be instrumented into a fully implemented design incrementally using the RapidWright CAD tool, resulting in debug iterations under 15 minutes even for very large benchmarks.

Another interesting possibility introduced by the use of memory LUTs for debug trace buffers is preallocating these resources. Setting aside a certain number of LUTs before implementation of the user design leaves them available for incremental debug instrumentation. Experiments with a preallocation scheme show that, with virtually no penalty to the user design, debug critical paths are lowered by approximately 1ns and 2-3X the number of trace buffers can be instrumented into most benchmarks.

Keywords: FPGA, debug, embedded logic analyzer, distributed memory

ACKNOWLEDGMENTS

I would first like to profoundly thank my advisor Dr. Brad Hutchings. Without his wealth of wisdom, patience, and awareness, the completion of this degree would certainly have not been possible. His influence has positively affected me academically and beyond.

I would like to thank members of my family for helping me down this path. A short conversation with my brother, Wayne, was what first lead me to pursue a degree in Electrical and Computer Engineering, and his advice has continued to remain useful through the years. My mother has given me positive academic encouragement from the beginning, helping me succeed in high school and beyond. Many others of my family have shown support and interest in my endeavors that helped me keep going through these several years of research.

I would like to thank my friends and peers in the Configurable Computing Lab. First was Josh Monson, who began as my graduate student advisor but ended as my fellow PhD student. Josh consistently provided teaching and mentoring to help me along the way. Next are Adam Burnett and Adam Hastings, my partners in crime when I was pursuing a Master's Degree. Followed by several other lab friends, including John-Paul Anderson, Travis Harold, Tanner Gaskin, Andrew Keller, and others. Many of our interactions, some scholarly and some not, some in normal working hours and some not, helped me immensely in getting through this long journey. Finally, while not a member of the lab at the same time, Chris Lavin deserves special recognition both as my contact at Xilinx while working with their tools and as one of the primary creators of the RapidWright CAD tool (and its predecessor, RapidSmith) that was a vital part of my research endeavors. This research was supported financially by Xilinx Research Labs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

Many of the contributions presented in this dissertation have been previously published by the author within research conference papers [1–3]. The ideas, methods, experiments and results from those papers are expounded in much more detail than was possible in conference paper format. They are also supplemented with additional background, experiments, and contributions. Chapter 4 builds upon the work presented in [1] concerning enabling embedded debug in large designs. Chapter 5 covers the timing impact discussion from [2], while the simulated annealing placement method from that paper is covered in Chapter 6. Chapter 6 also includes and expands on the preallocation scheme and lengthened debug trace buffer ideas originally published in [3].

CHAPTER 1. INTRODUCTION

1.1 Motivation

Field programmable gate arrays (FPGAs) are a powerful alternative to more common integrated circuits such as CPUs and ASICs. FPGAs can be repeatedly re-programmed exactly to an engineer's need, allowing for fully customized, efficient designs with no manufacturing overhead and low power consumption. This flexibility makes them well suited to a variety of low-volume, compute-intensive applications, such as spacecraft or data centers, as well as inexpensive prototyping of designs destined for eventual fabrication.

However, this flexibility comes at the cost of programming difficulty. FPGAs are hardware devices, which makes designs challenging to create and even more challenging to debug. Unlike software debug, FPGA circuit data are not easily observed and programs cannot easily be halted in order to view data changing one step at a time. Design signals must be individually probed at a digital level and observed in real time by a logic analyzer. Resulting binary data must then be carefully understood and interpreted in order to locate bugs. Physical limitations will typically prohibit the observance of all design signals simultaneously. If the set of design signals that need to be observed changes, the entire design typically needs to be re-implemented. Implementation for large FPGA designs can require many hours of compilation time. A recent study of functional verification trends revealed that the percentage of total FPGA project time spent on verification has been rising year to year, reaching an average of 50% in 2018. Debug consumes an average of 42% of verification time [4].

While it is possible to use an external logic analyzer to physically probe FPGA design signals for debug, it is more common to use an embedded logic analyzer for this purpose. With this method some of the circuitry on the FPGA itself is dedicated to debug. FPGA-to-host interfaces are used to feed captured signal data back out to the host computer for observation. In addition to the aforementioned challenges present to FPGA debug, embedded logic analyzers introduce another

potential roadblock: consumption of FPGA circuitry. If the user has created a large design, it is possible that there simply will not be enough FPGA left to include an embedded logic analyzer. Another situation that can create this problem is a design that requires most or all of the memory on the FPGA, such as memory intensive image processing applications [5]. Current embedded logic analyzers require the use of Block-RAM (BRAM) on the FPGA to store observed data. These on-chip memory resources are scarce [6] and can become unavailable for debug if an engineer requires most, or all, of them for their design. In these situations currently available embedded logic analyzers become infeasible options for FPGA debug.

1.2 Summary of Research

In consideration of the difficulties discussed in the previous section, this research has aimed to create a novel embedded debug tool to alleviate some of the challenges involved with FPGA debug. First, this tool enables debug in extremely large designs that use as much as 94% of the LUTs or 100% of the BRAMs on the FPGA. This is achieved by using small LUT-based memory for debug trace buffers. In addition, the time required for debug iterations (when the set of signals to be observed is changed) is significantly reduced using this method. Where currently available embedded logic analyzers may require many hours for an iteration, the tool presented here requires an average of 8.8 minutes. This speedup is achieved by performing debug instrumentation incrementally, after the user design has been completely implemented.

Titled Distributed-Memory (DIME) Debug, this new method leverages LUT resources left unused by the user design. When Xilinx FPGA LUTs are used as memory they are referred to as distributed memory, providing the name of the tool. Even the largest and densest user design will leave sparse LUT resources unused and DIME Debug takes advantage of this leftover circuitry. These leftover LUTs are configured as Xilinx Shift-Register LUTs (SRL). SRLs can act as small, completely contained trace buffers to store signal data in first-in, first-out fashion.

A tool from Xilinx Labs called RapidWright [7] is used for incremental instrumentation of DIME Debug. RapidWright is a library written in the Java coding language that allows manipulation of FPGA designs outside of what Xilinx provides with the Vivado Design Suite [8]. With RapidWright, SRL trace buffers can be placed in post-PAR designs in under three minutes on average. The standard Vivado debug iteration requires a full implementation of the user design and

can easily consume several hours for large designs – long enough to consume an engineer's entire work day. In comparison, the entire DIME Debug instrumentation cycle can be completed in, on average across all benchmarks and experiments, under nine minutes.

An important consideration for an embedded debug system is how much it will alter the original user's design. The less this impact, the less the risk of the debugger hiding bugs or preventing implementation of the combined user and debug circuit. The two primary concerns are (1) area overhead and/or placement alteration and (2) increase in critical path delay.

DIME Debug can be instrumented with, effectively, no FPGA area overhead. This is achieved by scavenging for and only using leftover, unused LUTs on the device after the user circuit is fully implemented. Placement of the user design is never altered. LUTs are an abundant, small resource on FPGAs. DIME Debug trace buffers, consuming as little as two LUTs each, are thus extremely lean. This allows the DIME debugger to fit onto the FPGA even when 90% or more of the LUTs on the FPGA are consumed by the user design. In our experiments, when targeting designs of these sizes, commercial embedded logic analyzers were unable to fit onto the FPGA. Using distributed memory for trace buffers also allows the user design to consume all BRAM on the FPGA if needed.

As with all embedded logic analyzers, DIME trace buffers connect directly into design signals. This carries the risk of increasing the wirelength of those signals and possibly increasing the critical path delay of the design. Experiments on Kintex Ultrascale FPGAs indicate that instrumented DIME Debug circuitry has a minimum propagation delay in the range of 6-8ns (166-125MHz). Benchmarks with critical path delays already in this range saw relatively small timing closure penalties (under 20% slowdown) after DIME Debug is instrumented, however, benchmarks capable of operating at faster clock periods saw their minimum clock period brought into this 6-8ns range. For some benchmarks, this forced significant clock slowdown in order to meet design timing closure. Maintaining a clock period reasonably close to the original frequency can be critical for finding design bugs. Several optimizations, described below, are implemented into DIME Debug in order to reduce these timing penalties.

One method of lowering propagation delays is by bringing connected design elements closer together and thereby reducing wirelengths. For DIME Debug, this can be done by placing trace buffers near the signal they are monitoring. Initially, a greedy approach is used for

placement. Trace buffers are placed on the closest available LUTs at the time they are created, first-come first-served. This is later enhanced by following up the initial greedy placement with a simulated annealing placement algorithm. Simulated annealing placement iterates over many possible solutions with the goal of a placement with the lowest total distance between DIME Debug trace buffers and the sources of the nets being probed. Dependent on the benchmark and requested number of debug probes, this enhancement reduced the critical path penalty from 2-3ns and/or allowed as many as twice as many probes to be instrumented while maintaining the same timing constraint. This optimization is especially effective at improving DIME Debug results when higher numbers of debug probes are being instrumented.

As another method of improving signal-to-buffer proximity, allowing debug circuitry to have priority on some LUT resources is considered. Rather than rely strictly on leftover resources for debug, a small amount (approximately 1% of LUTs on the FPGA) of device resources are set aside before implementation of the original design. The user is unable to use these resources, leaving them available for DIME Debug trace buffers to later occupy. Preallocated locations may allow DIME buffers to be placed closer to their respective sources. Benchmarks with the preallocation scheme in place saw debug critical path penalties reduced by up to 2ns. In addition, since unused LUTs on partially used FPGA sites are difficult to instrument post-PAR, preallocation ensures that LUT resources are organized such that they remain valid locations for DIME Debug trace buffers. For most benchmarks, this resulted in a 2-3x increase in the number of trace buffers that could be instrumented.

While preallocation was shown to benefit the capabilities of DIME Debug, it also introduces a question of how it might affect the original design. Limiting the design from using some FPGA resources can have obvious repercussions. However, experiments show that preallocating only 1% of LUT resources has almost no effect on the original design. With the preallocation scheme in place, all benchmarks implemented without issue and saw, at most, a critical path penalty of 0.1ns.

Other paths within the DIME Debug system are considered for timing issues as well. Routes between DIME trace buffers only ever operate at the frequency of the on-chip JTAG clock and never at user clock speed, which is typically several times faster than the JTAG clock. These paths could be the bottleneck restricting the maximum frequency at which the design, when in-

strumented with debug circuitry, can operate. Multicycle path constraints inform Vivado routing software that these paths are acceptable in the final implementation even if they cannot operate at the same clock frequency as the user design. Experiments with multicycle path constraints revealed the buffer-to-buffer path to indeed be the critical path for most benchmarks. Once the clock period requirements on debug paths is loosened with multicycle path constraints, these benchmarks saw a critical path decrease of up to 2ns in some cases. Similar to simulated annealing placement, this optimization typically had the greatest effect when high numbers of probe counts were requested.

In addition to improving timing impacts, experiments were also conducted into seeing if DIME Debug trace buffers could be deepened. One of the primary drawbacks of this debug system is the small memory capacity of LUT-based trace buffers. The initial iteration of DIME Debug, in which a trace buffer occupies a single memory LUT, could only hold 16 bits of signal history at a time. While better than no data at all, 16 bits is very shallow and likely limiting to debug efforts. Rather than use a single 16-bit SRL for each trace buffer, eight 32-bit SRLs are chained together on a single FPGA site. This allows for 256-bit DIME trace buffers for a 16x depth increase. A 256-bit trace buffer requires eight LUTs instead of one, resulting in roughly 1/8th the total number of design nets that can be probed. However, overall debug memory is approximately doubled and lower routing congestion improves critical path penalties, typically by around 1ns.

Another significant drawback of DIME Debug at present is the lack of advanced triggering mechanisms, an important part of any on-chip debug system. For the experiments in this dissertation, a signal from the user design is used to indicate to the control system that data gathering has completed. A more complete version of DIME Debug would include more complex and robust triggering options. This, and other potential improvements to DIME Debug, are discussed in more detail in the Future Work section of Chapter 9.

All experiments are conducted on a unique suite of five benchmarks created for this research project. Singular FPGA designs that are both large enough to adequately test the hypothesis of this work and simple to manipulate using RapidWright are challenging to either create or procure. Instead, smaller modules were used and duplicated enough times to reach certain LUT utilization threshholds, namely, 70, 80, and 90% (or as close to these cutoffs as module size would allow). Benchmark modules include an LC3 processor (LC3), a sudoku puzzle solver (sudoku), a random number generator (RNG), a micro-FIFO queue (uFIFO), and a random pulse generator

(RPulseG). The LC3 module is based off of the work in [9] and programmed by the author of this dissertation. The other four modules were obtained as open-source code from OpenCores [10].

1.3 Contributions

1. Demonstrated that LUT-based distributed memory can be used in place of traditional BRAM trace buffers for embedded FPGA debug using the DIME Debug tool.

2. Demonstrated that distributed memory trace buffers can be instrumented into FPGA designs that are already consuming as much as 94% of the LUTs on the device.

3. Showed the impact of instrumented DIME Debug trace buffers on a design in terms of critical path delay using experiments that are repeated several times with different timing constraints.

4. Demonstrated that a simulated annealing based placement algorithm for DIME trace buffers is effective in reducing critical path penalties, allowing increased debug probe counts operating at faster clock frequencies.

5. Demonstrated that preallocating a small portion of FPGA resources towards debug before implementing the user design significantly improves debug abilities in most cases with almost no penalty to the original design.

6. Demonstrated that buffer-to-buffer routes in the DIME Debug system can become the critical path in a design. Showed that loosening timing requirements on these routes with multicycle path constraints significantly improves debug instrumentation success rates for most benchmarks.

7. Demonstrated that DIME trace buffers can be deepened from 16-bit to 256-bit.

8. Created a set of 5 unique benchmarks created using module duplication to reach desired percentages of FPGA LUT utilization. These benchmarks are used to test instrumentation of distributed memory trace buffers on very large designs.

CHAPTER 2. BACKGROUND AND RELATED WORK

In this chapter the present landscape of FPGA debug will be briefly reviewed. This discussion includes features and shortcomings of currently available logic analyzers and academic research projects and how they relate to the contributions of this dissertation.

2.1 FPGA Debug

After an FPGA design has been created and the logic passes verification tests, the next step in finding remaining bugs is to program the design onto an FPGA and observe it in real time. This section will discuss the challenges involved in this process and current methods used to address these challenges.

2.1.1 Observability

An important step of fixing any type of problem is to observe and interpret relevant information. This is especially true of programming, where many variables and processes are intertwined and rapidly changing. When debugging most software programs a wealth of information is readily available. A programmer can print out variable values at pertinent points in the code or even halt the code and step through it one instruction at a time while virtually every bit of data involved is easily observable. The physics of the underlying hardware are abstracted away from the programmer.

FPGAs, however, are hardware devices. While the mathematical logic of hardware description language (HDL) code can be verified with simulation, the functionality of the code may fail when programmed onto real circuitry and operated at speed. New bugs can surface as the physical limits of wires and transistors come into play and test the soundness of a design – for example, the propagation delay of some device wires may be slower than the code development software estimates. This issue would be impossible to simulate. Without almost any abstraction of these

physical issues, an engineer must debug by probing signals in the design and observe voltages changing in real time. The engineer must also be careful that the debug process itself does not alter the operation of the circuit and hide or create additional bugs.

2.1.2 Logic Analyzers

For on-chip FPGA debug, a device called a logic analyzer is employed. A logic analyzer physically probes wires on an FPGA while the design is operating. Voltage on the wires is sampled at intervals that match the clock frequency of the design. The observed voltages are recorded in digital memory and can be displayed as linear waveforms for the engineer to view and interpret. Numerous design signals are probed and their outputs aligned and compared in an attempt to discover flaws within the design. Logic analyzers can be either external or embedded.

An external logic analyzer is an additional piece of hardware separate from the FPGA. The design nets under consideration are programmatically connected to external hardware pins on the FPGA development board. Wired probes from the logic analyzer are connected to these pins. The design is operated and, once signaled by a trigger condition, the logic analyzer will begin recording the values observed on the pins. External logic analyzers are valuable for keeping the amount of FPGA resources required for debug relatively low. The nets will require some additional routing to be wired to pins, which may affect timing closure, but FPGA memory and logic cells are not necessary. These types of logic analyzers are also valuable for monitoring multiple devices at once, simultaneously providing visibility for signals from multiple sources. However, hardware logic analyzers are limited by the number of external pins available on the development board and can be monetarily expensive.

Embedded logic analyzers are made up of some of the circuitry within the FPGA itself. FPGA vendors have created their own IP for this purpose, such as the Xilinx Integrated Logic Analyzer [11] or Intel's SignalTap [12]. Embedded logic analyzers avoid the need for additional expensive hardware. Instead of routing signals out to development board pins, signals are routed to on-chip memory. Control and triggering systems must be included within FPGA fabric. Embedded logic analyzers require resources on the FPGA beyond what the user design requires. Rather than being limited by external board pins, debug becomes bottlenecked by the number of on-chip resources left available.

When using either external or embedded logic analyzers, the engineer must consider how the logic analyzer itself could impact the user design being debugged. This is because, in the case of either type of analyzer, additional wires are being instrumented on the FPGA that will be in contact with design wires and need to meet design timing constraints. If this additional circuitry alters the timing of the user design, new bugs can be introduced and/or current bugs may seemingly be hidden. For example, if the debug circuit forces the entire design to operate at a slower clock period than it was originally created for, bugs related to timing may no longer be present at all. Finding such bugs would be critical if the design is required to operate at or above a certain clock frequency.

Another common challenge when using logic analyzers is the time consumed for each debug iteration. Typically, regardless of the logic analyzer used, it will not be possible to simultaneously observe all design signals relevant to a bug. After a first debug iteration an engineer may realize that additional design signals need to be probed. This will require new routing on the FPGA and typically involves a full re-compilation of the design. For small designs this can take tens of minutes, and for the largest and most complex designs it can take many hours. In either case, multiple debug iterations can become unacceptably time consuming. Commercially offered embedded logic analyzers typically offer a method of incremental compilation that attempts to reduce time consumed by reusing some design elements that have already been placed and routed. However, in our experiments using Xilinx software, this feature only resulted in trivial speedup of debug iterations.

2.1.3 FPGA Memory

Memory on an FPGA typically falls into two categories: dedicated and distributed.

The first is dedicated memory such as Block-RAM (BRAM). There will be relatively few BRAMs on an FPGA compared to other logic pieces, however, each BRAM can hold memory volume in the kilobyte range. All BRAM on an FPGA cumulatively have memory capacity in the dozens of megabytes range. BRAMs are preferable for relatively large memory needs of a design. BRAM are typically located in clusters in certain areas of the FPGA layout and require a clock cycle after addressing for data to be available.

The second type of memory available on an FPGA is Lookup Table (LUT) based distributed memory. This is when certain logic resources on the FPGA are purposed to act as addressable memory. There are significantly more LUTs on an FPGA than BRAM, however, a single LUT RAM can only hold memory volume in the range of dozens to hundreds of bits. If all device LUTs were to be implemented as RAM, cumulative memory capacity is typically in the range of several megabytes. Distributed memory is fast and memory can be accessed in the same clock cycle that it is addressed. Not all LUTs can be leveraged as memory, but both memory- and non-memory-LUTs are finely spread throughout the majority of the FPGA's layout.

On Xilinx FPGAs, memory-LUTs can be implemented as Shift-Register LUTs (SRL). SRLs are limited in size compared to simple LUT RAMs, but all the logic required to act as a shift register is included within the LUT [13].

2.2 Related Research

Commercially available embedded logic analyzers are powerful tools for providing observability into FPGA designs. However, they require significant FPGA resources and engineer time and are not perfect solutions. Many academic research projects have attempted to further improve the process of on-chip FPGA debug. This section will review the recent developments in this field and discuss some similarities and differences to the contributions in this dissertation.

2.2.1 Increasing Observability

One method of reducing the time consumed when debugging an FPGA is to circumvent debug iterations altogether. These iterations are necessary because, using traditional methods, only a relatively small subset of design nets can be probed and observed at a time. Research projects have aimed to eliminate this problem by attempting to speculate an inclusive set of signals to be probed beforehand [14, 15] or by attempting to probe nearly all the nets in the design simultaneously. The latter method is achieved by multiplexing a large percentage of nets under consideration into a smaller number of memory buffers. Debug iterations thus become trivial as the only change necessary is deciding which signal is passed through the multiplexers for observation [16–19]. These

methods are effective in reducing iteration count, but, when necessary, require longer recompilation times and some incur significant logic overhead [16].

These multiplexed methods are one form of a debug overlay. Overlays are a somewhat recently popular idea that have been researched for several possible enhancements to FPGA use, including debug [20]. An overlay is a virtual layer between the FPGA and user that acts as a form of user interface, designed to simplify how the user interacts with the device. Debug overlays can reduce iteration time by managing the debug configuration and simplifying alteration steps. Overlays allow post-PAR changes at a fraction of the time of a full recompilation [21]. In addition to multiplexing debug signals, overlays have been used for assisting in debug triggering [22, 23]. Complex overlays have even been used to provide near-software levels of debug visibility [24]. Once implemented, overlays can significantly simplify the debug process. However, they can require significant overhead in chip resources to implement and/or engineer time to create.

2.2.2 Scan-Based Debug

Scan-based debug is another method for improving design observability. This approach involves capturing a snapshot of the entirety of the device state at a given moment and then passing this data out to the host. One option for scan-based debug is scan-chains, similar to those included on non-field-programmable integrated circuits. This requires using significant additional FPGA logic to tie into each flip-flop on the chip, requiring as much as an 84% area overhead [25, 26]. Alternatively, some FPGAs have built in "readback" functionality, eliminating the need for additional circuitry [27, 28].

Both methods of scan-based debug are excellent for offering extensive visibility into the design and, in the case of readback, may be able to do so without significant overhead or design changes. However, these techniques require the design to be halted in order to draw design data to the host. Extracting this information consumes 2-8 seconds [27], resulting in significant design execution slowdown. Design halt also poses a risk of disrupting interactions with other devices communicating with the FPGA [29]. In addition, readback is not capable of viewing the design state of all FPGA logic. Combinational design elements don't retain a state that can be observed, nor are the contents of shift-register LUTs visible (as discovered during the course of this research).

Some projects have attempted to alleviate the burden of design halt and take advantage of the high visibility benefits of scan-based debug by adding additional control circuitry, providing a more comprehensive debugging environment. Iskander et al. use readback to enable a somewhat software-like breakpoint debug system in [27]. Hybrid methods, that combine aspects of both scan-based and trace-based debug (discussed in the following subsection), are employed in [30, 31] to minimize the number of design halts and provide signal history.

2.2.3 Trace-Based Debug

The most commonly used method for embedded debug is trace-based. Trace buffers are memories used to store the data observed on design signals at run time. A triggering mechanism is used to tell buffers when to begin (or cease) storing new signal data. A short history of changes on the signal being probed is saved. Histories from a number of design nets can be saved in trace buffers and then compared to find issues. The number of design signals that can be viewed is significantly lower than a scan-based system, however, trace-based debug has the critical advantage of being able to observe the design being tested while it is operated at speed [32].

Most commercially available embedded logic analyzers are trace-based debug systems [11, 12, 33]. However, these tools all have similar drawbacks. They require significant on-chip resources, alter the timing and placement of the user's design, and/or consume significant engineering time in debug iterations. Research projects focused on trace-based solutions typically aim to address at least one of these concerns.

One common way to decrease debug iteration time is by moving the point of instrumentation. Commercial tools typically instrument debug logic alongside the user design. While this does allow for more control of placement and routing of the combined circuit, it can also make significant changes to the user design and is hugely time consuming each time the debug net selection needs to be altered. Instead, many research projects insert debug circuitry incrementally, moving the point of instrumentation until after the user's design is already placed and routed [34–40]. Isolating debug implementation from design implementation allows for much faster debug iterations [38] and, when done carefully, leaves the design unaltered [40].

2.2.4 Reducing Debug Critical Path

To prevent any possibility of obscuring bugs, many research projects related to FPGA debug strive to keep their impact on the user design as low as possible. Most commonly, this is done by incrementally instrumenting debug logic using only FPGA fabric that the user design has left unused [18, 23, 36, 38–41]. However, one disruption to the user design that is very difficult to avoid is that debug probes must physically tie into the nets being observed. This means that probe nets will be under the same timing constraint as the net being observed. If the net being probed is the critical path of the design, connecting the probe introduces a strong possibility of increasing this critical path. The entire design, when instrumented with debug circuitry, is therefore forced to operate at a lower clock frequency, potentially obscuring bugs related to timing. The authors of [36–39] acknowledge that inclusion of debug circuitry negatively affects the critical path of the combined design. While this impact should only affect the combined circuit, resolving when the debug circuit is removed, it remains a risk during the debug phase.

Hung and Wilton address this issue in some of their works. In [36], probed nets with the least timing slack are given priority to nearby BRAM trace buffers. In [40], a novel pipelining method is used to virtually isolate debug nets from design nets. In both projects, a trivial critical path penalty is observed when using these optimizations.

2.2.5 DIME Debug and Related Work

DIME Debug is a trace-based embedded debug approach that shares many similarities with those described in this section. Debug circuitry is inserted incrementally, improving design visibility by keeping debug iteration time low. Incremental insertion respects placement of the user design in order to avoid bug-obscuring changes. The primary novel contribution of this approach is in the use of LUT-based, distributed memory for trace buffers, rather than FPGA BRAM. Until now, no other debug research has explored distributed memory for this purpose.

Using distributed memory for trace buffers allows DIME Debug to be extremely lean and able to target benchmarks that consume nearly all FPGA resources–as high as 94% of device LUTs. Most similar research projects have not considered instrumenting such large designs. Hung, Eslami and Wilton target devices that are only just large enough to hold the benchmarks being tested in

some of their works [36, 37], however, these experiments were conducted on theoretical FPGA devices using the academic VPR tool [42]. DIME Debug is implemented targeting a commercially available Xilinx Ultrascale FPGA using the RapidWright toolset [7]. Hung and Wilton later use Torc [43] to conduct experiments to maximize routing to debug trace buffers on Xilinx Virtex FPGA [40, 41, 44]. The largest benchmark targeted in those experiments consumes 95% of the *sites* on a Virtex 6. It is difficult to draw a comparison between site and LUT utilization, as sites contain several LUTs. Only one LUT must be occupied to consider the site used. Of the benchmarks instrumented with the DIME Debug tool, the largest consumes 99.3% of sites on a Kintex Ultrascale. In addition, benchmark BRAM utilization is another important consideration. Hung and Wilton's tool requires as much as 75% of the device BRAM for trace buffers where DIME Debug requires zero. The user design is able to consume up to 100% of BRAM on the chip.

Distributed memory trace buffers provide opportunities for novel methods of reducing debug critical paths. Proximity between elements on an FPGA route is directly related to propagation delay — the closer the source to the sink, the lower the delay [45]. This principle is leveraged for DIME Debug placement. Similar to [36], the ability to connect any net of interest to any trace buffer is leveraged. However, unlike BRAM trace buffers, LUT-based trace buffers can be placed virtually anywhere across the FPGA. Greedy and simulated annealing algorithms are used for trace buffer placement to minimize distance between buffers and probed signals.

Another technique employed to optimize trace buffer placement and timing is a resource preallocation scheme. Using this method, the user design is prevented from using certain LUT resources on the chip, ensuring they are available for debug. Preallocation improves the chances that a trace buffer will be located very near the net it is to be tied into, minimizing wirelength and improving critical path delay. While some moderately similar methods of prioritizing FPGA resources for debug have been employed previously, they were intended to ensure debug logic would fit into the device at all [39] or to reduce iteration time [46], not minimize critical path delay. The effect of resource preallocation on FPGA debug and the user design as explored in this dissertation has not been addressed in any prior work (with the exception of being suggested as a possibility for future work [18, 40]).

CHAPTER 3. DISTRIBUTED MEMORY BASED FPGA DEBUG

This Chapter will thoroughly introduce Distributed Memory (DIME) Debug. This will include our original hypothesis for DIME Debug, how DIME trace buffers are implemented and utilized, and the limitations of this method.

3.1 Hypothesis: Utilizing Shift Register LUTs for FPGA Debug is Feasible.

The primary objective of the research presented in this dissertation is to explore the possibilities of using distributed, LUT-based FPGA memory for embedded debug trace buffers. It is hypothesized that distributed memory can provide two primary advantages over BRAM memory that is traditionally used for this purpose.

Availability. LUTs are a lean, abundant resource on FPGAs. There are 64,000 memory-LUTs spread across a Kintex KU025 [47, 48]. Each of them can be configured as a shift-register LUT (SRL), allowing the LUT to act as a self-contained trace buffer. If debug control circuitry is kept similarly lean, it would be possible to instrument these tiny trace buffers into even the largest of FPGA designs and provide some signal visibility. Since most embedded debuggers require significant on-chip resources, a distributed memory based debug system could enable embedded debug on very large designs where conventional debuggers may not have enough spare resources to fit.

While Block RAM represent a much larger memory space, there are significantly fewer individual units on FPGA devices. In comparison to the 64,000 memory-LUTs on the KU025, there are only 360 Block RAMs. An engineer's design could conceivably require all of the BRAM on the chip, but such a design consuming every available LUT is far less likely. Distributed memory trace buffers could also enable embedded debug in memory-intensive designs where no BRAMs are available.

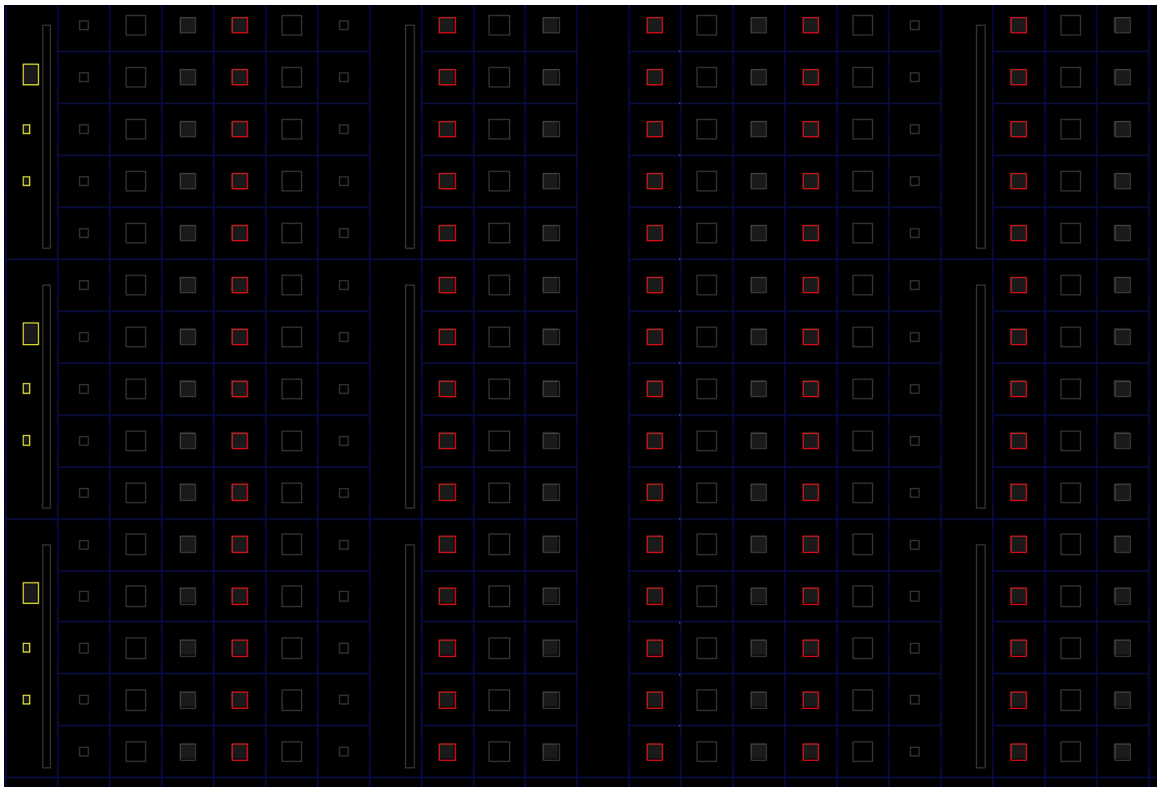


Figure 3.1: Section of KU025 FPGA with BRAM (yellow, left of image) and memory-LUT (red) highlighted. In comparison to BRAM, LUTs are more abundant and more evenly dispersed across the FPGA.

Proximity. The high volume of memory-LUTs on FPGAs also represent a very thorough, fine grained distribution across the device. While BRAM are evenly distributed across the chip, they are low in number and spaced far apart when compared to LUTs. On the Kintex KU025, BRAM are horizontally separated by 12-30 device tiles in comparison to 2-8 tiles between memory-LUTs (Figure 3.1). This becomes significant for debug because of the potential for tight proximity between trace buffers and probed signals. As discussed in Chapter 2, tighter proximity allows for shorter wirelengths and critical paths, keeping timing impact from debug low. Memory-LUTs, due to their abundance and distribution in comparison to BRAM, have the potential to be located much closer to the signals they probe.

The ubiquitous nature of LUTs presents unique opportunities for deciding where and how to place debug trace buffers. Distributed memory trace buffers can be placed on any of many available memory-LUT locations. Placement algorithms can be used to further optimize proxim-

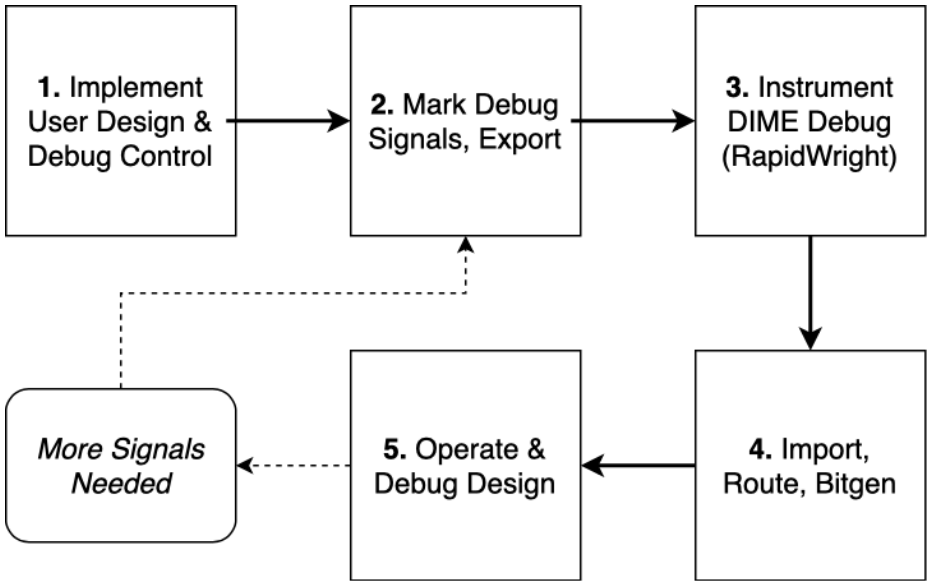


Figure 3.2: DIME Debug workflow.

ity between buffer and signal. Experiments can also be conducted to determine if giving debug circuitry some priority over LUT resources (preallocation) improves the overall system.

The RapidWright CAD tool [7] can be used to quickly create and instrument distributed memory trace buffers into Xilinx designs. Isolating the design and debug implementation steps (incremental instrumentation) will allow for fast debug iteration times.

3.2 Overview of DIME Debug Flow

DIME Debug is instrumented into a design and used for debug according to the following steps (Figure 3.2).

1. The engineer creates their design using the Xilinx Vivado Design Suite up through the place and route steps. The engineer may complete design implementation while remaining largely agnostic of specific debug needs, with the exception of two small pieces of generic control circuitry that must be included. The first is a simple state machine that will be used to govern the operation of DIME trace buffers. The second is a BSCAN primitive. BSCAN primitives are Xilinx device blocks that allow the FPGA circuitry to communicate to the host machine using the JTAG interface [49]. This BSCAN primitive will be used to export signal histories from FPGA to host. The state machine and BSCAN primitive are both 'generic' in that they

will not need to be customized based on the configuration of the debug system. This control logic need only be inserted once, will not require recompilation during debug iterations, and consumes minimal FPGA resources (approximately six LUTs and two flip-flops).

2. Assuming bugs exist in the implemented design, the engineer will mark design signals to be traced and export the design. Signals are marked for debug in Vivado in the same manner as though the Xilinx ILA was going to be used [11]. A Xilinx Design CheckPoint (DCP) is created and exported into the RapidWright [7] toolset.

3. DIME Debug trace buffers are instrumented into the design. A Java program, leveraging the RapidWright library, imports and analyzes the design and identifies the nets marked for debug. For each net, one DIME trace buffer is created and placed. A new altered DCP is created and exported.

4. The instrumented DCP is imported into Vivado. Routing is finalized and a bitstream created.

5. The combined circuit is operated on the FPGA in order to gather and view signal data. During operation, probed data is being continuously stored in DIME trace buffers; the oldest data is shifted out to make room for new. Once triggering has occurred, the DIME Debug system is halted (the user design continues to operate). The engineer can then request debug data via the Vivado command prompt. The data is converted into a waveform for viewing and interpretation.

If debug data was insufficient to find the bug and different or additional probes are needed, the engineer can return to the second step and repeat this process. The instrumentation flow, from probe request up to bitstream generation, consumes 8.8 minutes on average. This time can vary, depending on the design being tested and the number of debug probes requested, between around 4 and 14.4 minutes. Approximately 65% of this time is spent finalizing routing in Vivado (step 4, above) and could be reduced in future work if routing was completed with RapidWright.

3.3 DIME Debug System Details

3.3.1 Trace Buffer Composition

Each DIME trace buffer consists of only two FPGA cells: an SRL to store debug data, and a 2-to-1 MUX for control. The MUX output is tied to the input of the SRL. One of the inputs of the MUX will be connected to whichever design net is being probed, while the other is tied into the rest of the debug system. Specifically, this second MUX input is tied to another DIME trace buffer pair. In this way, all trace buffers in the system can be configured into a single chained shift register (Figure 3.3).

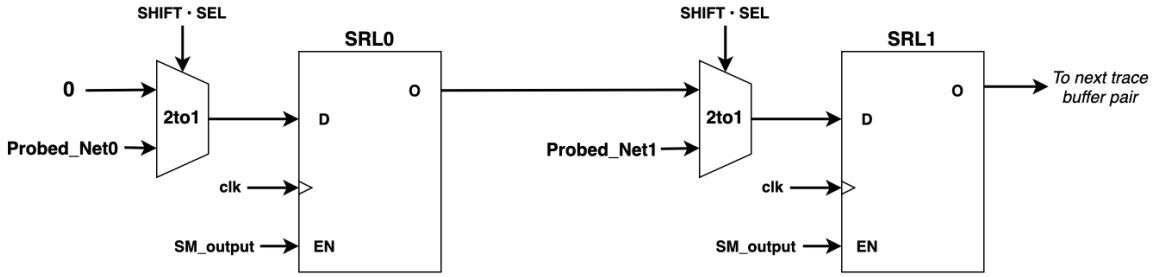


Figure 3.3: Chained DIME trace buffer pairs

The chain terminates at the BSCAN primitive inserted alongside the user design and is used to serially retrieve recorded signal histories. The MUX is switched by JTAG signals from the BSCAN primitive. The SRL is clocked using the same user clock that drives the signal being probed. However, the SRL clock enable signal is driven by the state machine inserted alongside user circuitry. These control signals are described in more detail in the following section.

3.3.2 Debug System Control

In order for signal data to be accurately recorded and later observable, a debug system needs more than memory. The system also needs control logic to send commands to the trace buffers and later send signal data to the engineer at the host machine.

As mentioned in the previous chapter, embedded logic analyzers typically employ either a scan-based or trace-based approach. A hybrid approach was originally considered for DIME

Debug. After signal data had been captured in distributed-memory trace buffers, readback could be used to bring that data to the host machine without any additional circuitry being added to the design. This technique was considered in order to minimize debug resource overhead, since leanness is one of the primary advantages of DIME Debug. However, it was discovered that Xilinx readback does not provide visibility to the contents of a LUT that has been programmed as an SRL and therefore not feasible. Instead, a run-time system is needed to properly deliver the contents of an SRL to the host. The state machine and BSCAN primitive implemented during the initial design phase are used for this purpose.

Figure 3.4 gives a visual overview of the DIME Debug control system. Note that signals SHIFT, SEL, and DRCK are standard JTAG interface signals that are outputs of the BSCAN primitive. SEL asserts high to indicate that this particular BSCAN primitive is the one being addressed by the JTAG system (as there are multiple present on the FPGA). SHIFT is asserted high when the JTAG chain is receiving serial data via the BSCAN input, TDO. DRCK is the clock operating the JTAG system. These signals respond to commands sent to JTAG from the host via Vivado.

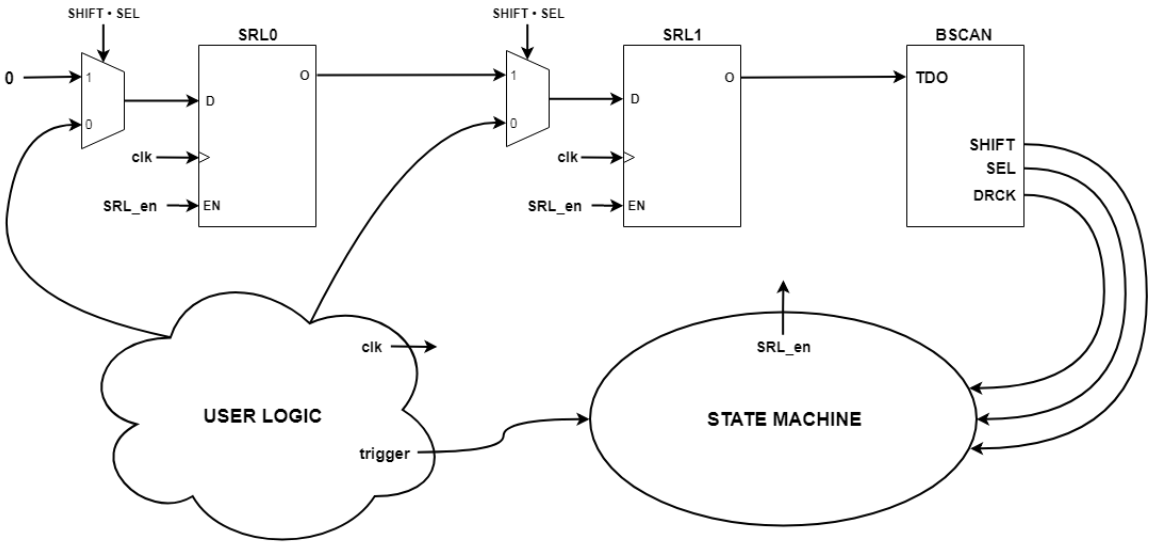


Figure 3.4: Visual overview of an entire two-probe DIME Debug system

The DIME Debug system has two operating modes: run mode and debug mode.

Run mode. This mode is, essentially, standard operating mode. No debug actions have yet taken place since the trigger has not been asserted. The design is operating at speed and probed

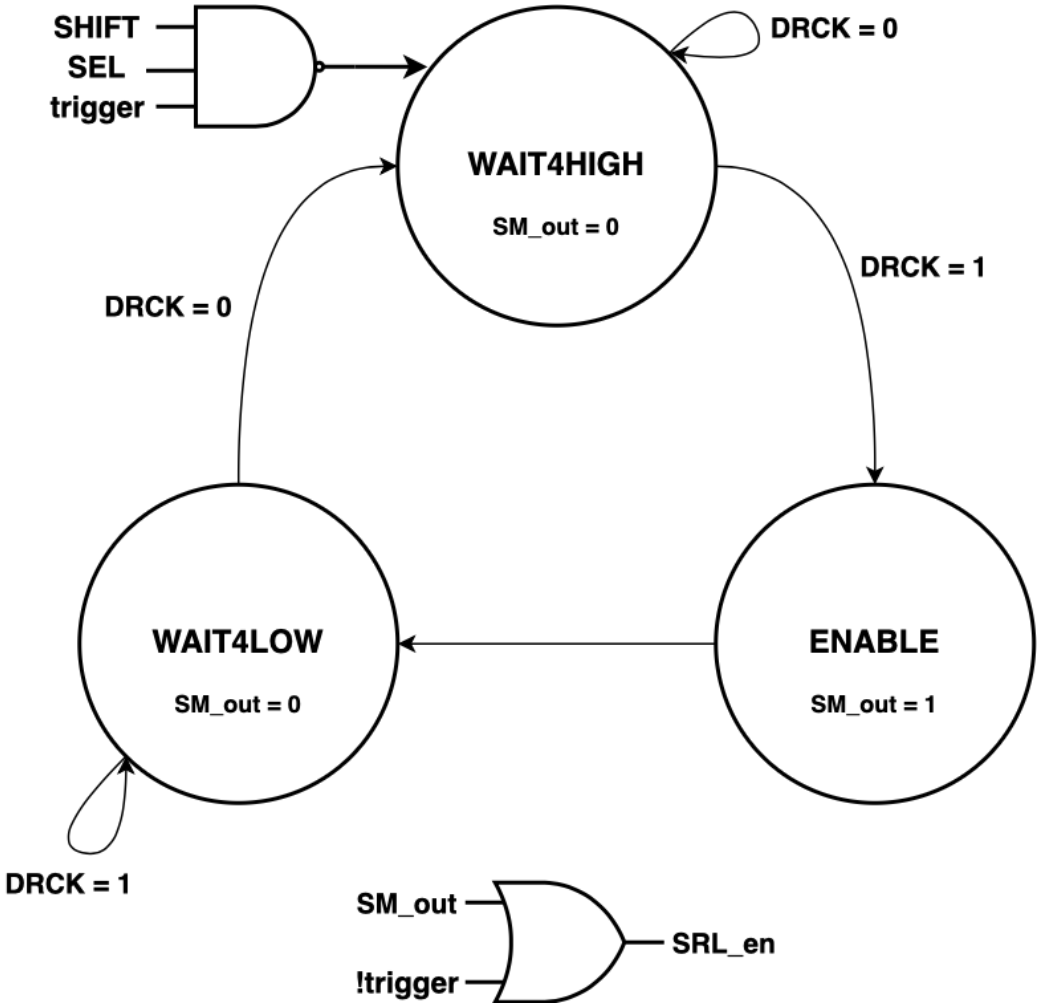


Figure 3.5: DIME Debug control state machine diagram

signal data is being constantly collected into the SRLs and disposed of in first-in, first-out fashion. The state machine is held in reset. SRLs are constantly clock enabled so long as the trigger has not fired. MUX select signals remain low, such that design data is passed into SRLs, since no BSCAN commands are received from the host.

Debug mode. Once the trigger has fired, the system enters debug mode. Data recording is halted and signal histories are available to be passed from the FPGA to the host for observation. Once requested by the host via JTAG commands, signal histories will be transmitted through the BSCAN primitive.

The primary function of the state machine is to ensure recorded data is passed to the JTAG chain at the correct clock speed (Figure 3.5). SRLs are clocked at user speed. In run mode, they

are constantly enabled and can record data at user clock speed. However, the JTAG clock is several times slower than the user clock. The state machine is also clocked by the user clock, but uses the JTAG clock as an input. For each rising edge of the JTAG clock, the state machine will activate the SRL clock enable signal for the duration of a single user clock period. Since the SRL uses the user clock for its own clock signal, this allows it to effectively cross the clock domains and pass data at the rate of the JTAG clock. A waveform depicting these clock relationships is shown in Figure 3.6.

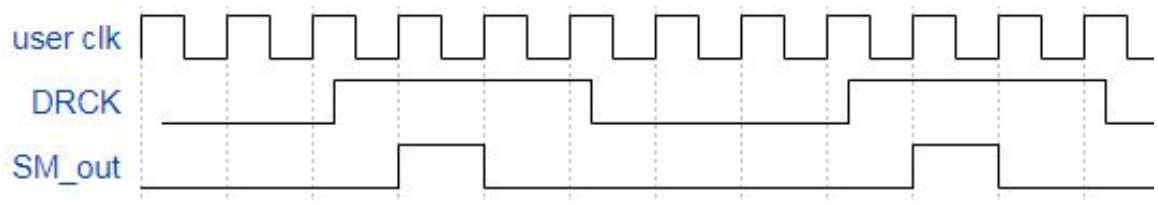


Figure 3.6: Waveform comparing JTAG clock (DRCK) to state machine output.

The Vivado command line is used to send instructions to the on-chip JTAG chain (though any command line tool capable of interfacing with a JTAG chain could be used for this step). These instructions are manifested on the FPGA circuitry as BSCAN signals. The debug system responds to these commands by activating the state machine, chaining together all trace buffer SRLs (by configuring the MUXs), and serially passing recorded data to the BSCAN primitive. This data is received at the host in binary format, which is processed by additional scripting commands in order to create a Value Change Dump (VCD) file. Knowing the order in which SRLs are chained makes it possible to identify individual trace histories by signal name. The VCD is then converted into a visual waveform with any software that supports the format, such as ModelSim [50].

3.3.3 Creation and Instrumentation of DIME Debug

To instrument DIME Debug independently from the rest of the user design, an incremental implementation method is needed. The Xilinx Vivado Suite includes a few options, such as an incremental compilation method and a set of command line tools in the TCL language. However, both of these options execute too slowly for many of the possible benefits of DIME Debug to be realized. Instead, the open-source Java based CAD tool RapidWright [7] is used.

RapidWright is capable of altering Xilinx Design Checkpoints (DCP), giving backend access to much of the same functionality available in Vivado. These alterations can be done in a fraction of the time needed to do the same operations with TCL commands or standard implementation methods. Once a user design has been placed and routed, a DCP is created and imported into RapidWright. RapidWright represents the various parts of the design (such as cells, nets, even the design as a whole) as Java objects. These objects can be analyzed and manipulated at a software level. For further information, complete RapidWright documentation is available online [51].

In the case of instrumenting DIME Debug trace buffers, the RapidWright program iterates through the list of nets that have been marked for debug. For each one, two design cells are created. The first cell is programmed as a 2-to-1 MUX. It is placed into the design on an unused LUT as close as possible to the source of the signal being probed. The second cell is programmed as an SRL and is placed as near as possible to the MUX. Netlist entries are created tying the MUX to the signal being probed, the SRL to the MUX, and both to other control and clock signals (as shown in Figure 3.3). Once all debug probes have been placed, a new altered DCP is saved. The entirety of the RapidWright phase is completed in an average of 3 minutes. Additional details about DIME Debug instrumentation with RapidWright can be found in Appendix A.

After trace buffers are placed and netlist entries are created with RapidWright, Vivado route is used to complete instrumentation. Existing route rip-up is allowed if necessary to meet timing constraints. RapidWright is capable of more quickly finalizing design routing, however, Vivado is used for simplicity as routing is not the primary focus of this project. The Vivado router is more powerful than the basic tool currently available in the RapidWright library and a more advanced RapidWright router is saved for future work.

3.3.4 Proof of Concept

DIME Debug can be instrumented and provide signal visibility on currently available Xilinx FPGAs. As an initial proof of concept, this tool is used to observe the output of a simple 4-bit counter. The correct functionality of the counter was verified with the Xilinx ILA before attempting to observe the same signal with DIME Debug. The waveform captured with DIME Debug is shown in Figure 3.7.

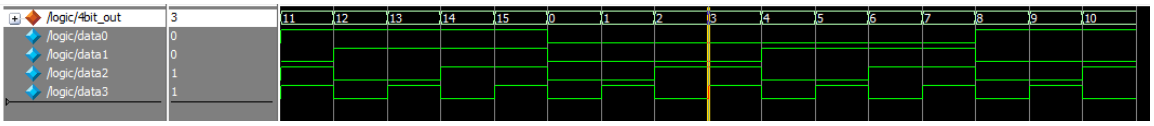


Figure 3.7: Waveform produced by a 4-bit counter observed with DIME Debug.

3.4 Limitations

As with any debug system, DIME Debug faces several limitations. Some of these limitations are addressed within this dissertation while others are left for future work.

3.4.1 Timing Impact

An inherent part of embedded debug is physically probing the signals in question to observe if behavior is as expected. This requires additional circuitry to be connected to important routes in the user design, and can cause the timing of those routes to be altered. Altering the timing constraint of a design has the potential to create new bugs and/or hide existing ones.

DIME Debug typically introduces a minimum critical path in the 6-8ns range that increases with higher numbers of probes. Benchmarks already operating at maximum clock speeds in this range suffer mild slowdown with debug circuitry in place. However, designs with faster clock periods were forced to slow down the design clock as much as 5X in order to meet timing constraints. Further details about this timing impact are discussed in Chapter 5. Reducing this penalty is one of the goals of the enhancements discussed in Chapters 6 and 7. Further reduction of timing issues is a subject of future work.

3.4.2 Trace Buffer Depth

Many thousands of signal changes are happening in any given second even on the slowest of FPGA designs. When debugging, the more of this signal data that can be made visible, the better. DIME Debug is a very lean tool able to debug very large FPGA designs, but this comes with the trade-off of short trace buffer depth. The initial version of DIME Debug buffers are only capable of storing 16-bits of signal history. Alterations to trace buffer composition, discussed in

Chapter 8, are used to demonstrate a depth increase to 256-bits at the cost of fewer probes. This method can be used to further increase depth in future work.

3.4.3 Triggering

A critical part of any debug system is cueing trace buffers to begin storing signal data. Memory is finite and the most important data must be identified for storage–particularly considering the depth limitation of DIME trace buffers. However, one of the most significant obstacles for DIME Debug in becoming a full-fledged debug tool is the current lack of advanced triggering options. For our experiments, a signal from the user design is selected to act as a trigger when asserted. Simple LED indicators or additional BSCAN primitives are used to provide trigger visibility to the host and ensure debug data is not requested until triggering has occurred. More advanced triggering methods would be a vital part of a future robust DIME Debug system. Potential solutions are discussed in the Future Work section of Chapter 9.

CHAPTER 4. DIME DEBUG FEASIBILITY STUDY

This chapter will describe the preliminary experiments used to test the capabilities of DIME Debug. These experiments are used to find out if DIME Debug can feasibly be used to probe nets in highly utilized designs. The same experiments are conducted with the Xilinx Integrated Logic Analyzer [11] and results compared. All experiments target the Xilinx Kintex Ultrascale KU025 FPGA [47]. This FPGA model was selected because Ultrascale devices were among the first to be supported by RapidWright, however, any Xilinx FPGA supported by RapidWright could be instrumented with DIME Debug. Xilinx Vivado version 2017.3 is used for experiments in this chapter as well as the remainder of this dissertation.

4.1 Experiment Details

4.1.1 Benchmark

In order to test the efficacy of DIME Debug on large designs, large benchmarks are needed. Partially due to some difficulties with RapidWright[1], it was found to be very challenging to procure or create a single design that was both adequately large and suited for our experiments. Instead, a small module is used to create a larger design. The small module is duplicated many times within the design, resulting in a single large benchmark.

Benchmarks created with duplication come with the advantage of being easily grown or shrunk to reach certain thresholds of device utilization. An LC3 processor [9] is used as the module for the benchmarks experimented on in this chapter. The LC3 processor was duplicated on-chip enough times to reach KU025 LUT utilization thresholds of 70, 80, and 90 percent.

[1] Using RapidWright to modify designs with extensive hierarchy is a major endeavor outside of the scope of this research. The hierarchy of most easily accessible IP and FPGA design modules cannot be flattened.

4.1.2 Variables

In order to test the hypothesis in a robust manner, experiments will be repeated while making adjustments to a variety of variables, listed below.

Embedded Logic Analyzer. This is the primary comparison being made in these experiments. The two embedded logic analyzers being compared are the Xilinx Integrated Logic Analyzer (ILA) and the Distributed-Memory (DIME) Debug tool presented in this dissertation.

Benchmark LUT Utilization Percentage. Embedded logic analyzers require a certain amount of the FPGA logic left unused alongside the user design. The larger the user design, the fewer leftover resources. It is expected that larger benchmarks will be able to accommodate fewer debug probes than smaller benchmarks. The design utilization thresholds that will be experimented on are the aforementioned LC3 benchmarks that utilize 70, 80, and 90% of the LUTs, respectively.

Debug Probe Count. During FPGA debug it is probable that many design signals will need to be observed in order to track down a bug. However, each additional probe means more logic consumed and more routes that may introduce higher critical path delays. It is expected for instrumentation success rates to fall as higher numbers of probes are requested. The number of probes that will be requested will start at a reasonably low value, determined based on benchmark size, and raised in even intervals for additional experiments.

Design Net Selection. Probe count is not the only aspect of debug probes that must be considered. The selection of design nets to be observed by these probes is important as well. This is because net selection can have an impact on critical path. For example, if the source of a net of interest is located in a more sparsely utilized area of the implemented design, it is more likely that leftover logic for debug circuitry will be found nearby this net. The routing path from net to trace buffer will be shorter and less likely to increase the critical path of the instrumented design.

FPGA designs typically have hundreds or thousands of signals, making full coverage of net selection infeasible for these experiments. Instead, a reasonably large number of random net selections will be used. While controlling for logic analyzer, LUT utilization, and probe count, each experiment will be repeated 200 times. A new random set of nets will be targeted for debug in each of these repetitions. These repeated experiments will reveal an average rate of success, given the controlled variables, regardless of net selection. Two hundred repetitions was determined through trial and error to be sufficient for reasonably consistent results.

4.1.3 Execution

The Brigham Young University supercomputer [52] is used to complete all steps of these experiments in a timely fashion. A series of Bash and Tcl scripts are used to send jobs to and interact with the supercomputer, keeping experiments and the results thereof organized. Completion time will be logged for each experiment in addition to instrumentation success or failure.

Experiments are executed in the following steps.

1. Benchmark and probe count are selected.
2. A random selection of nets within the benchmark is made, equivalent in number to probe count.
3. Xilinx ILA instrumentation is attempted using Vivado incremental implementation techniques, targeting the selected nets.
4. DIME Debug instrumentation is attempted using RapidWright and Vivado incremental route, targeting the selected nets.
5. Steps 2-4 are repeated a total of 200 times.
6. Steps 1-5 are repeated for each benchmark and probe count configuration.

4.2 Results

The resulting implementation success rates from these experiments are charted in Figures 4.1, 4.2, and 4.3. A bar that reaches 100% indicates that implementation succeeded for all 200 randomized probe net selections with the given parameters of benchmark, logic analyzer, and probe count. Complete lack of bar indicates that none of the experiments succeeded with those parameters, regardless of net selection.

At 70% utilization (Figure 4.1), a very high number of probes can be instrumented using either debug method; around a 600 probe cutoff for the ILA, while DIME Debug maintains success $>$50% up to 1500 probes. When 80% of LUTs are utilized (Figure 4.2), up to 240 ILA probes can be instrumented, and for DIME Debug, success remains above 50% up to 900 probes. The most significant results are seen at 90% LUT utilization. The Xilinx ILA cannot be instrumented into

this design even at the lowest probe count of 12. DIME Debug was able to probe up to 144 nets with a high success rate.

There are two primary causes for implementation failure in these experiments. The first is failure to meet timing constraints. This failure is seen in the charts when success rates decline gradually as higher probe counts are requested. At higher probe counts, it became more and more likely that debug net selection created an adverse impact on the critical path delay of the design and caused timing failure. Timing failure is mostly seen from experiments with DIME Debug, though it can also be seen with a few ILA experiments on the 80% utilized benchmark. This is likely due to the ILA having more refined timing optimizations than DIME Debug, particularly at this early stage of DIME Debug's development.

The other common type of failure is from resource exhaustion. This is indicated by a sudden drop of success to zero as probe counts increase. This is seen for the Xilinx ILA on the 70 and 80% utilized benchmarks (above 600 and 240 probes, respectively), and for DIME Debug on the 90% utilized benchmark (above 144 probes). This is discussed in more detail in Chapter 5 and partially mitigated for DIME Debug in the preallocation experiments of Chapter 6.

Implementation time was recorded for each successful experiment. For this set of experiments, DIME Debug instrumentation is at least 2.8x faster than ILA instrumentation on average (see Table 4.1). Unlike the ILA, DIME Debug does not require re-implementation of the entire design in order to instrument debug circuitry.

Table 4.1: Average instrumentation time for DIME Debug and ILA across all probe counts.

Average Time of Successful Instrumentation (minutes)		
Design LUT density	DIME Debug	Xilinx ILA
70%	12	76
80%	15	42
90%	14.5	N/A

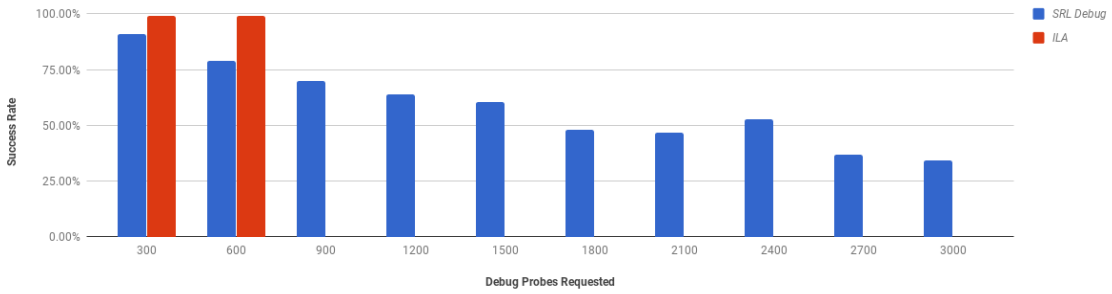


Figure 4.1: Outcomes for Experiments on a 70% Utilized LC3 Design.

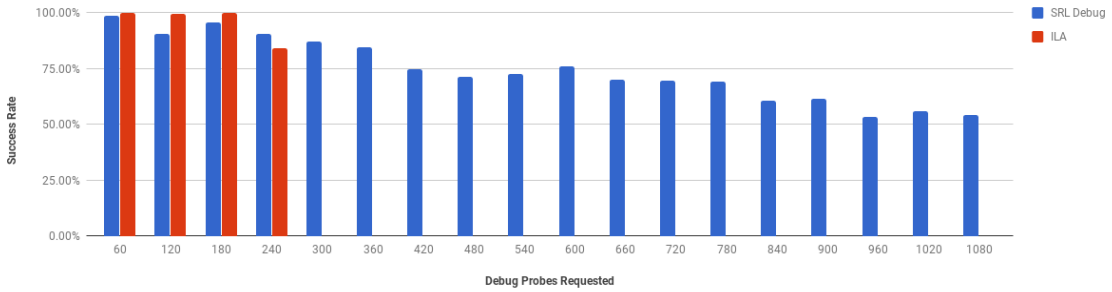


Figure 4.2: Outcomes for Experiments on a 80% Utilized LC3 Design.

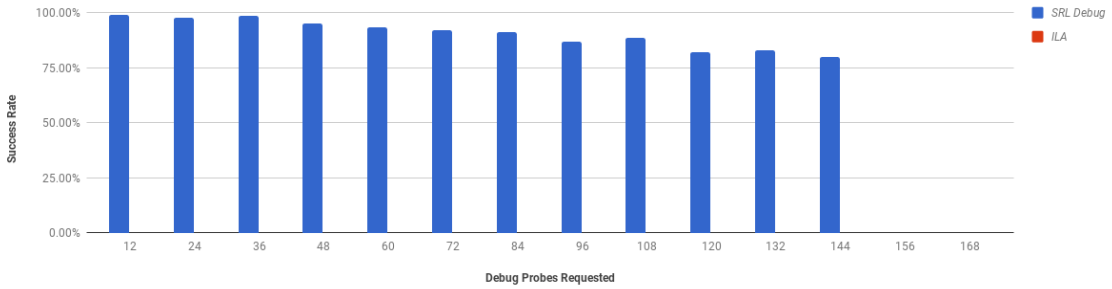


Figure 4.3: Outcomes for Experiments on a 90% Utilized LC3 Design.

4.3 Conclusion

The results of the experiments in this chapter successfully demonstrate that DIME Debug can feasibly be used to enable signal observability into a design that is too large to accommodate the Xilinx Integrated Logic Analyzer. For the smaller benchmarks where the ILA was able to be instrumented, higher numbers of probes could be instrumented when using DIME Debug. DIME Debug instrumentation is also completed in under 15 minutes on average by leveraging Rapid-Wright, several times faster than ILA instrumentation. The primary advantage of the ILA in these experiments is in trace depth. Each DIME Debug probe has a depth of 16-bits where the ILA

probes provide 1024-bits of signal history. Extending the depth of DIME trace buffers is explored in Chapter 8 of this dissertation.

An interesting result shown in these experiments is the tapering off of DIME Debug success rates as higher numbers of probes are requested (most noticeable in Figure 4.1). This indicates that the higher the probe count, the more success becomes dependent on net selection. The failed tests in these situations are timing failures, emphasizing the importance of net selection in maintaining a low critical path delay. Timing constraint is not a variable of these experiments and was kept at a constant 10ns (100MHz). This constraint was used when creating the benchmarks, as well. Further exploration of timing constraint and its impact on DIME Debug is explored in the following Chapter of this dissertation. Enhancements to DIME Debug to improve timing results are presented in Chapters 6 and 7.

CHAPTER 5. DIME DEBUG IMPACT ON TIMING CLOSURE

Chapter 4 showed the feasibility of using DIME Debug in large FPGA designs, however, design timing constraint was not considered as a variable in those experiments. This chapter will discuss how instrumenting a DIME Debug system affects the timing closure of designs and present experiments to quantify this impact.

5.1 On-Chip Debug Timing Impact

Any additional circuitry added to an FPGA runs the risk of increasing the complexity of the design and reducing performance. This is especially true of an embedded logic analyzer since it will need to be physically tied to important design signals (Figure 5.1). This fan-out may increase the propagation delay of the route, forcing the user clock that drives the signal being probed to operate at a lower frequency. Changing the timing characteristics of the design can force additional timing closure effort or hide bugs.

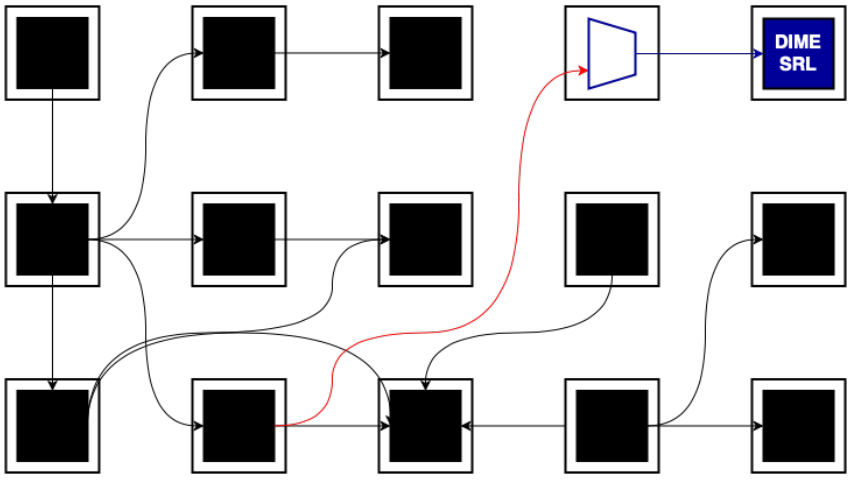


Figure 5.1: Long debug routes like the red one in this diagram can become the critical path of the design.

It is hypothesized that using memory LUTs as debug trace buffers reduces this risk in comparison to BRAM trace buffers due to the ubiquitous nature of LUTs on FPGAs. One factor that plays into propagation delay is wirelength. Unlike BRAM, which are fixed in a few sparse locations on an FPGA, LUTs can be found in many locations, possibly very close to the signal of interest. A shorter path between buffer and design signal decreases the chance of modifying the timing characteristics of the route. This chapter will explore the timing penalty incurred from instrumenting DIME Debug into benchmark designs.

5.2 Testing Timing Impact

The previous chapter of this dissertation demonstrated that DIME Debug can be instrumented into designs when a commercial logic analyzer failed. However, minimal context was provided as to how the timing of those designs was affected, since the same, somewhat relaxed 10ns constraint was used in all cases. In this chapter the impact on design critical path incurred from instrumenting DIME Debug will be investigated. This will be done by repeating the experiments of Chapter 4, but adding a new variable: timing constraint. Instead of a single constant clock period across all experiments, the clock period will be swept.

In order to provide results for a diversity of designs, an additional four benchmarks will be tested along with the LC3 benchmark used in the previous chapter. The range of clock periods tested will vary based on benchmark. First, each benchmark's baseline clock period will be found. The baseline is the fastest timing constraint that can be used while successfully implementing the design and meeting timing closure prior to instrumenting debug circuitry. Baselines will be found through trial and error. Implementation will be repeated while lowering (speeding up) the clock constraint in 0.1ns intervals. The last clock constraint used prior to timing closure failure will be considered the baseline.

Experiments will then be conducted beginning with a clock constraint rounded down to the nearest nanosecond, slightly faster than the baseline. The clock period will then be increased (slowed) by 1ns and experiments repeated. The clock period will continue to be increased by 1ns and experiments repeated until further slowing the clock no longer improves implementation results. This broad sweep of clock constraints will provide a quantification of the impact DIME Debug has on the critical path delay of the benchmarks.

All benchmarks will be created in Vivado with the aforementioned baseline fastest possible clock period that allows implementation to complete and timing to be met (these values are provided in Table 5.1 below). The LC3 benchmark used in Chapter 4 was created using a slower 10ns clock constraint which resulted in different success rates than seen in this chapter. Note also that only benchmarks with LUT utilization greater than or equal to 90% will be presented and discussed moving forward. This is because results of experiments on smaller benchmarks either track closely to the results on 90%+ utilized benchmarks or, for some optimization experiments, show no noteworthy improvement. The primary focus of this research is embedded debug on the very largest of designs and experiments are designed for that case.

As in Chapter 4, each experiment maintains the same number of probes, clock speed, and LUT utilization while being repeated 200 times. Each of these repetitions will target a different, random set of nets to be probed. In this way, these results show overall likelihood of success of a certain configuration regardless of nets selected for debug.

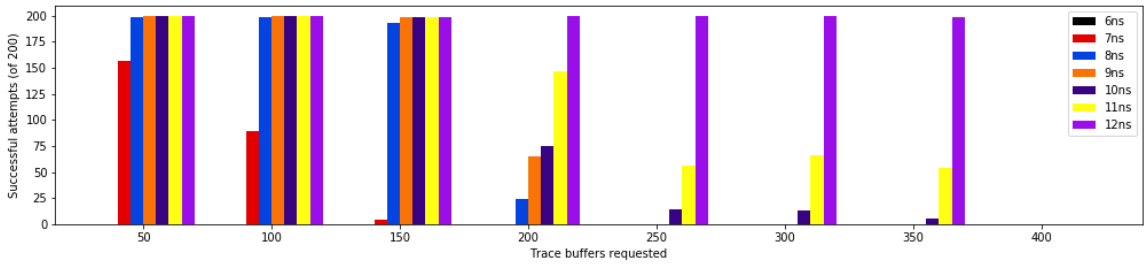
5.3 Results

Results of all experiments are shown in Figure 5.2. The sets of 200 experiments that represent a single timing constraint are clustered together in each colored bar of a chart. If the bar reaches 200, that means 100% of experiments with that configuration of variables were successful. Note that some experiments using clock constraints that offered no additional insight have been removed; only one clock period on the fast end that results in zero successful experiments is shown, as well as only one or two clock periods on the slow end that result in 100% success (for experiments where 100% success was achieved).

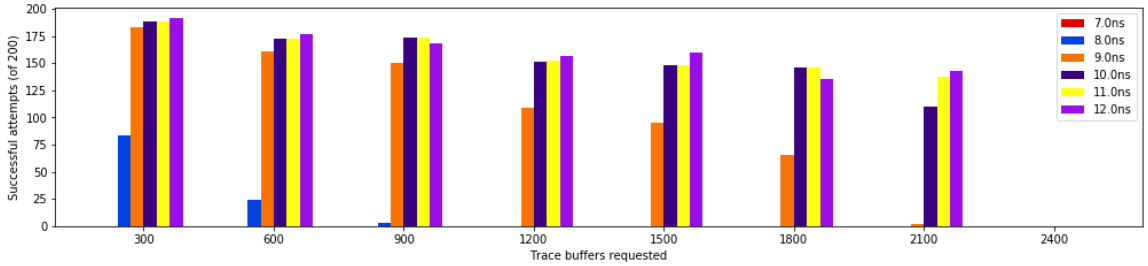
These experiments provide further insight into the two primary factors that prevent implementation of the instrumented design: timing constraint and device resources.

5.3.1 Timing Constraint

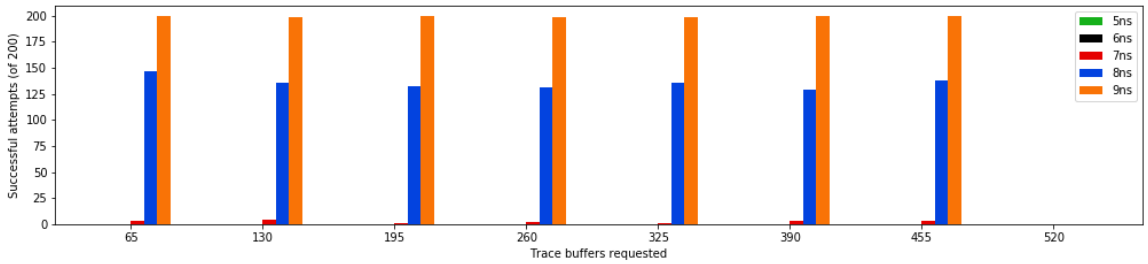
One observation from these results is a minimum propagation delay of the instrumented DIME Debug circuitry. This can be seen by comparing the baseline clock period of each benchmark to the fastest clock period successfully implemented in these experiments (Table 5.1). De-



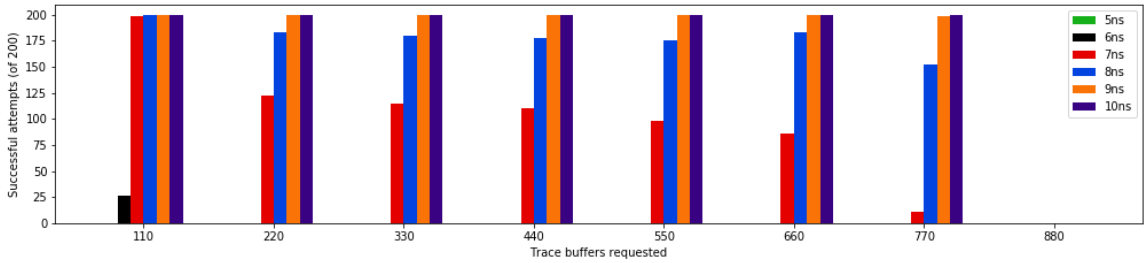
(a) LC3 benchmark with 90% LUT Utilization



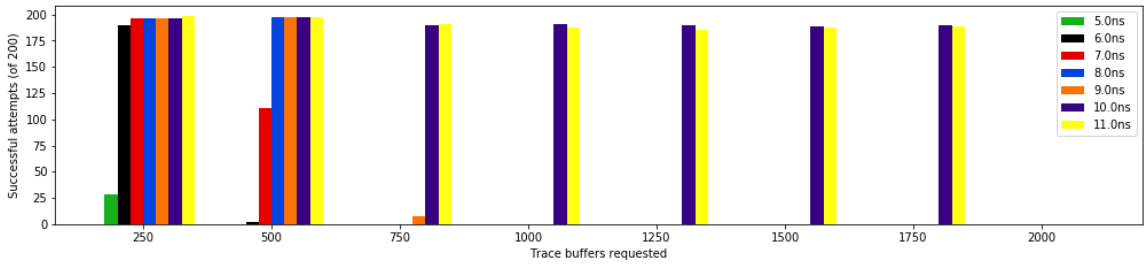
(b) sudoku benchmark with 94% LUT Utilization



(c) RNG benchmark with 90% LUT Utilization



(d) uFIFO benchmark with 90% LUT Utilization



(e) RPulseG benchmark with 90% LUT Utilization

Figure 5.2: Success rates for all five benchmarks while sweeping timing constraint.

pending on the benchmark, designs instrumented with DIME Debug have a minimum critical path of 6-8ns. This only requires a minor clock slowdown when instrumented into benchmarks that were already operating near this period, however, this forces as much as a 5X slowdown in order to meet timing on benchmarks that could operate at much faster clock periods without debug circuitry.

Table 5.1: Comparison of minimum clock period before and after DIME trace buffers are instrumented.

Minimum Clock Period			
Benchmark	Baseline	Instrumented	Penalty
LC3 (90%)	6ns (166.7MHz)	7ns (142.8MHz)	1ns (17%)
sudoku (94%)	7ns (142.8MHz)	8ns (125MHz)	1ns (14%)
RNG (90%)	1.6ns (625MHz)	8ns (125MHz)	6.4ns (500%)
uFIFO (90%)	3.7ns (270.3MHz)	7ns (142.8MHz)	3.3ns (190%)
RPulseG (90%)	1.6ns (625MHz)	6ns (125MHz)	4.4ns (375%)

The timing penalty grows beyond this minimum as higher numbers of DIME trace buffers are requested. This can be easily seen, for example, when observing the results of a 7ns timing constraint on the uFIFO benchmark (red bar on Figure 5.2 (d)). Success begins very high at 110 probes, drops to around 50% success from 220-660 probes, and drops again to near-zero at 770 probes. A similar trend is observed on numerous clock periods for most benchmarks.

5.3.2 Device Resources

In addition to falling success rates due to timing constraint, resources left available on the FPGA become a limitation at a certain point. This cutoff can be seen in results for all benchmarks, when the highest number of probes is requested (probe counts were intentionally increased until complete failure). At this point, the device no longer has enough unused LUTs that can be used for DIME trace buffers.

There is a discrepancy here that should be noted. The Kintex Ultrascale KU025 is a very large FPGA with 145,440 LUTs in total. A design using 90% of this resource leaves behind 14,544 unused LUTs. DIME trace buffers require two LUTs each. Theoretically, a 90% LUT utilized design could still host 7,272 DIME trace buffers. While that count is unlikely to be reached even

in ideal circumstances (due to other limiting factors such as routing etc.), it is still over 3x greater than the maximum of 2100 we see in these experiments. There are two probable reasons for this discrepancy.

The first is that each DIME trace buffer requires one memory LUT and one non-memory LUT. When these benchmarks were created, the 90% utilization figure included both of these types of LUTs. Not only can memory LUTs be used as standard LUTs in the user design, but there are also somewhat fewer memory LUTs on the KU025 than non-memory. Thus, of the remaining 10% of LUTs left unused on a 90% utilized FPGA, it is possible for less than half of those remaining LUTs to be the memory type. This would restrict the number of DIME trace buffers that could be instrumented, even if additional non-memory LUTs were available.

The second, and likely more significant, reason is that LUTs are not perfectly packed into FPGA sites during implementation of the original design. While each site contains eight possible LUTs, it is very common for fewer than that to be implemented within any given site. Out of concern for inordinately affecting the circuitry of the user design, the DIME Debug instrumentation tool is restricted from placing trace buffers onto sites that are partially used by user design logic. Thus, of the 10% of LUTs left unused on the FPGA, a high number are likely to be within sites that are partially used by the user design and therefore not available for DIME Debug. A LUT preallocation scheme intended to alleviate this issue is presented in Chapter 6 of this dissertation.

5.4 Conclusion

Instrumenting DIME trace buffers into a design, as with most embedded logic analyzers, comes at a penalty to the minimum clock period of the design. Even with relatively few probes requested, DIME Debug introduces a critical path delay of 6-8ns. This penalty increases as higher probe counts are requested. A slower clock period can have significant consequences to a design being debugged, such as obscuring the issues the engineer is attempting to track down.

The results of these experiments also reveal that the maximum number of trace buffers that can be instrumented, regardless of clock constraint, is far fewer than could be theoretically placed based on the number of LUTs left unused on the FPGA due to placement limitations.

In consideration of these limitations, optimizations to DIME Debug implementation are presented in the following Chapters 6-8 of this dissertation.

CHAPTER 6. PLACING DIME DEBUG TRACE BUFFERS INTO USER CIRCUIT

DIME Debug provides valuable debug functionality in being able to probe signals on very large user designs. However, when instrumented, it also presents a critical path delay that forces the combined user and debug circuit to operate at a slower clock frequency than the user design alone. This may be unacceptable and it is desirable to mitigate this penalty.

Computing the exact propagation delay of any given route before it is implemented is challenging, however, there are contributing factors that can be considered. One factor that plays a large role is the length of the route [45]. This chapter presents three approaches used to lower DIME Debug critical path delays by reducing distance between elements.

6.1 Greedy Placement

The experiments presented in this dissertation up until this point have used a greedy placement algorithm. A greedy algorithm is a first come, first served heuristic that always takes locally optimal choices during problem solving [53]. A greedy algorithm may not find a globally optimal solution, however, it is simple to implement, runs quickly, and may still find a good solution.

When deciding placement of DIME trace buffers, the objective of the algorithm is to minimize wirelength. Wirelength is assumed to be directly correlated with Manhattan distance from source to sink. When the MUX of a DIME trace buffer pair is created, it is placed on the closest available non-memory LUT to the source of the net being probed. The DIME SRL is then placed on the closest memory LUT to the MUX. Placement order and priority is determined simply by the order in which nets were requested for debug.

A greedy approach is far from ideal for this application. For a simple example, consider the layout of sources (1 and 2) and leftover LUTs (A and B) in Figure 6.1. With a goal of minimizing Manhattan distance, greedy placement will decide that leftover LUT A, because it is closest, is the best place for a MUX that will be tied to Source 1. Source 2, instrumented after Source 1, will

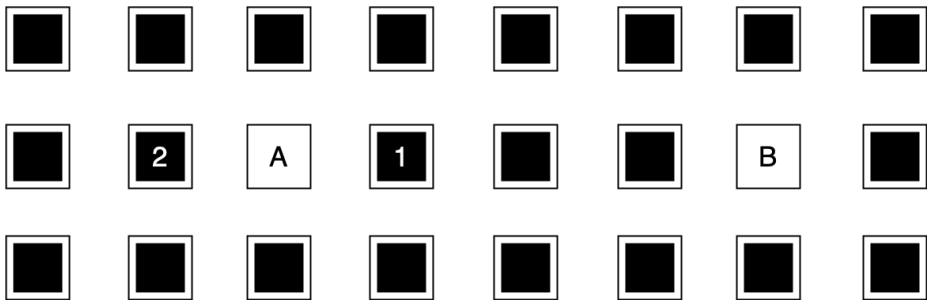


Figure 6.1: Sources to be probed in order (1 and 2) and available LUTs (A and B) that will result in sub-optimal placement using greedy algorithm.

end up being tied to a MUX placed on LUT B. The average Manhattan distance of these routes is (1+5) / 2 = 3. Even though this average would be reduced to (1+3) / 2 = 2 if the configuration was swapped, a greedy algorithm has no way of finding this globally optimal solution.

6.2 Probabilistic Placement with Simulated Annealing

Greedy placement was effective for demonstrating that DIME Debug can be instrumented into large designs. However, as shown, greedy algorithms easily fall short of an optimal solution. In order to further optimize the critical paths in a DIME Debug system, an algorithm that can see beyond local optima is needed.

Simulated annealing is an approach well suited to this problem. Annealing is a probabilistic heuristic that approximates a global optimum with a broad exploration of the search space [54]. The name is taken from annealing principles in metallurgy, which provide a metaphor for the algorithm. A greedy algorithm iterates over the search space only once and takes moves that are optimal at the time they are analyzed. In contrast, a simulated annealing algorithm will iterate over the search space many times at random, attempting swaps. At each step, swaps that improve the solution will be taken. However, moves that weaken the solution also have a chance of being taken. The search begins at a high 'temperature' that lowers over time. The higher the temperature, the higher the chance the algorithm will accept a poor move. Allowing poor moves to be accepted broadens the search space. A poor move may result in a much stronger move later, a stronger move that may not have been discovered in only accepting locally optimal moves.

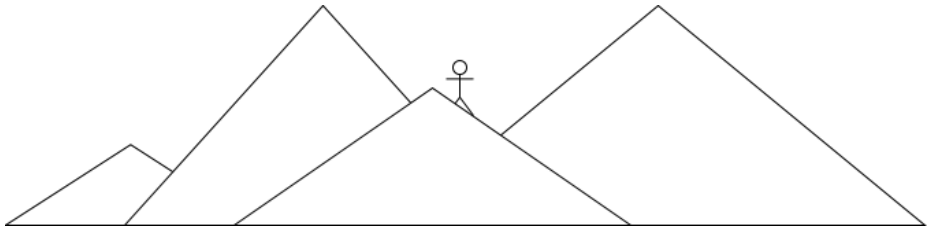


Figure 6.2: The greedy hiker unwilling to search from higher peaks does not see the lower valley on the left side of the mountain range.

A common metaphor for simulated annealing is that of a hiker seeking the lowest valley in a mountain range. A greedy hiker would look around his current location, find the lowest valley in sight, and accept it as the lowest valley in the mountain range (Figure 6.2). A hiker familiar with simulated annealing will recognize that the lowest valley may not currently be in sight. It is necessary to first seek after the highest peaks, the opposite locations of where he would like to end up, in order to see the lowest valleys. The hiker will have the most energy to do this early on in his search. Once he has too little energy to continue the search, he settles on the lowest valley that he has spotted so far. It may not be the lowest valley in the range, but he substantially increased his odds in comparison to the greedy hiker that decided on the first valley he saw.

Now let us put this in the context of FPGA placement with another example using Figure 6.1. The greedy algorithm decides on a placement where Source 1 is tied to LUT A and Source 2 is tied to LUT B. If a simulated annealing algorithm is applied, a solution where Source 1 is tied to LUT B might be explored. This move increases the Manhattan distance of the route from one to three, however, if the algorithm is at high energy, the move may still be taken. LUT A is now available for Source 2. The algorithm has found the optimal solution, resulting in a 50% reduction of average Manhattan distance. This example is somewhat more simplistic than the actual approach used for DIME Debug placement, but adequately demonstrates how simulated annealing can improve proximity between elements.

For DIME Debug, after an initial placement is decided with the greedy algorithm, the following steps will be taken to incorporate simulated annealing.

1. Determine the initial average Manhattan distance between all source-to-MUX and MUX-to-SRL connections.

2. Initialize a reasonable starting temperature.

3. Randomly select either two random SRLs or two random MUXs within the debug system.

4. Determine if swapping the location of these two elements would increase or decrease the average Manhattan distance of the system.

5. If decrease, make swap. If increase, decide whether or not to make swap based on current temperature.

6. Perform predetermined number of repetitions of steps 3-5.

7. Lower temperature.

8. Repeat steps 3-7 until average Manhattan distance has not changed in many iterations.

The initial temperature, number of repetitions, and other algorithm settings that produce the best results are determined through trial and error.

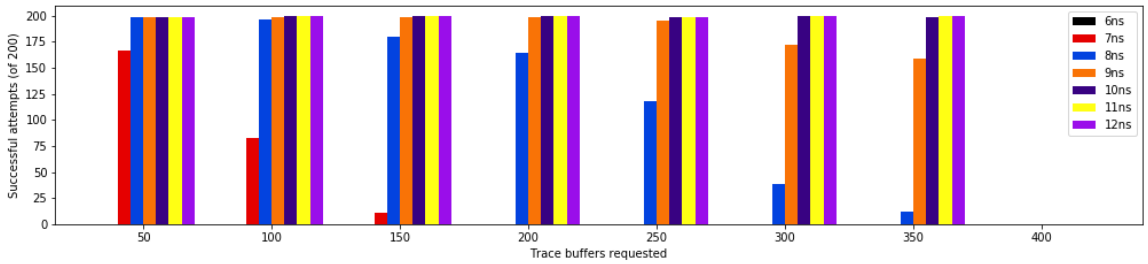
6.2.1 Testing Simulated Annealing

To test the hypothesis that lower average Manhattan distance reduces critical path delays in the circuit, the experiments described in Chapter 4 are repeated with simulated annealing placement incorporated. A sweep of timing constraints and probe counts is tested on all five benchmarks. Each configuration is repeated 200 times while altering the set of nets targeted for debug. Charted bars thus represent an approximate average success rate given each configuration regardless of design nets probed.

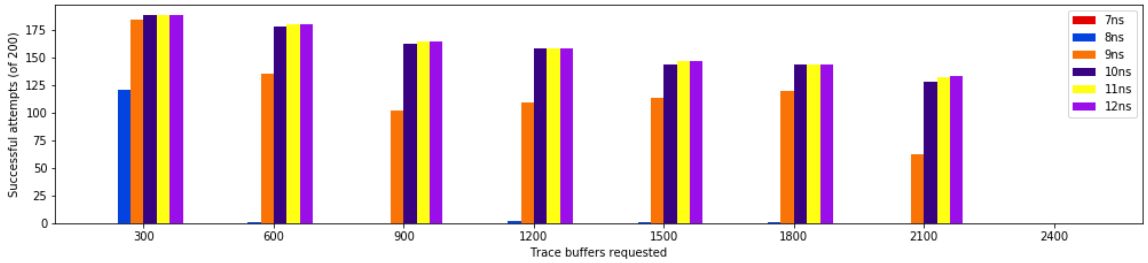
Results

Complete results are charted in Figure 6.3. For ease of comparison, results from a single representative clock period are charted against previous results using the greedy algorithm in Figure 6.4.

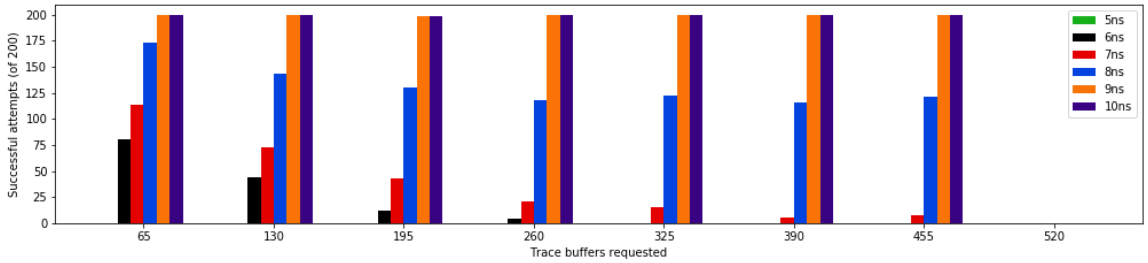
Using the simulated annealing based placement method increases success rates overall for three of five benchmarks. Significantly more probes can be instrumented into the LC3 and RPulseG



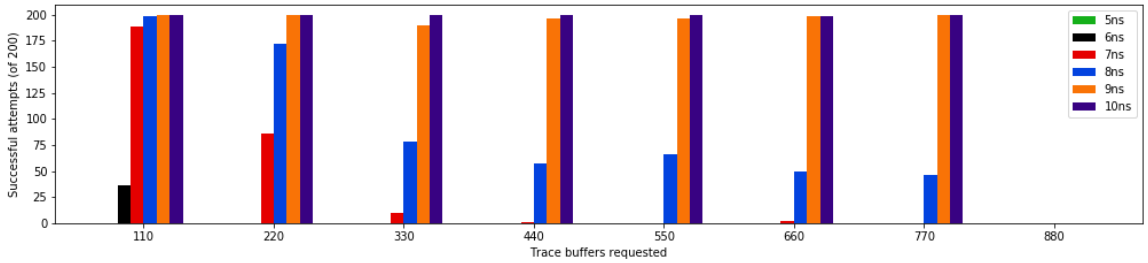
(a) LC3 benchmark with 90% LUT Utilization



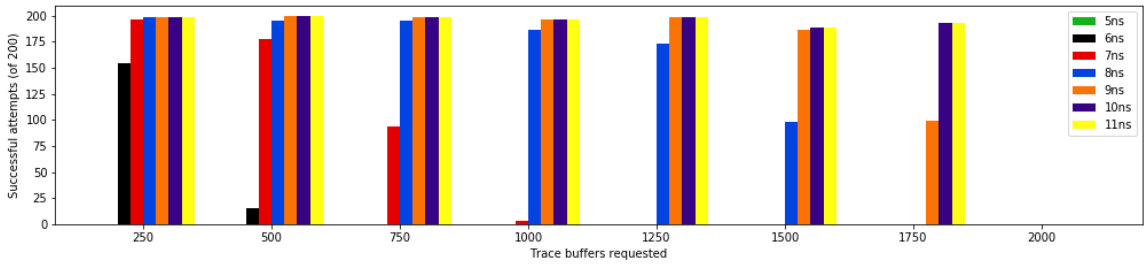
(b) sudoku benchmark with 94% LUT Utilization



(c) RNG benchmark with 90% LUT Utilization

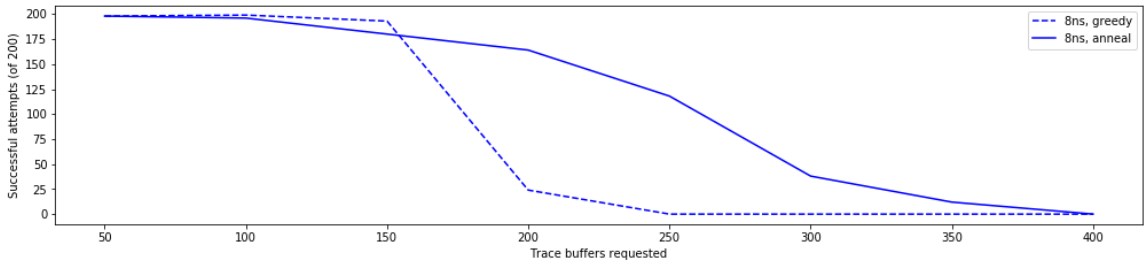


(d) uFIFO benchmark with 90% LUT Utilization

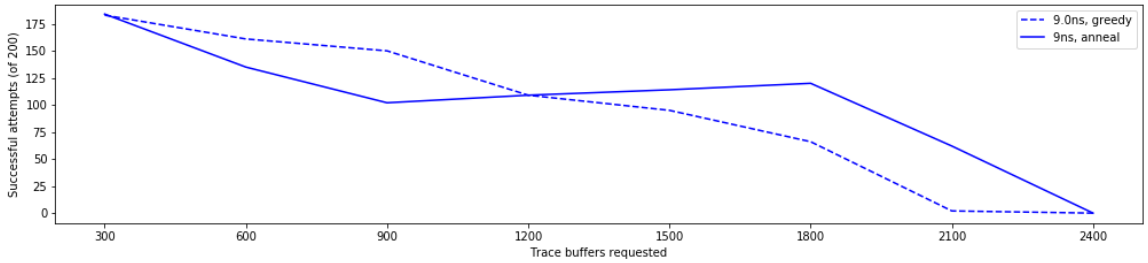


(e) RPulseG benchmark with 90% LUT Utilization

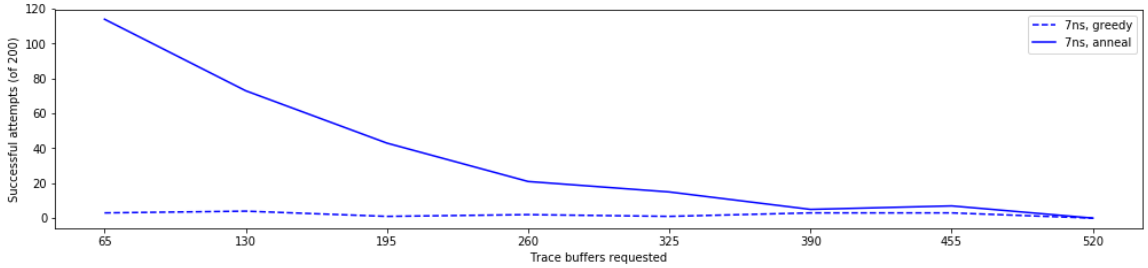
Figure 6.3: Implementation success rates after incorporating simulated annealing placement algorithm into DIME Debug instrumentation



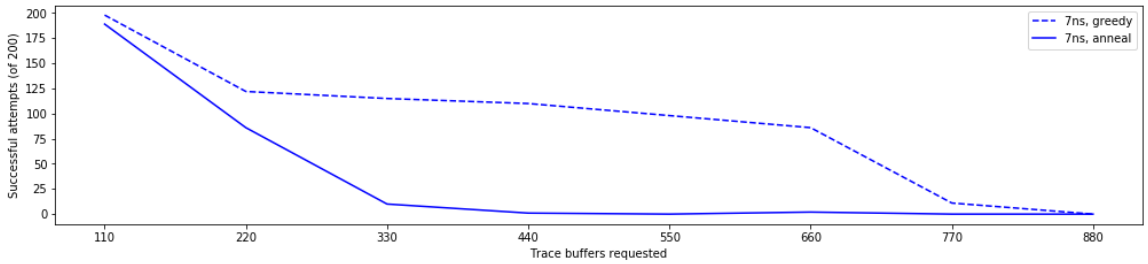
(a) LC3 benchmark with 90% LUT Utilization



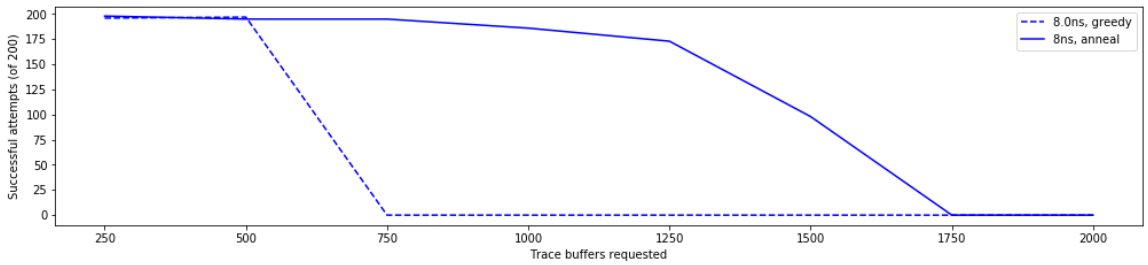
(b) sudoku benchmark with 94% LUT Utilization (note shortened Y axis)



(c) RNG benchmark with 90% LUT Utilization (note shortened Y axis)



(d) uFIFO benchmark with 90% LUT Utilization



(e) RPulseG benchmark with 90% LUT Utilization

Figure 6.4: Implemention success rates of simulated annealing placement compared to greedy placement

benchmarks while maintaining the same clock rate (Figure 6.4, (a) and (e)). Faster clock rates were enabled at low probe counts on the RNG benchmark (Figure 6.4, (c)). The sudoku benchmark experienced only minimal and mixed changes, while the uFIFO benchmark experienced a marked decrease in success rates (Figure 6.4, (b) and (d)).

The mixed results from this experiment are likely due to the various shortcomings of simulated annealing algorithms. Simulated annealing is a probabilistic heuristic that does not guarantee an optimal solution. Even in ideal circumstances it is possible for the algorithm to produce a poor result. While a reasonable effort was put into configuring the algorithm for DIME trace buffer placement, finer tuning of temperature, repetition count, etc. may result in even better results. Parameters used for these experiments were tuned via trial-and-error on the LC3 benchmark. Customizing these parameters for each benchmark individually is likely to provide optimal results and is saved for future work. Investigation into the poor results seen on the uFIFO benchmark, in particular, would be of interest. This benchmark provided unique results in other experiments of this dissertation as well. The characteristics of the uFIFO benchmark were explored in an effort to find the cause of this and other anomalies, however, no strong cause was found and further exploration is saved for future work.

Additionally, the objective function used during simulated annealing could use improvement. The goal of the placer is to minimize average Manhattan distance between debug elements. Even if successful in this objective, timing results may not always improve. While reducing distance between elements has been shown to improve propagation delay, it is not the only contributing factor and this outcome is not guaranteed. In addition, reduction in *average* Manhattan distances may actually result in a solution with a single, lengthy, high-propagation-delay path being left in the solution. Future work with simulated annealing includes an altered objective function that factors in *worst-case* Manhattan distance, which is more likely to have a direct impact on design critical path.

Another noteworthy observation of these results is that simulated annealing did not substantially improve the 6-8ns baseline critical path of DIME Debug. The RNG benchmark did observe approximately 50% of experiments at a 7ns clock period with low probe counts succeed. For other benchmarks, no improvement is seen to this baseline–the fastest clock period reached

is unchanged from the results presented in Chapter 5. Further optimizations will be necessary to eliminate this minimum penalty.

6.3 Preallocating FPGA Resources for DIME Debug

In Chapter 5, a large discrepancy was observed between the number of DIME trace buffers that can be instrumented into a design and the number of remaining unused LUTs on the chip. It was hypothesized that a factor in this discrepancy is unused LUTs residing in locations of the FPGA (sites) already partially used by the user design. The DIME Debug instrumenting tool avoids such locations in an effort to avoid changing the functionality of the user design (this policy was adopted under advisement from Xilinx Labs; see Future Work section of Chapter 9). This section describes a possible alternative method to work around this limitation. Rather than try to place trace buffers in partially used locations, these experiments aim to increase the number of entirely unused LUT locations.

This will be done by preallocating some FPGA resources for DIME trace buffers before the user design is implemented. Design constraints will be used to prevent user logic from occupying these resources. The preallocated sites will then be entirely available for debug logic. The resources that are preallocated will be selected such that DIME Debug trace buffers can easily take advantage of them. In addition, the distribution of these preallocated sites may impact timing closure of the final combined circuit. If preallocated sites can be located such that proximity between user signals and trace buffers is improved, critical path delays may be reduced.

This section will describe the preallocation scheme used and discuss the impact preallocation has on both the user design and the DIME Debug system.

6.3.1 Preallocation Scheme

Three primary factors were considered in deciding how to preallocate FPGA resources for DIME Debug: type, quantity, and location.

Resource Type: A DIME trace buffer requires two LUTs, one of which must be memory capable. Since the non-memory LUT (acting as a 2-to-1 MUX) will tie directly into the memory LUT (the SRL holding probed data), these LUTs would ideally be placed very near one another to

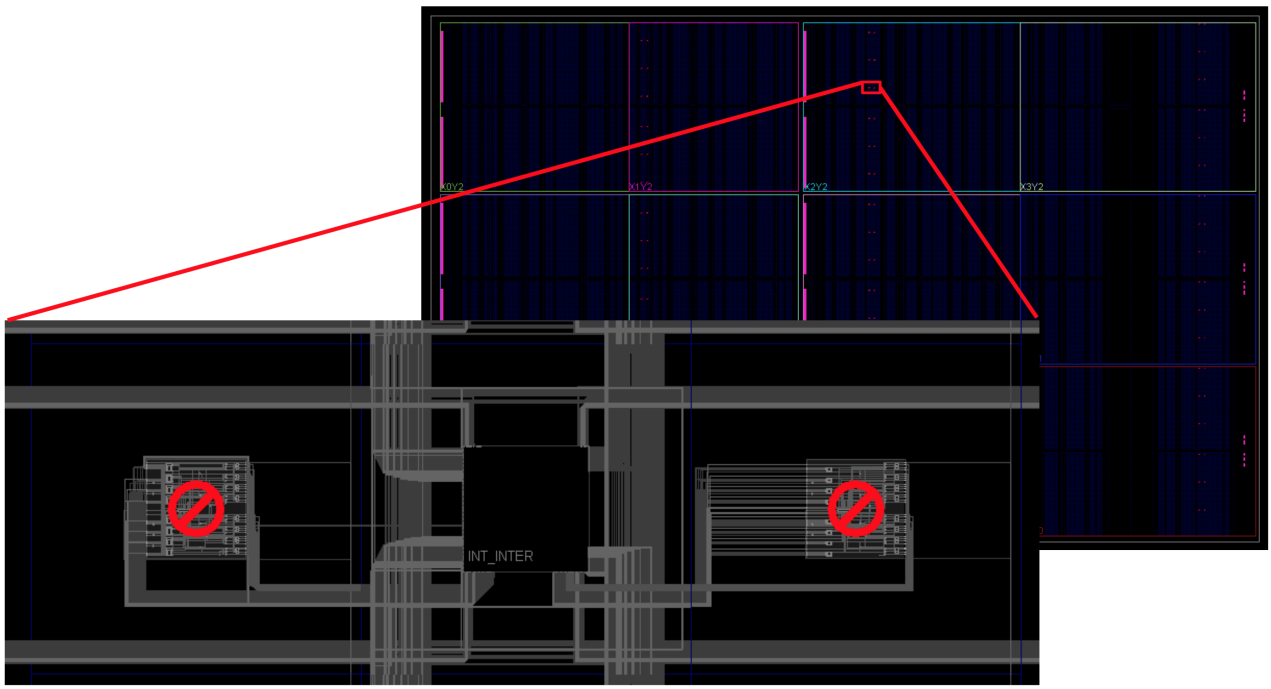


Figure 6.5: A Vivado device view showing preallocated sites (small red dots in upper-right image) and close-up view of adjacent preallocated LUTs.

reduce routing complexity. To this end, the preallocation scheme will always target pairs of LUTs – one memory, one non-memory – adjacent to one another (see detailed view of Figure 6.5). When DIME Debug is being instrumented into the design, the placement algorithm is highly likely to find these adjacent locations and place the LUT duos together whenever possible.

Resource Count: The preallocation scheme will need to be crafted such that it improves the ability to debug while not excessively affecting the engineer's design. Too many preallocated sites will excessively limit the user design, while too few will be unlikely to improve debug. Through some trial and error, it was found that preallocating approximately 1% of all memory LUTs on the FPGA, each paired with an additional adjacent non-memory LUT, strikes a good balance. Preallocating fewer LUTs provided little benefit to debug, while preallocating many more did eventually begin to cause implementation failures of the pre-debug benchmark. It will be demonstrated that, with ~1% of LUTs preallocated, the user design is virtually unaffected and positive results are seen for debug purposes.

Preallocated Locations: Poorly distributed preallocation sites would likely have no positive impact on debug and may increase the chance of negatively affecting the design under test.

In order to create a scheme that has the greatest chance of being universally beneficial, regardless of benchmark or net selection, an evenly patterned distribution is used. Preallocated sites will be evenly spaced apart across the entire FPGA (see zoomed-out view of Figure 6.5). Since nets of interest can be located anywhere on the chip, an even distribution gives all nets an equal chance of having an available LUT relatively nearby to act as a trace buffer.

LUTs are preallocated using the Vivado PROHIBIT constraint. The PROHIBIT constraint is used on physical sites of the FPGA device to disallow Vivado implementation from placing any design logic on that site [55]. A TCL script is used to iterate over all LUTs in the KU025. Every 100th time a pair of adjacent memory- and non-memory LUTs is found, PROHIBIT constraints are created in text form. The resulting list of constraints can be easily saved and placed in the constraints file of the user's Vivado project. While the TCL script is time consuming to execute, the constraints can be re-used each time an engineer wishes to target the same FPGA. Adding these constraints is the only step the engineer will need to take before design implementation in order to implement this preallocation scheme. Vivado will ensure the sites with the constraint are left unused and available to be later found and utilized for debug. The place step within the DIME Debug instrumentation tool already seeks out unused logic for trace buffers and will require no modification to take advantage of the preallocated resources.

Different preallocation schemes, such as higher numbers of preallocated LUTs and different distribution patterns, may further improve results but are left for future work.

6.3.2 Preallocation Impact on Original User Design

The primary drawback of using a preallocation approach is that it places restrictions on the implementation of the user design. Limiting the locations where logic can be placed forces the vendor tool to work around these locations. The logic may be redirected more tightly into fewer sites, causing routing congestion, or spread further across the device, increasing distance between elements. The vendor tool will ensure basic functionality remains the same, but other factors, such as timing, may be affected. In addition, if too many sites are preallocated, it is possible the design will no longer fit on the chip at all. Implementation would fail entirely in this case.

The fastest possible clock period for each benchmark without preallocation in place was found previously, during the experiments in Chapter 5. Here, the preallocation scheme is included

within the constraints of these benchmarks and the test repeated. Timing constraints are tightened until failure, indicating the fastest possible clock period at which the design can be successfully implemented and meet timing. This experiment will show whether or not preallocation affects (1) design timing, due to altered placement, or (2) implementation, due to lack of FPGA resources.

Results

Preallocation did not prevent successful implementation for any benchmark. Enough logic resources remained for the user design despite 1% of LUTs being made unavailable.

The results of the timing closure test are shown in Table 6.1. The clock period change induced by implementing the design with the preallocation scheme is +/- 0.1ns. Even for the fastest baseline clock periods of the RPulseG and RNG benchmarks, this penalty is $<$10%. No change occurred in six of fourteen benchmarks, and two benchmarks successfully implemented a 0.1ns faster clock. This change is so trivial as to possibly be considered noise due to variance in repeated implementation attempts. It may even be feasible to leave preallocation constraints in place on some designs without concern after debug is complete.

Table 6.1: Minimum clock period of benchmarks before and after LUTs are preallocated.

Benchmark	**Original min. period**	**Prealloc min. period**
LC3 70%	4.9ns	5.0ns
LC3 80%	5.2ns	5.2ns
LC3 90%	6.2ns	6.3ns
Sudoku 75%	6.6ns	6.7ns
Sudoku 94%	7.0ns	6.9ns
RNG 70%	1.6ns	1.6ns
RNG 80%	1.6ns	1.6ns
RNG 90%	1.6ns	1.7ns
uFIFO 70%	3.5ns	3.6ns
uFIFO 80%	3.8ns	3.8ns
uFIFO 90%	3.7ns	3.6ns
rpulseg 70%	1.6ns	1.6ns
rpulseg 80%	1.6ns	1.6ns
rpulseg 90%	1.6ns	1.6ns

6.3.3 Preallocation Impact on DIME Debug

The experiments originally described in Chapter 4 are once again repeated, this time with preallocation constraints used during benchmark implementation. A sweep of timing constraints and probe counts is tested on all five benchmarks. Each configuration is repeated 200 times with each repetition targeting a different, random selection of design nets. Bars in each chart thus represent an approximate average success rate given each configuration, regardless of which design nets are being probed. Experiments will utilize the simulated annealing placement algorithm alongside the preallocation scheme (results without simulated annealing placement can be seen in Appendix B).

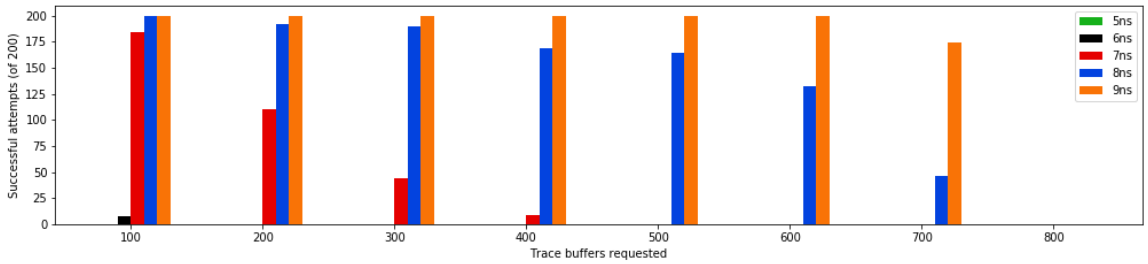
Results

Complete results are charted in Figure 6.6. Note that the x-axis of these charts, representing the number of debug probes requested, is extended in comparison to previous results. For ease of comparison, results from a single representative clock period are charted against results without the preallocation scheme, but using simulated annealing placement, in Figure 6.7.

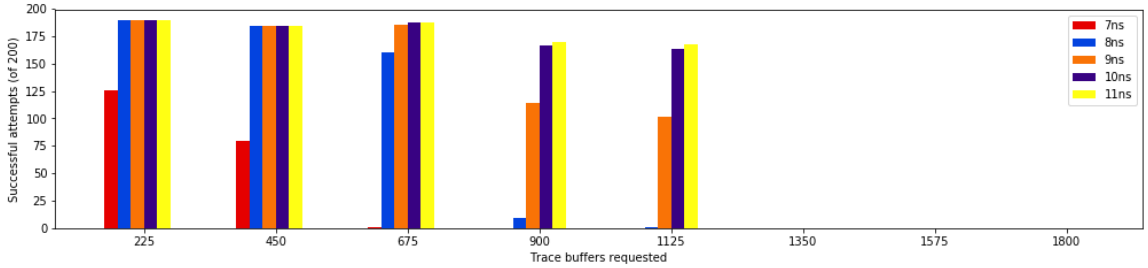
Note also that in these results, different ”representative” clock periods are used between the two experiments in the comparative results. The representative clock period is the fastest period that maintains a reasonably high success rate as maximum probe count is reached. This clock period represents a pseudo-best-fit curve across all configurations for that benchmark. Since preallocation had a significant impact on DIME Debug critical path delays for most benchmarks, a single clock period fails to properly represent both sets of results. For example, for experiments on the LC3 benchmark, an 8ns clock period best represents the results before preallocation, but a 7ns period better represents the results with preallocation used (Figure 6.7, (a)).

Impact on Timing Closure

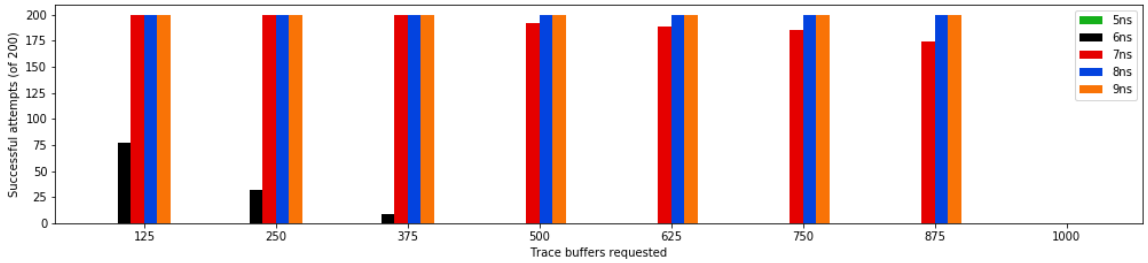
Four of five benchmarks saw higher success rates at faster clock periods with the preallocation scheme in place. Figure 6.7 (a, b, c, f) all show increased success rates across nearly all probe counts with a 1ns faster timing constraint when using preallocation. The uFIFO benchmark saw higher success rates, though at the same clock period (Figure 6.7, (d)).



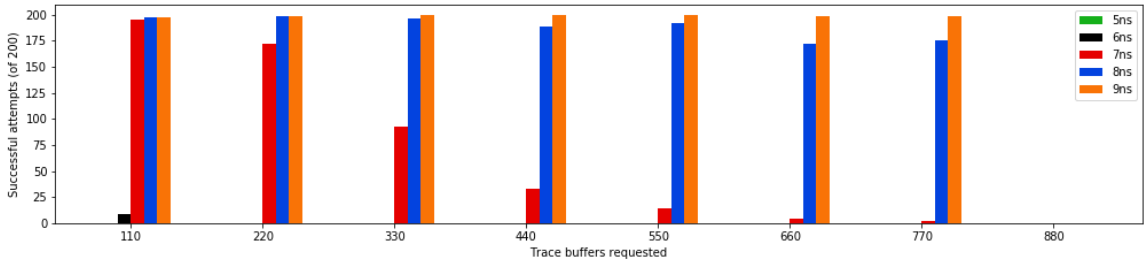
(a) LC3 benchmark with 90% LUT Utilization



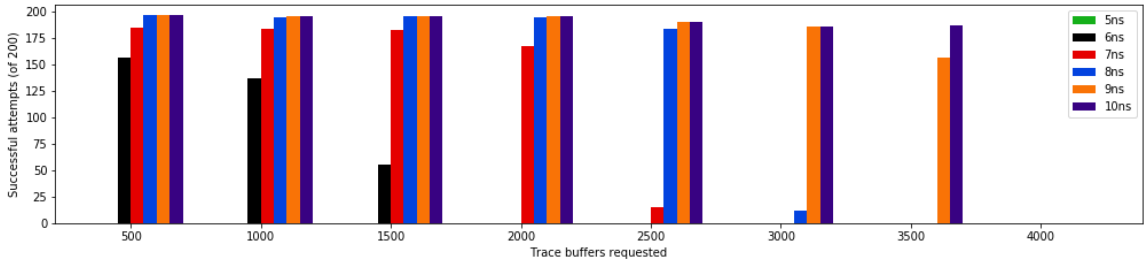
(b) sudoku benchmark with 94% LUT Utilization



(c) RNG benchmark with 90% LUT Utilization

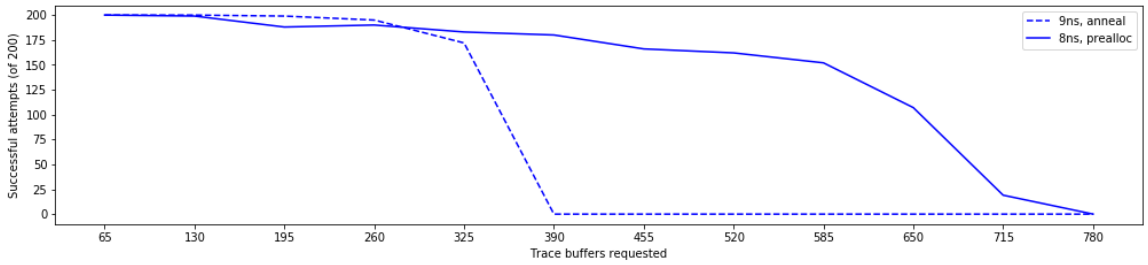


(d) uFIFO benchmark with 90% LUT Utilization

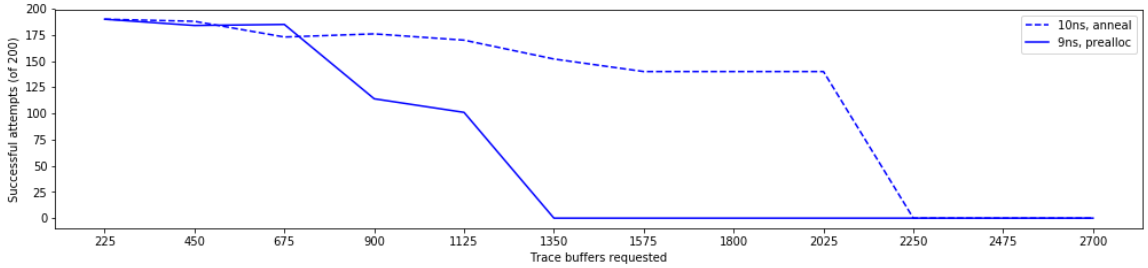


(e) RPulseG benchmark with 90% LUT Utilization

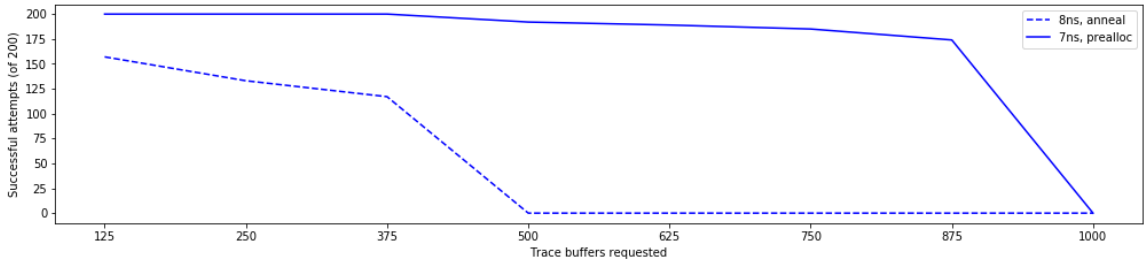
Figure 6.6: Implementation success rates on benchmarks using preallocation scheme design constraints



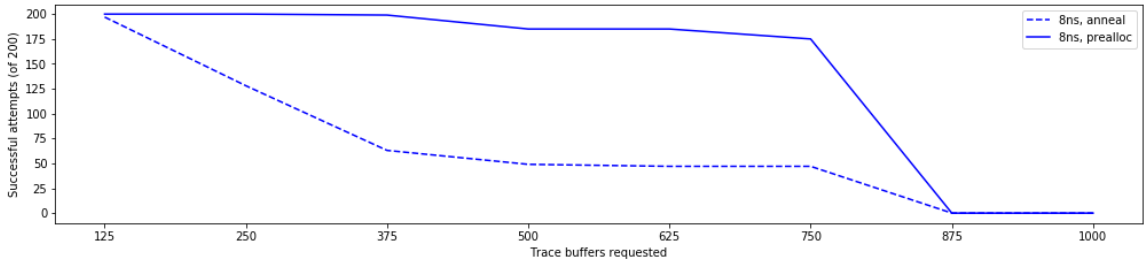
(a) LC3 benchmark with 90% LUT Utilization



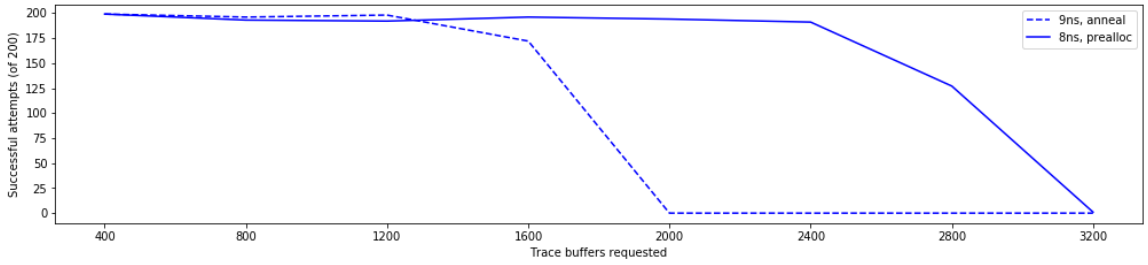
(b) sudoku benchmark with 94% LUT Utilization



(c) RNG benchmark with 90% LUT Utilization



(d) uFIFO benchmark with 90% LUT Utilization



(e) RPulseG benchmark with 90% LUT Utilization

Figure 6.7: Implementation success rates of benchmarks using preallocation compared to simulated annealing alone

Table 6.2: Comparison of minimum clock period of benchmarks before DIME Debug instrumentation, after instrumentation, and after instrumentation using preallocation

Minimum Clock Period, Preallocation			
Benchmark	Uninstrumented	Original DIME	Preallocation DIME
LC3 (90%)	6ns (166.7MHz)	7ns (+17%)	7ns (+17%)
sudoku (94%)	7ns (142.8MHz)	8ns (+14%)	7ns (0%)
RNG (90%)	1.6ns (625MHz)	8ns (+500%)	7ns (+438%)
uFIFO (90%)	3.7ns (270.3MHz)	7ns (+190%)	7ns (+190%)
RPulseG (90%)	1.6ns (625MHz)	6ns (+375%)	6ns (+375%)

Another notable timing result is a small reduction in the baseline critical path penalty on some benchmarks. Prior to using the preallocation scheme, the minimum critical path with DIME Debug instrumented ranged from 6-8ns. The preallocation scheme reduces the highest penalties and brings this range to 6-7ns (Table 6.2). In the case of the sudoku benchmark, this means that a relatively small number of DIME Debug probes ($\leq$450) can sometimes (depending on net selection) be instrumented with no critical path penalty whatsoever.

Impact on Trace Buffer Count

Three of five benchmarks displayed a dramatic improvement in the maximum number of DIME trace buffers that could be instrumented when using preallocation. The LC3 and RPulseG benchmarks (Figure 6.7 (a,e)) reached probe counts roughly 2x higher than their non-preallocated counterpart, while the RNG benchmark (Figure 6.7 (c)) saw nearly triple the previous maximum. The uFIFO benchmark (Figure 6.7 (d)) experienced no change in probe count.

The sudoku benchmark (Figure 6.7 (b)) is the anomaly in terms of probe count. Preallocation roughly halves the maximum number of probes that can be instrumented. In an effort to understand this anomaly, the original reasoning for attempting a preallocation method was checked. DIME trace buffers must be placed on FPGA sites with no user logic. Is it possible that, for some designs, preallocating LUTs could force the design to spread out and actually reduce the number of unused LUTs? To test this hypothesis, each benchmark was run through a simple RapidWright program that counts the number of unused LUTs. These results are in Table 6.3, with notable results in bold. The sudoku benchmark is indeed an outlier–unlike nearly every other benchmark, preallocation actually reduced the number of free LUT sites on the FPGA.

Table 6.3: LUT sites left unused in each benchmark with and without preallocation scheme applied

Benchmark	Original unused sites	Prealloc unused sites
LC3 70%	1589 (8.7%)	1919 (10.6%)
LC3 80%	751 (4.1%)	991 (5.4%)
LC3 90%	129 (0.7%)	381 (2.1%)
sudoku 75%	2588 (14.2%)	2678 (14.7%)
sudoku 94%	**556 (3.1%)**	**338 (1.9%)**
RNG 70%	1925 (10.6%)	2004 (11%)
RNG 80%	305 (1.7%)	381 (2.1%)
RNG 90%	132 (0.7%)	252 (1.4%)
uFIFO 70%	1505 (8.3%)	1605 (8.8%)
uFIFO 80%	192 (1.1%)	223 (1.2%)
uFIFO 90%	217 (1.2%)	245 (1.3%)
rpulseg 70%	**2438 (13.4%)**	**2083 (11.5%)**
rpulseg 80%	1268 (7%)	1315 (7.2%)
rpulseg 90%	561 (3.1%)	960 (5.3%)

6.4 Conclusion

It is important that an embedded logic analyzer carefully avoids excessively altering the design being debugged. If design timing is altered, it may obfuscate the very bugs the logic analyzer is intended to find. One option of improving timing results is to increase proximity of design elements. If elements are closer together, wires between them can be shorter, and propagation delay can be reduced. This chapter has described the three approaches used to increase debug element proximity for DIME Debug.

First, a greedy placement is used that selects local optima for each DIME trace buffer. This is the method used in the first two sets of experiments of this dissertation, when the capabilities of DIME Debug are compared against the Xilinx ILA and the impact of DIME Debug on timing closure is explored. While adequate for demonstrating the ability of DIME Debug to provide observability into extremely large designs, a 6-8ns critical path delay is also introduced into the design when using greedy placement.

Second, greedy placement is improved by following the initial placement with a simulated annealing optimization algorithm. Simulated annealing broadens the search space and increases the chances of finding a global optimum. This placement method was effective in reducing the average Manhattan distance between debug elements when compared to greedy, improving timing

results and increasing DIME Debug implementation success rates for most benchmarks. The imperfect nature of this algorithm did not improve results in every case and the baseline critical path introduced by DIME Debug was mostly unaffected.

Third, a novel preallocation scheme is used. Rather than continue to improve placement with placement algorithms, debug placement is improved with an alteration to the layout of the user design. A small number of LUTs, arranged such that they are well-suited for DIME trace buffers, are set aside for use by debug circuitry. While this raises concerns as to whether or not it would affect the user design, only trivial impact is seen. The improvements to DIME Debug prove to be substantial. Two to three times the number of probes can be instrumented with approximately 1ns faster timing constraints on most benchmarks when compared to results before preallocation is used. The minimum critical path delay of DIME Debug is reduced for some benchmarks. The sudoku benchmark experienced no timing penalty whatsoever in some experiments. Anomalies to these results include the uFIFO benchmark, which saw no improvement in timing results, and the sudoku benchmark, which, despite timing improvement, was only able to instrument approximately half the number of probes with preallocation in place.

CHAPTER 7. ISOLATING LOW-PRIORITY DEBUG PATHS FROM USER DESIGN

Improving trace buffer placement is not the only option for improving critical path penalties incurred by DIME Debug. Some routes within the debug system do not need to be optimized to improve timing results since they never need to operate at the same clock period as the user design. Instead, these routes should be isolated from the timing constraints of the rest of the system. This chapter will review relevant aspects of the DIME Debug system and discuss experiments and results of isolating low-timing-priority debug paths from user timing constraints.

7.1 Review of DIME Debug System

As discussed in Chapter 3, the DIME Debug system operates in two modes: run mode and debug mode. Run mode is relatively simple. A 2-to-1 MUX is programmed to pass signal data from a design net to an SRL trace buffer. Debug mode, which the system enters after a trigger is asserted, is more complex. The 2-to-1 MUX select signals are toggled, chaining every DIME trace buffer in the system into a single long shift-register. Then, upon command from the host, a state machine carefully governs the clock enable signal on each SRL. This state machine is effectively crossing the clock domain from the user clock to the JTAG clock, since the single long shift-register formed by the chain of SRLs terminates at a BSCAN primitive. A BSCAN primitive allows the host to interface with the JTAG system on the KU025. This is how the engineer gathers the observed data.

Note the routes that run from DIME trace buffer to DIME trace buffer (Figure 7.1). These nets are driven by DIME SRLs that are clocked by the user design. Because of this, Vivado considers these routes to be part of the user design–and therefore under the same timing constraint. However, as discussed above, these paths will never operate in sync with the user clock. In debug mode, the state machine is governing the clock enable input on the SRLs such that these paths are being operated at the same speed as the JTAG clock. The JTAG clock on an FPGA is often

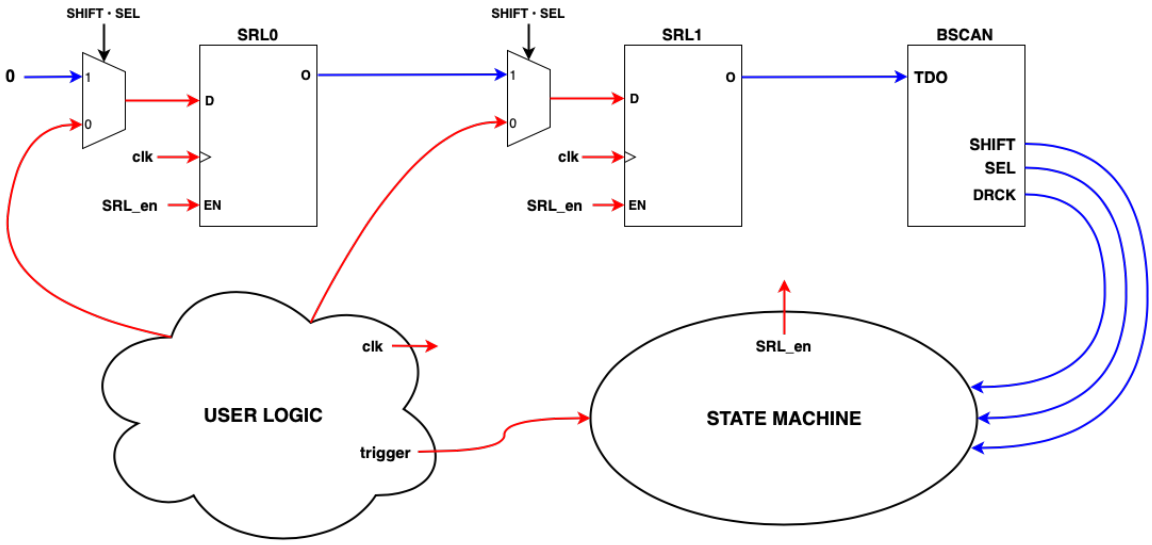


Figure 7.1: Two-buffer DIME Debug System. Red nets must function at user clock rate, but blue nets never will.

several times slower than the user clock. When instrumented with DIME Debug, benchmarks used in these experiments can operate at, in the worst case, 100MHz or faster. The JTAG clock on Kintex Ultrascale FPGAs can operate at a maximum frequency of 66MHz [56], however, the default and Xilinx-recommended frequency when communicating over USB is only 6MHz [57]. This frequency is more than adequate for pulling observed data from the DIME Debug system. In other words, the JTAG clock need never operate more than 1/10th the speed of the user clock in these experiments.

This indicates that buffer-to-buffer routes do not need to be under the same timing constraint as the user design. This is especially of interest because the methods used to improve critical path delays from DIME Debug do not take these routes into account. Neither greedy nor simulated annealing placement consider buffer-to-buffer routes, only paths from user net-to-MUX and MUX-to-SRL. The preallocation scheme is designed to improve those routes, though it offers no improvement for buffer-to-buffer connections. In addition, signals of interest can be located anywhere across the entire user design. Without altering the user design itself, it is impossible to optimize the locations of these nets for debug purposes. The DIME Debug system could be optimized to factor in buffer-to-buffer paths, however, the lower clocking needs do not justify the effort. If buffer-to-buffer paths contain the critical path of a combined circuit, simply relaxing

their timing constraint may allow implementation to successfully complete and meet timing. Since buffer-to-buffer routes are never optimized and debug net selection could place buffers anywhere on the chip, these paths are likely to be long and have high propagation delays.

7.2 Method

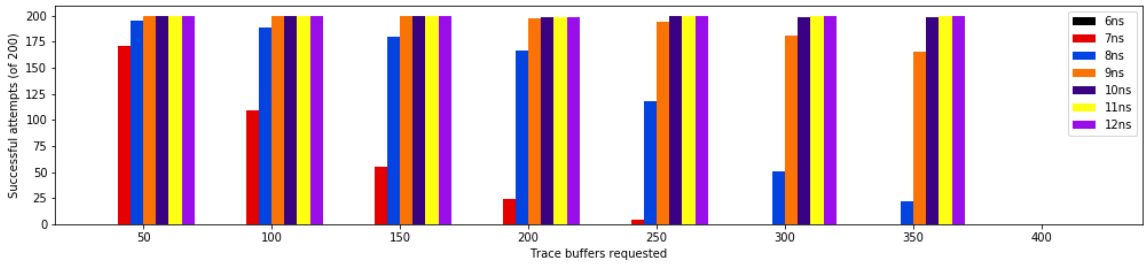
We will effectively isolate the buffer-to-buffer paths from the user clock using Xilinx design constraints. After DIME trace buffers are instrumented, a TCL script is used to finalize routing. This script is used to add constraints to the design that establish the routes between DIME trace buffers as "Multicycle Paths" [55]. By relaxing the hold and setup requirements on the path by one clock cycle, the vendor tool is informed that these routes only need to be able to operate at half the clock speed of the user design. A more customized multicycle path constraint could be formulated based on the clock speed of each benchmark, however, using a generalized half-speed constraint will reasonably demonstrate whether or not buffer-to-buffer routes are becoming the critical path of combined circuits.

To test this hypothesis, previous experiments are repeated with multicycle path constraints in place. For each configuration of timing constraint and probe count, an attempt is made to instrument DIME Debug with 200 random net selections in order to find an approximate average rate of success. Simulated annealing will be used for trace buffer placement (results without simulated annealing placement can be seen in Appendix B).

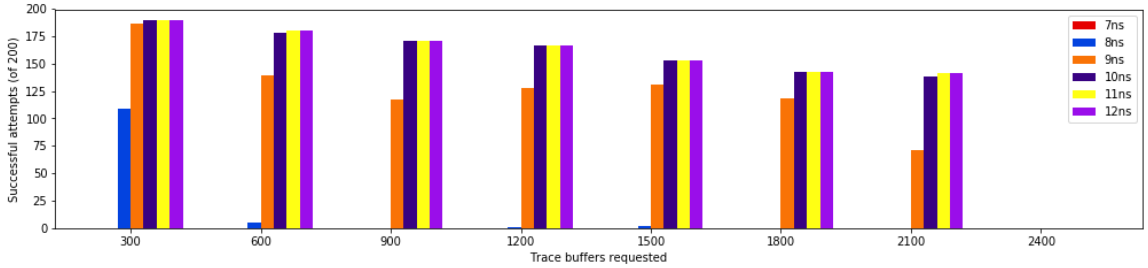
Results

Complete results are charted in Figure 7.2. For ease of comparison, results from a single representative clock period are charted against previous results without multicycle constraints in Figure 7.3.

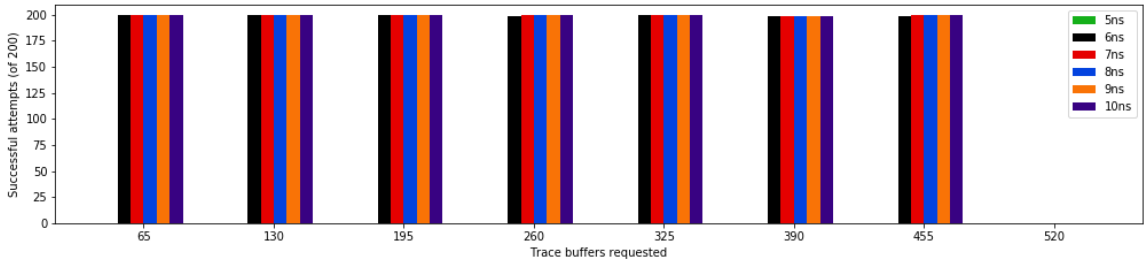
Using multicycle constraints resulted in trivial changes to the success rates on the LC3 and sudoku benchmarks (Figure 7.3, (a,b)). However, these experiments provide a dramatic increase in implementation success on the other three benchmarks. uFIFO and RPulseG benchmarks both see significant increases in success for their fastest clock periods as higher probe counts are requested (Figure 7.3, (d, e)). The most remarkable results came from the RNG benchmark. Originally, the



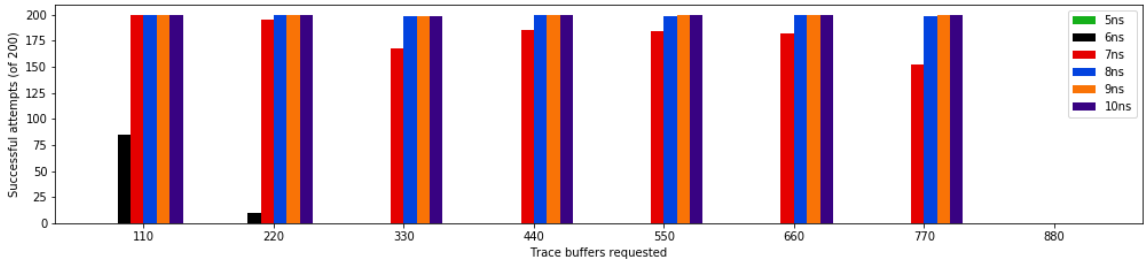
(a) LC3 benchmark with 90% LUT Utilization



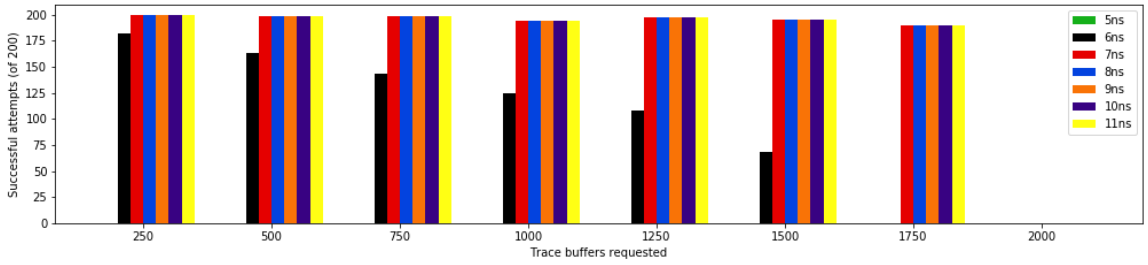
(b) sudoku benchmark with 94% LUT Utilization



(c) RNG benchmark with 90% LUT Utilization

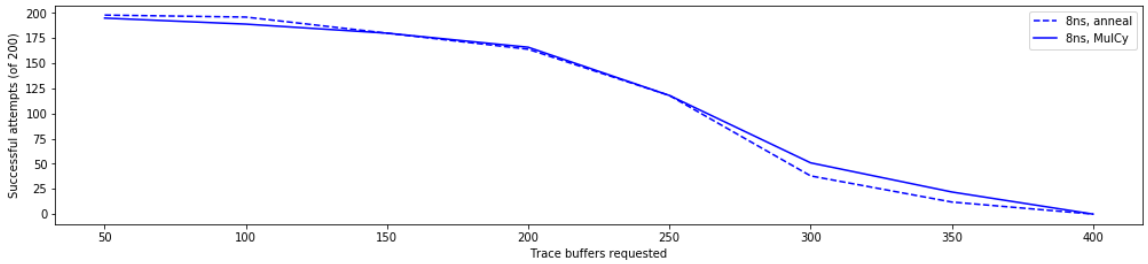


(d) uFIFO benchmark with 90% LUT Utilization

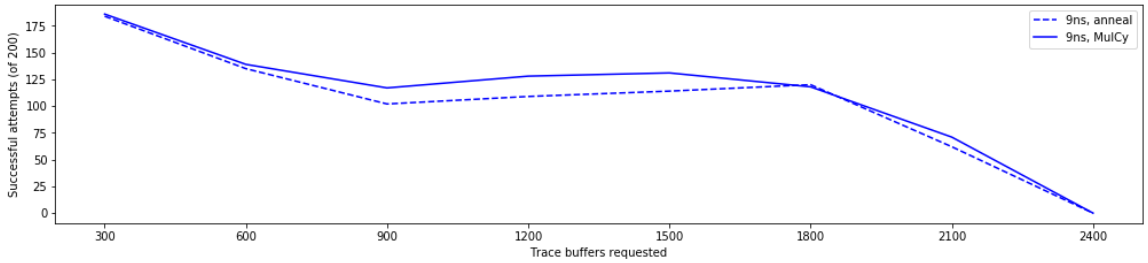


(e) RPulseG benchmark with 90% LUT Utilization

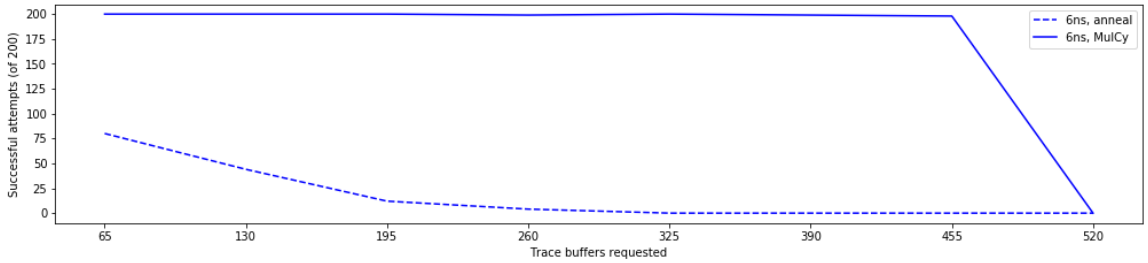
Figure 7.2: Success rates for all five benchmarks with multicycle path constraint on buffer-to-buffer nets.



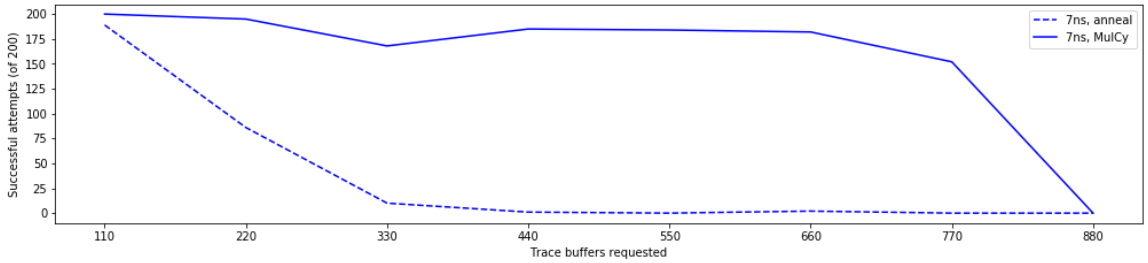
(a) LC3 benchmark with 90% LUT Utilization



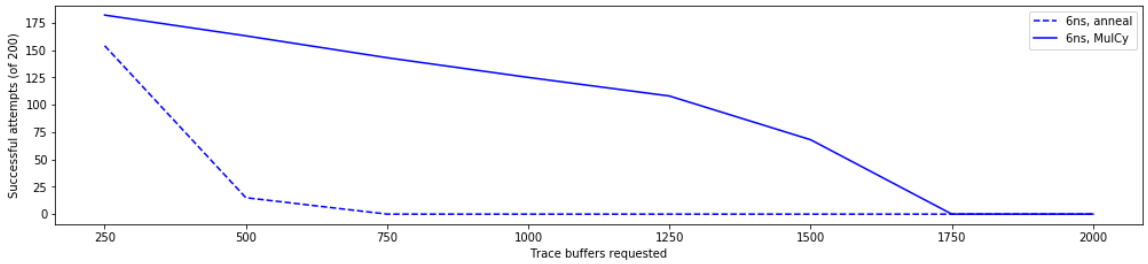
(b) sudoku benchmark with 94% LUT Utilization (note shortened Y axis)



(c) RNG benchmark with 90% LUT Utilization



(d) uFIFO benchmark with 90% LUT Utilization



(e) RPulseG benchmark with 90% LUT Utilization (note shortened Y axis)

Figure 7.3: Results of multicycle path experiments compared to simulated annealing alone.

fastest clock period possible on this benchmark once instrumented with DIME Debug was 8ns. Using multicycle path constraints, this minimum period is not only brought down 25% to 6ns, but perfect success is continually observed at this frequency up until maximum probe counts are reached.

The LC3 and sudoku benchmarks did not experience improvement from multicycle path constraints, but also suffered no drop in success rates. Multicycle path constraints will not improve results if the buffer-to-buffer paths are not the critical paths of the combined circuit. Compared to the other three benchmarks, these two benchmarks have relatively slow minimum clock periods before debug circuitry is instrumented (6-7ns, compared to 1.6-3.7ns of other benchmarks). It is probable that user nets already present in these designs, especially if they have been tied into DIME Debug probes, have an equal or greater impact on design critical path than the buffer-to-buffer routes.

7.3 Conclusion

Multicycle path experiments significantly improved implementation success rates in three of five benchmarks, especially at high probe counts. This indicates that buffer-to-buffer paths do represent one of the prohibiting factors for timing closure in combined DIME Debug circuits. Including multicycle constraints allows the router to recognize that these paths do not need to operate at full user clock speed, enabling successful implementation and timing closure in these cases.

The successful results of these experiments lead to considering other possible locations of critical paths within the DIME Debug system. Paths between user nets and DIME trace buffers and paths between DIME trace buffers have been analyzed and optimized. However, the routes of the DIME Debug control system have not been considered. These paths fan out to each trace buffer within the system, which may result in long routes and high propagation delays. Since the enhancements to DIME Debug presented in this dissertation have only somewhat reduced the minimum propagation delay of DIME Debug, it's entirely possible that these control routes are now the bottleneck. Optimizing these paths for timing or altering their design such that they could be constrained as multicycle paths could further reduce critical paths of the combined user and DIME Debug circuit. This endeavor, however, is saved for future work.

CHAPTER 8. EXTENDING DIME DEBUG TRACE BUFFER DEPTH

DIME Debug trace buffers up until this point have been 16-bits deep, since 16-bit SRLs are easily implemented on a single memory LUT within the FPGA. These tiny buffers have shown that embedded debug can be implemented onto very dense FPGA designs and are adequate for demonstrating various methods of optimizing this debug tool. A 16-bit trace buffer is preferable over no trace buffer at all, as in situations when the commercial ILA was unable to be instrumented into the design. However, 16-bits is a very small amount of data to observe for systems processing millions of bits of data every second. The following experiment aims to increase the length of DIME Debug trace buffers while retaining other benefits of the tool.

This chapter will review relevant background information concerning SRLs, describe the method used to extend trace buffer depth, and provide results of experiments implementing them.

8.1 Anatomy of a Kintex CLB

Each configurable logic block (CLB) on the Kintex KU025 contains eight programmable LUTs. On memory CLB, these LUTs can be implemented as 16- or 32-bit SRLs. However, implementing one of these LUTs as a 32-bit SRL precludes the remaining LUTs on the CLB from being implemented as SRLs at all (Figure 8.1, (a)). This restriction is imposed by Xilinx and cannot be circumvented, even when modifying a design with RapidWright. When DIME Debug was initially created, it was decided to implement one trace buffer per LUT in order to keep each probe as lean as possible. Implementing each trace buffer as a 16-bit SRL allows memory CLBs to host eight trace buffers each, maximizing the number of probes possible within dense FPGA designs (Figure 8.1, (b)).

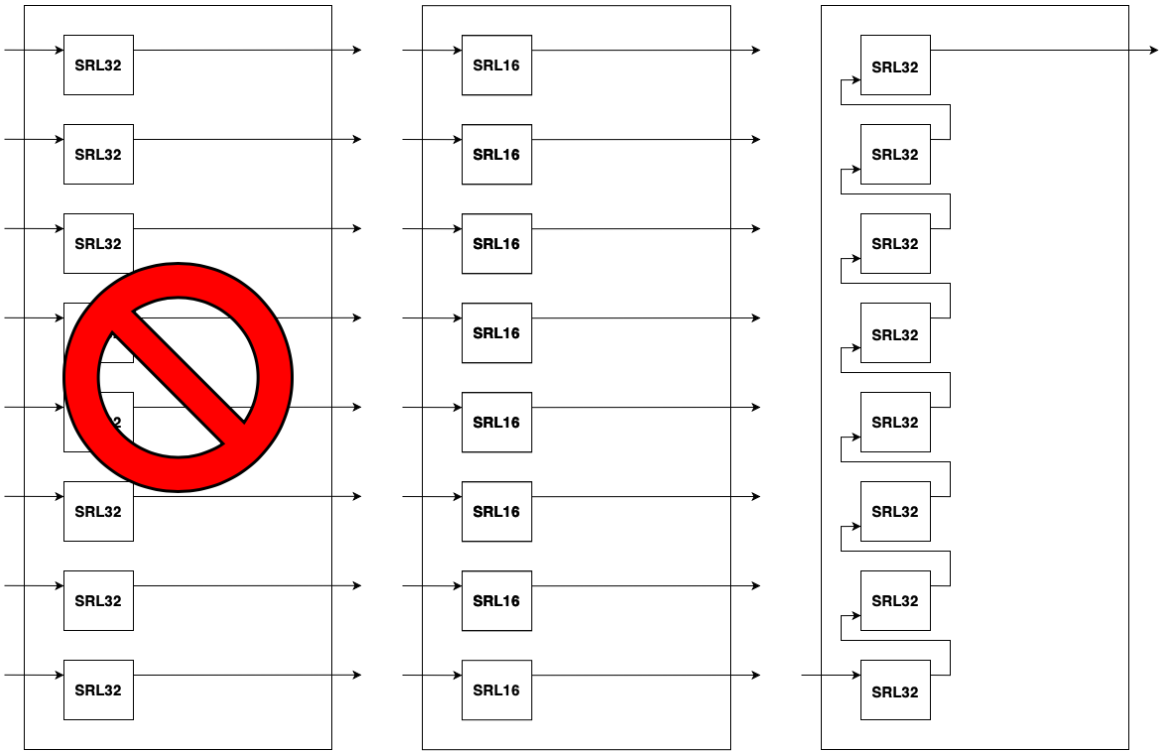


Figure 8.1: Arrangment of 16- or 32-bit SRL on Kintex Ultrascale CLB.

8.2 Method

To utilize more than one LUT on the tile as a 32-bit SRL, they must be chained. This feature is leveraged in order to lengthen DIME trace buffers (Figure 8.1, (c)). Instead of eight 16-bit trace buffers on one tile, a single 256-bit (32x8) trace buffer will be implemented instead, increasing buffer memory by 16x.

To test the feasibility of this method, experiments from prior chapters will be repeated with the RapidWright instrumentation tool modified to place 256-bit trace buffers. As each trace buffer now requires eight memory LUTs, the number of probes requested will be scaled accordingly. All other aspects of the experiments will be the same as previous iterations — implementation will be attempted 200 times, each repetition targeting a randomized net selection for each configuration of benchmark, timing constraint, and probe count. Experiments will also utilize simulated annealing placement, multicycle path constraints, and the preallocation scheme.

8.3 Results

Results of these experiments are charted in Figure 8.2. For ease of comparison, these results are shown again side-by-side to experiments instrumenting 16-bit DIME trace buffers in Figure 8.3. Note the significantly different probe counts attempted (x-axis) between 16-bit and 256-bit results.

Success rates when instrumenting 256-bit DIME trace buffers are nearly identical to rates for 16-bit buffers with approximately a 1:8 probe count ratio. This significant decrease in probe counts is fully expected, since 256-bit buffers require 8x the number of memory LUTs compared to 16-bit. While 256-bit buffers still only require a single non-memory LUT, there are fewer memory LUTs on the FPGA and they are more likely to become the resource bottleneck.

While probe counts are significantly lower, debug memory is significantly higher. Each individual trace buffer can now hold a 256-bit signal history, a 16x increase from single LUT, 16-bit buffers. In addition, the *overall* memory of the entire debug system is approximately doubled. When compared to a 16-bit DIME Debug system, roughly the same number of LUTs are dedicated to trace buffer memory. However, when 256-bit buffers are implemented, each of those LUTs is implemented as a 32-bit SRL, rather than 16-bit, and contains twice the memory.

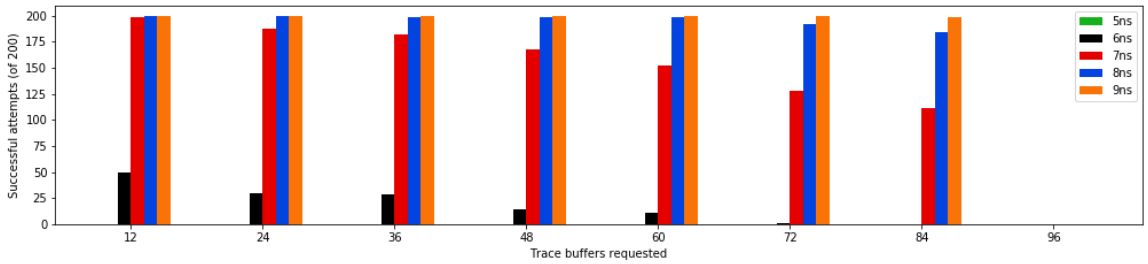
For some benchmarks, timing closure is also improved when using 256-bit buffers. This is most easily seen in results for the LC3 benchmark with a 7ns timing constraint (Figure 8.3 (a) vs (b)) and for the sudoku benchmark with a 7 or 8ns timing constraint (Figure 8.3 (c) vs (d)). The 16-bit and 256-bit results are at nearly an 8:1 probe count ratio, yet higher success rates are seen for these clock periods at higher probe counts. This is likely due to the simple fact that fewer probes mean fewer routes and lower routing congestion. A group of eight LUTs on a single CLB now only requires routing to a single user net, rather than eight. The routes between the eight LUTs of a 256-bit buffer are chain routes within the CLB and have trivial propagation delay.

8.4 Conclusion

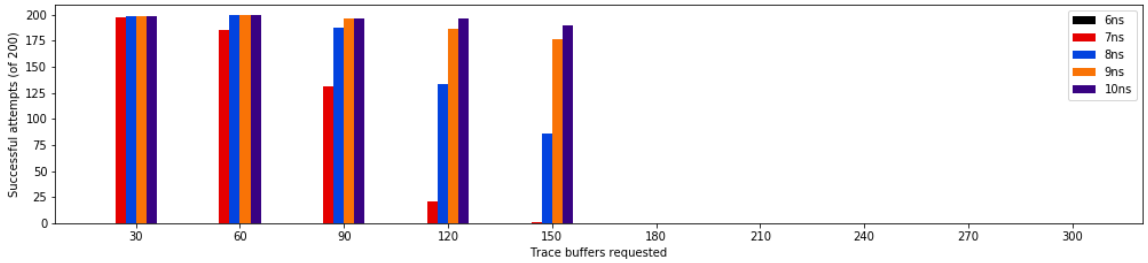
The experiments in this chapter have successfully demonstrated that DIME Debug trace buffers can be deepened from their original 16-bit form to 256-bit, a 16x increase. Approximately 1/8th the number of 256-bit buffers can be instrumented into designs in comparison to 16-bit, since

they require 8x the number of memory LUTs. Fewer probed design nets also result in somewhat lowered timing penalties.

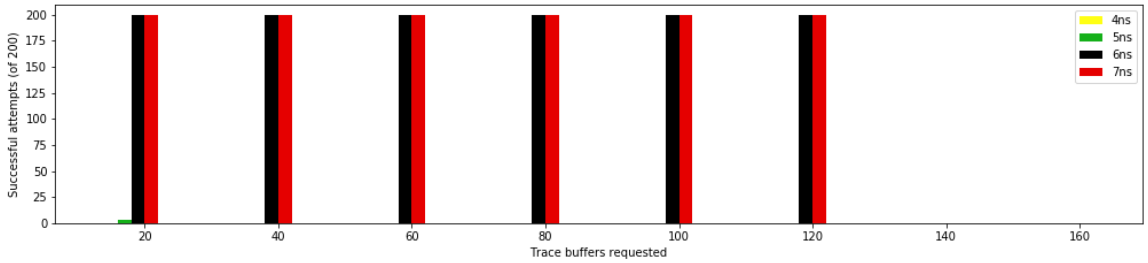
In application, the choice between 16-bit and 256-bit DIME Debug trace buffers would come down to debug needs. An engineer requiring more signals to be probed might favor 16-bit buffers, but one needing longer trace histories would select 256-bit buffers. Future enhancements to DIME Debug could allow the engineer to select which signals will be probed with the shorter or longer buffers. In addition, Xilinx FPGAs include functionality to chain SRLs across CLBs. This feature could be leveraged to increase DIME trace buffer depth beyond 256-bit.



(a) LC3 benchmark with 90% LUT Utilization

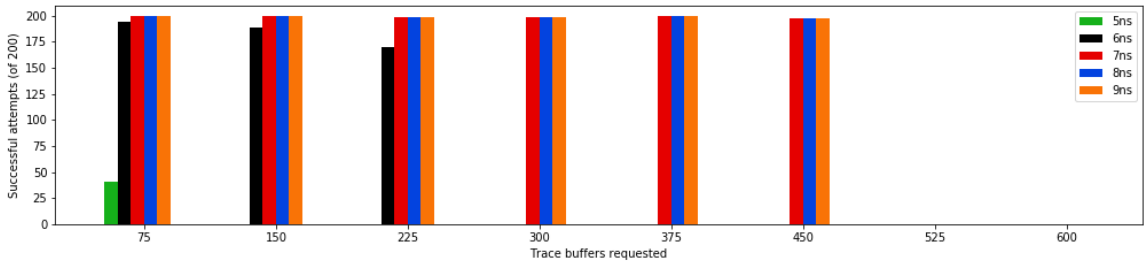


(b) sudoku benchmark with 94% LUT Utilization



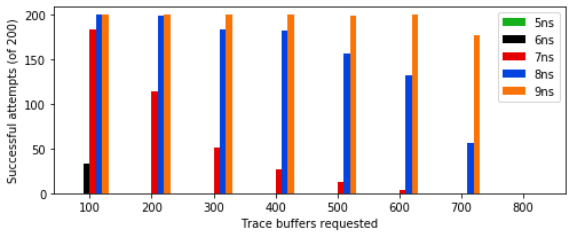
(c) RNG benchmark with 90% LUT Utilization

(d) uFIFO benchmark with 90% LUT Utilization

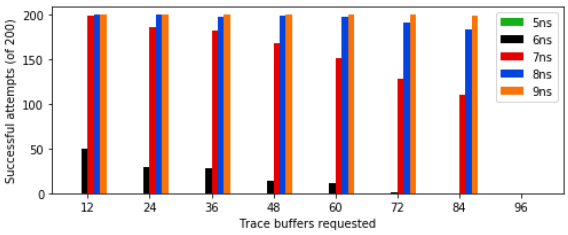


(e) RPulseG benchmark with 90% LUT Utilization

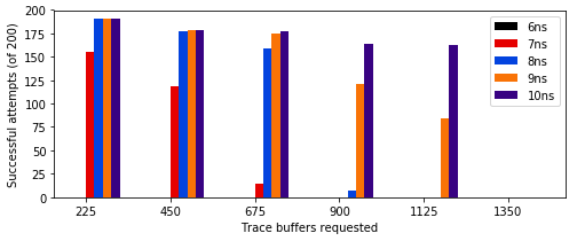
Figure 8.2: Implementation success rates for all five benchmarks implementing 256-bit DIME Debug trace buffers



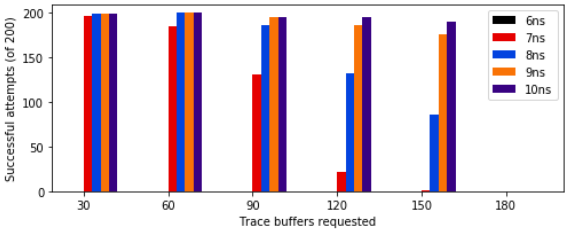
(a) LC3, 90% LUT Util, 16-bit buffers



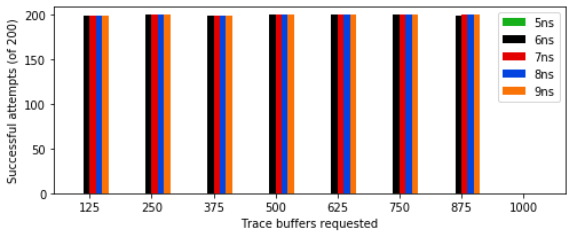
(b) LC3, 90% LUT Util, 256-bit buffers



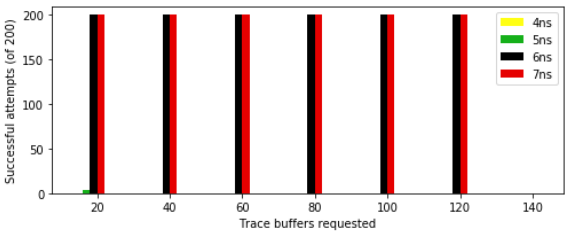
(c) sudoku, 94% LUT Util, 16-bit buffers



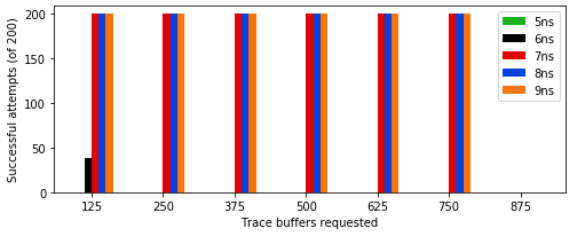
(d) sudoku, 94% LUT Util, 256-bit buffers



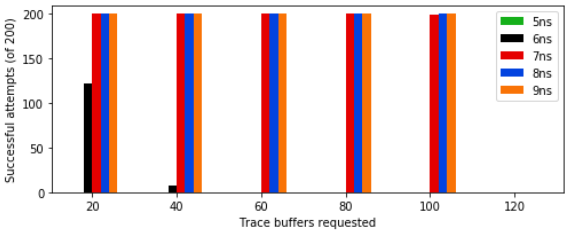
(e) RNG, 90% LUT Util, 16-bit buffers



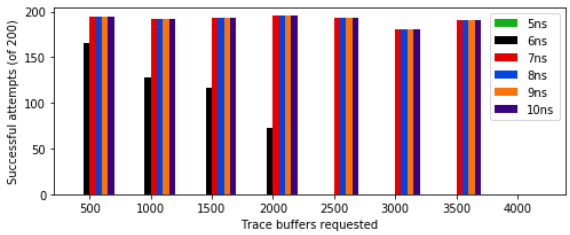
(f) RNG, 90% LUT Util, 256-bit buffers



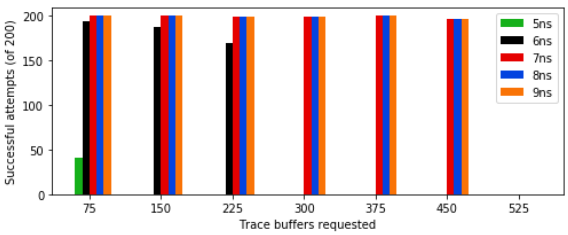
(g) uFIFO, 90% LUT Util, 16-bit buffers



(h) uFIFO, 90% LUT Util, 256-bit buffers



(i) RPulseG, 90% LUT Util, 16-bit buffers



(j) RPulseG, 90% LUT Util, 256-bit buffers

Figure 8.3: Side-by-side comparison of 16-bit and 256-bit trace buffer results

CHAPTER 9. CONCLUSION

This chapter will review the research presented in this dissertation, provide concluding remarks, and suggest several possibilities for future work.

9.1 Summary of Research

Chapters 1 and 2 introduce the motivation and background for this work, namely, FPGAs and the challenges presented in debugging designs built on them. While there are many debug solutions for FPGA projects, they all come with differing limitations. One limitation that had previously been largely unaddressed is instrumenting an embedded logic analyzer into an FPGA design that consumes nearly all of the resources on the chip. In addition, no previous work has attempted to use distributed LUT-based memory for debug trace buffers.

Chapter 3 introduces Distributed Memory (DIME) Debug. Rather than using BRAM, a relatively scarce resource on FPGAs, DIME Debug uses the LUTs spread across the device as small memories for debug trace buffers. Since LUTs are so abundant on an FPGA, it is virtually guaranteed that even large designs will leave some unused. The RapidWright CAD tool is used to scavenge leftover LUTs to instrument small trace buffers into an implemented design. A state machine and BSCAN primitive, tied to the JTAG interface, allow the host to gather debug data once triggering has occurred.

Chapter 4 compares DIME Debug to a commercial logic analyzer, the Xilinx ILA, to test the hypothesis that a distributed memory based embedded logic analyzer can fit into designs when other options cannot. It is shown that, while shorter in terms of signal history, a much higher number of DIME Debug probes can fit into large designs compared to the ILA. For the largest benchmark, where 90% of the device LUTs were already used, a small number of DIME trace buffers could be still be instrumented when the ILA failed entirely. Experiments in this chapter demonstrate that LUT memory trace buffers are capable of enabling embedded debug even in

very large FPGA designs or designs that require all BRAM on the device. It was also shown that leveraging RapidWright for instrumentation reduces debug iteration time by at least 3x.

Chapter 5 investigated the impact DIME Debug will have on the user design once instrumented in terms of timing closure. If a design is forced to operate at a lower clock rate with debug circuitry in place, it is possible that the very bugs being sought may be hidden. Instrumenting DIME Debug into a design introduces minimum critical path delays between 6-8ns. This penalty grows as more and more probes are requested. These critical paths incur a relatively small penalty to the timing closure of benchmarks that were only capable of operating near these speeds to begin with. However, some benchmarks originally operated at much higher frequencies. DIME Debug forces a significant slowdown on these benchmarks (up to 500%). This chapter presents a thorough exploration and quantification of the timing impact that instrumented debug circuitry has on an FPGA design.

Chapter 6 explores various ways of optimizing the placement of DIME Debug trace buffers into the design. First, a greedy algorithm, which favors local optima, is described. This method has been effectively used for buffer placement during experiments in Chapters 4 and 5. Second, the probabilistic simulated annealing optimization algorithm is described. Using simulated annealing in DIME trace buffer placement improves critical path penalties at high probe counts for most benchmarks. Finally, a resource preallocation scheme is introduced as an alternative placement approach. Rather than rely solely on leftover unused LUTs, a small number are set aside before the user design is implemented. While posing almost no penalty to the user design, preallocation allows 2-3x the number of DIME Debug probes to be instrumented at approximately 1ns faster clock rates for most benchmarks. The baseline critical path penalty first seen in Chapter 5 is also lowered by 1ns in some cases.

The experiments in Chapter 6 demonstrate that distributed memory trace buffers can be optimized for placement, and doing so is typically effective in lowering critical paths. Preallocation experiments offer a novel contribution in demonstrating that setting aside a small percentage of CLBs can have near-negligible effect on the user circuit while providing significant gains for distributed-memory debug.

Chapter 7 concerns the routes between DIME Debug trace buffers. These paths will only ever operate at JTAG clock speed, which is significantly slower than the clock period of benchmark

circuits. Therefore, buffer-to-buffer routes do not need to be considered as strictly for timing closure of the combined circuit. The timing requirement for these paths can be loosened with Xilinx multicyle path constraints. Experiments with multicycle path constraints demonstrated that, for benchmarks that originally operated at relatively fast clock frequencies, buffer-to-buffer paths were often becoming the critical path bottleneck of the combined circuit. Timing results improved significantly, especially at high probe counts. The RNG benchmark saw particularly good results that include a decreased minimum clock period from 8 to 6ns.

Finally, in Chapter 8, the depth of DIME Debug trace buffers is increased. Leveraging the ability to chain 32-bit SRL within a single CLB, signal history is expanded from 16- to 256-bits. As each buffer now requires eight memory-LUTs instead of one, the number of probes that can be instrumented is understandably divided by approximately eight. However, these probes hold longer trace histories, and the reduced number of routing resources resulted in some improvement in timing results. While still small buffers in comparison to those implemented on BRAM, a 16x increase in signal history may be helpful for finding design bugs. This chapter successfully demonstrates that DIME Debug trace buffers can be lengthened to 256-bits while retaining most of the benefits of distributed memory trace buffers.

9.2 Concluding Remarks

FPGAs are powerful tools for prototyping, academics, low overhead/low volume production, and a variety of other applications. They are also, however, very challenging to program, and the debug step is a major factor. The signals inside of a hardware device are difficult to view and analyze. Commercially available options provide visibility, but at significant penalties in terms of programmer time and design impact. Academic research has stepped in to provide tools to lower the debug iteration time, area impact, and critical path penalties. The research in this dissertation builds upon existing advances in FPGA debug in several ways.

The primary contribution of this work is exploring the use of distributed LUT memory, rather than traditionally used BRAM, for debug trace buffers. It was hypothesized that distributed memory trace buffers could be used to research possibilities of trace-based debugging that have previously been largely unexplored, such as debug visibility for FPGA designs that are extremely large or require all device BRAM. The Distributed Memory (DIME) Debug tool was created for

this purpose. DIME Debug is used to demonstrate that distributed memory trace buffers can enable embedded debug on very large designs even when commercially available logic analyzers are unable to fit within leftover resources. DIME Debug requires no BRAM whatsoever, allowing the user to use as many of this resource as needed.

Signal visibility is valuable, but only if the integrity of the signal is intact. To prevent obfuscation of bugs, an embedded logic analyzer must be designed such that timing impact on user design is kept reasonably low. When using BRAM for trace buffers, this is done by carefully selecting which design net will be tied into which BRAM and optimizing routing. When using distributed memory trace buffers, a unique opportunity exists to leverage the ubiquitousness of LUTs on the device and instead optimize the *placement* of the buffers. To this end, in addition to using optimization algorithms like simulated annealing, a novel preallocation scheme is developed. The user design is prohibited from occupying a small amount of FPGA logic so it can later be utilized by debug circuitry. These experiments demonstrate not only that preallocation provides significant improvement to DIME Debug, but also that setting aside a small amount of device resources in this manner poses almost no penalty to the original design.

In conclusion, distributed memory is a viable substitute for BRAM trace buffers in situations where BRAM based methods are not feasible for FPGA debug. The DIME Debug tool is a working prototype that demonstrates this capability. In its present form, DIME Debug can be instrumented into designs with the proper logic inserted alongside the user design and some basic scripting within the Vivado Suite. This process could be reduced to a single step with some refinement of the RapidWright instrumentation code. In 15 minutes or less, DIME Debug can provide some signal visibility into fully implemented designs, even those that are extremely large or that consume all BRAM on the FPGA. DIME Debug can also be useful for rapid debug iterations in situations where only a small amount of debug data is needed to quickly identify a problem, regardless of user design size.

DIME Debug could feasibly play a role in the field of FPGA debug in these situations. There are, however, some limitations of DIME Debug to be overcome in future work for this tool to become fully realized. Probably the most significant of these is the lack of a refined triggering mechanism. Routing is also currently completed in Vivado, when several opportunities for improvement could be realized if completed in RapidWright. These limitations, ideas for their

implementation, as well as other possibilities for future research into distributed memory debug are described in detail in the following section. Should future endeavors address some of the most crucial of these shortcomings, DIME Debug could become a very powerful tool in the FPGA field. In situations where a modest trace history is sufficient, DIME Debug could replace commercial logic analyzers by allowing engineers to rapidly debug any size FPGA design with virtually no area overhead and minimal design impact.

9.3 Future Work

Triggering. Triggering is a crucial part of an FPGA debug system. The trigger cues trace buffers when to store signal data. A precise trigger is especially important with DIME Debug since it currently only supports, at maximum, a short 256-bit trace buffer depth. DIME Debug presently uses a signal from the user design wired into the state machine as a trigger. This is a very simplistic method that could be inadequate in many applications. Altering this trigger would require a complete recompilation of the design, since it is part of the user design, slowing debug iterations. Future work would include creating more advanced triggers for DIME Debug. This trigger could be instrumented with RapidWright alongside the trace buffers for fast implementation and control of placement. The potential increased impact on the user design from trigger circuitry would also need to be investigated. As pointed out by Keeley in [39], trigger instrumentation can be more complex than trace buffers. Hung, Eslami, and Wilton have presented relatively simple triggering mechanisms utilizing leftover BRAM or DSP on the chip [40], and even LUTs [37]. Incorporating these methods would provide significantly more advanced triggering functionality and may allow DIME Debug to remain reasonably lean in terms of design impact.

Routing. The current version of DIME Debug utilizes the Xilinx router to complete implementation of debug circuitry. This was a decision made out of convenience, as routing efficiency is not the primary focus of this research. It is entirely possible for the routing step to also be completed using the RapidWright tool. This would open the door to a couple of advantages.

First, shorter debug iteration time. On average, the entirety of DIME Debug instrumentation across all experiments presented in this dissertation, consumes 8.8 minutes, of which 6 minutes (68%) is consumed by Vivado's router. RapidWright is likely to complete routing much faster (as shown with the simple RapidSmith router in [39]), allowing even shorter DIME Debug iterations.

Second, reduced impact on user design. For the experiments in this study, when routing DIME trace buffers, user route rip-up is allowed if necessary. This means that, if needed in order to meet timing closure, the router can alter the routing of the user design when finalizing debug routes. This represents a trade-off that favors lower critical path delay in the combined circuit over preserving the initial placement of user design routes. Either of these effects on the user circuit (longer critical path or altering routes) comes with the possibility of hiding old bugs or creating new bugs. Prohibiting user design route rip-up can be configured into the Vivado route flow. However, more intricate optimizations could be more easily and quickly implemented using RapidWright. The ideal solution would be including additional optimizations to DIME Debug that minimize (or eliminate) timing issues while allowing the user design to remain undisturbed. One notable possibility to achieve this is adding pipelining stages to DIME trace buffer routes. Related work that has implemented this optimization nearly eliminated the critical paths introduced by their debug approach [40]. Incorporating pipelining into DIME Debug may eliminate critical path penalties while maintaining most of the benefit of distributed memory trace buffers.

Optimization of DIME Debug Control System Routes. In this dissertation, routes between user signals and trace buffers, as well as buffer-to-buffer routes, are considered for their propagation delays and how this woul affect the critical path of the entire design. Experiments are conducted to improve this impact. However, DIME Debug control system routes are likely playing a role in the impact DIME Debug has on combined circuit timing closure as well. A clock enable signal must be routed from the state machine to each DIME Debug SRL. BSCAN signals must be tied to both the state machine as well as each SRL. Optimizing for the timing impact of these routes would likely further reduce the timing closure impact from instrumenting DIME Debug into a design.

Additional Benchmarks. The five benchmarks used in this dissertation to display the capabilities of distributed-memory based debug have all been designed specifically to reach certain LUT utilization thresholds on the targeted FPGA device. This is accomplished by taking smaller modules and replicating them enough times until the desired thresholds are reached. As such, these benchmarks contain many repetitive circuits, rather than a single large, comprehensive design. A benchmark large enough to fill an FPGA without replication may provide differing outcomes when instrumented with DIME Debug. For example, a larger and more complex benchmark may

necessitate slower user clock speeds such that the propagation delays of the DIME Debug system become negligible. Replication-based designs were used in this dissertation due to ease of creation in comparison to a design large enough to fill even a majority of the resource rich Kintex KU025.

Alternative Preallocation Schemes. The preallocation experiments presented in this dissertation use a single layout of resources. Approximately 1% of the LUTs on the chip were preallocated in an evenly distributed pattern. This generalized method was effective for improving results on most benchmarks. However, for the sudoku benchmark, part of the results worsened. It is possible that a customized preallocation scheme, created based on the design under test and which nets are being targeted for probing, could further optimize the debug process. Alternative schemes could include different locations and/or amounts of preallocated resources.

Alternative Distributed Memory Trace Buffer Configurations. This dissertation showed that the original 16-bit DIME trace buffers could be extended to 256-bit by chaining all eight LUTs in a Kintex tile and configuring them as 32-bit SRLs. Trace buffer length could be further extended by chaining LUTs across multiple tiles. This would introduce additional routes between sites that could further impact timing closure. Enhancements to DIME Debug could be implemented to allow a user to select the requisite trace length for individual signals.

LUTs on Kintex FPGAs can also be configured as 64-bit non-shifting RAMs. With some added external circuitry to manage data in these RAMs, they could be utilized as trace buffers. An efficiency comparison could be made between 64-bit RAMs with additional logic and 32-bit standalone SRL trace buffers. Since SRLs are exclusive to Xilinx devices, using 64-bit RAM would open the door to distributed memory based embedded debug that could be ported to non-Xilinx FPGAs.

Multiple Clock Domains. This work assumes a design with a single user clock, however, the algorithms could be expanded to account for multiple clock domains–a task difficult to achieve with commercial embedded logic analyzers. SRLs would need to be appropriately clocked based on the clock domain that probes lie in, and one state machine would need to be instrumented for each clock.

LUTs in Partially-Used CLBs. It was initially determined that DIME Debug trace buffers should not be instrumented onto unused LUTs within partially-used CLBs. This decision was made under advisement from Xilinx Labs that using these LUTs would have too large of an impact

on the user design. However, it has since been discovered that it may be possible to leverage these LUTs without significant penalty. Including such LUTs as possible placement locations for DIME Debug trace buffers could allow instrumentation into even denser FPGA designs. Additional routing, timing, and other impacts on the user design would need to be evaluated.

REFERENCES

[1] Hale, R., and Hutchings, B. "Enabling Low Impact, Rapid Debug for Highly Utilized FPGA Designs." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 1

[2] Hale, R., and Hutchings, B. "Distributed-Memory Based FPGA Debug: Design Timing Impact." In *The 2018 International Conference on Field-Programmable Technology (FPT)*. 1

[3] Hale, R., and Hutchings, B., 2019. "Preallocating Resources for Distributed Memory Based FPGA Debug." In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 384–390. 1

[4] The 2018 Wilson Research Group Functional Verification Study `https://blogs.mentor.com/verificationhorizons/blog/2018/11/14/prologue-the-2018-wilson-research-group-functional-verification-study/` Accessed: 2.29.2020. 2

[5] Garcia, P., Bhowmik, D., Stewart, R., Michaelson, G., and Wallace, A., 2019. "Optimized memory allocation and power minimization for FPGA-based image processing." *Journal of Imaging,* **5**(1), p. 7. 3

[6] Dessouky, G., Klaiber, M. J., Bailey, D. G., and Simon, S., 2014. "Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures." In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. 3

[7] Lavin, C., and Kaviani, A., 2018. "RapidWright: Enabling Custom Crafted Implementations for FPGAs." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 133–140. 3, 15, 18, 19, 23

[8] Xilinx Vivado Design Suite - HLx Editions `https://www.xilinx.com/products/design-tools/vivado.html#overview` Accessed: 1.29.2020. 3

[9] Patt, Y. N., and Patel, S., 2000. *Introduction to Computing Systems: From Bits and Gates to C and Beyond.*, 1st ed. Osborne/McGraw-Hill, Berkeley, CA, USA. 7, 27

[10] OpenCores: The reference community for Free and Open Source gateware IP cores `https://opencores.org` Accessed: 1.20.2020. 7

[11] Vivado Design Suite Tutorial, Programming and Debugging `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug936-vivado-tutorial-programming-debugging.pdf` Accessed: 1.20.2020. 9, 13, 19, 27

[12] Quartus II Handbook Volume II https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/qts_qii53009.pdf Accessed: 1.20.2020. 9, 13

[13] Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs https://www.xilinx.com/support/documentation/application_notes/xapp465.pdf Accessed: 2.13.2020. 11

[14] Hung, E., and Wilton, S. J. E., 2011. "Speculative Debug Insertion for FPGAs." In *2011 21st International Conference on Field Programmable Logic and Applications*, pp. 524–531. 11

[15] Wilton, S. J. E., Quinton, B. R., and Hung, E., 2012. "Rapid RTL-based signal ranking for FPGA prototyping." In *2012 International Conference on Field-Programmable Technology*, pp. 1–7. 11

[16] Certus Silicon Debug Datasheet http://s3.mentor.com/public_documents/datasheet/products/fv/certus-ds.pdf Accessed: 1.20.2020. 11, 12

[17] Hung, E., and Wilton, S. J. E., 2013. "Towards Simulator-like Observability for FPGAs: A Virtual Overlay Network for Trace-buffers." In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, ACM, pp. 19–28. 11

[18] Hung, E., and Wilton, S. J. E., 2014. "Accelerating FPGA Debug: Increasing Visibility Using a Runtime Reconfigurable Observation and Triggering Network." *ACM Trans. Des. Autom. Electron. Syst.,* **19**(2), Mar. 11, 14, 15

[19] Kourfali, A., and Stroobandt, D., 2016. "Efficient hardware debugging using parameterized FPGA reconfiguration." In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, IEEE, pp. 277–282. 11

[20] Eslami, F., Hung, E., and Wilton, S. J. E., 2016. "Enabling Effective FPGA Debug using Overlays: Opportunities and Challenges." *CoRR,* **abs/1606.06457**. 12

[21] Coole, J., and Stitt, G., 2015. "Adjustable-Cost Overlays for Runtime Compilation." In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, IEEE Computer Society, pp. 21–24. 12

[22] Eslami, F., and Wilton, S. J. E., 2017. "An Improved Overlay and Mapping Algorithm Supporting Rapid Triggering for FPGA Debug." *SIGARCH Comput. Archit. News,* **44**(4), Jan., pp. 20–25. 12

[23] Eslami, F., and Wilton, S. J. E., 2018. "Rapid Triggering Capability Using an Adaptive Overlay during FPGA Debug." *ACM Trans. Des. Autom. Electron. Syst.,* **23**(6), Dec. 12, 14

[24] Iskander, Y., Patterson, C., and Craven, S., 2014. "High-Level Abstractions and Modular Debugging for FPGA Design Validation." *ACM Trans. Reconfigurable Technol. Syst.,* **7**(1), Feb., pp. 2:1–2:22. 12

[25] Tiwari, A., and Tomko, K. A., 2003. "Scan-Chain Based Watch-Points for Efficient Run-Time Debugging and Verification of FPGA Designs." In *Proceedings of the 2003 Asia and*

South Pacific Design Automation Conference, ASP-DAC 03, Association for Computing Machinery, p. 705711. 12

[26] Wheeler, T., Graham, P., Nelson, B., and Hutchings, B., 2001. *Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 483–492. 12

[27] Iskander, Y. S., Patterson, C. D., and Craven, S. D., 2011. "Improved Abstractions and Turnaround Time for FPGA Design Validation and Debug." In *2011 21st International Conference on Field Programmable Logic and Applications*, pp. 518–523. 12, 13

[28] Shanker, P., 2016. "Spatial Debug & Debug Without Re-programming in FPGAs: On-Chip Debugging in FPGAs." In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, ACM, pp. 3–3. 12

[29] Josephson, D., and Gottlieb, B., 2004. "The crazy mixed up world of silicon debug [IC validation]." In *Proceedings of the IEEE 2004 Custom Integrated Circuits Conference (IEEE Cat. No.04CH37571)*, pp. 665–670. 12

[30] Panjkov, Z., Wasserbauer, A., Ostermann, T., and Hagelauer, R., 2015. "Hybrid FPGA debug approach." In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. 13

[31] ul Hasan Khan, H., and Ghringer, D., 2016. "FPGA debugging by a device start and stop approach." In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6. 13

[32] MacNamee, C., and Heffernan, D., 2000. "Emerging on-ship debugging techniques for real-time embedded systems." *Computing Control Engineering Journal,* **11**(6), Dec, pp. 295–303. 13

[33] Identify RTL Debugger: Simulator-like Visibility into FPGA Hardware Operation `https://www.synopsys.com/implementation-and-signoff/fpga-based-design/identify-rtl-debugger.html` Accessed: 1.20.2020. 13

[34] Graham, P., Nelson, B., and Hutchings, B., 2001. "Instrumenting Bitstreams for Debugging FPGA Circuits." In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 41–50. 13

[35] Hung, E., and Wilton, S. J. E., 2012. "Limitations of incremental signal-tracing for FPGA debug." In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 49–56. 13

[36] Hung, E., and Wilton, S. J. E., 2014. "Incremental Trace-Buffer Insertion for FPGA Debug." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* **22**(4), April, pp. 850–863. 13, 14, 15

[37] Eslami, F., and Wilton, S. J. E., 2014. "Incremental distributed trigger insertion for efficient FPGA debug." In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4. 13, 14, 15, 72

[38] Hung, E., Goeders, J., and Wilton, S. J. E., 2014. *Faster FPGA Debug: Efficiently Coupling Trace Instruments with User Circuitry*. Springer International Publishing, Cham, pp. 73–84. 13, 14

[39] Hutchings, B., and Keeley, J., 2014. "Rapid Post-Map Insertion of Embedded Logic Analyzers for Xilinx FPGAs." In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 72–79. 13, 14, 15, 72

[40] Gaillardon, P., 2015. "Reconfigurable Logic: Architecture, Tools, and Applications." 1 ed. Taylor Francis, ch. 3, pp. 71–96. 13, 14, 15, 72, 73

[41] Hung, E., Jamal, A., and Wilton, S. J. E., 2013. "Maximum flow algorithms for maximum observability during FPGA debug." In *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 20–27. 14, 15

[42] Rose, J., Luu, J., Yu, C. W., Densmore, O., Goeders, J., Somerville, A., Kent, K. B., Jamieson, P., and Anderson, J., 2012. "The VTR project: architecture and CAD for FPGAs from Verilog to routing." In *In ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 77–86. 15

[43] Steiner, N., Wood, A., Shojaei, H., Couch, J., Athanas, P., and French, M., 2011. "Torc: Towards an open-source tool flow." pp. 41–44. 15

[44] Xilinx Virtex-6 Family Overview `https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf` Accessed: 1.24.2020. 15

[45] Baeckler, G., 2016. "HyperPipelining of High-Speed Interface Logic." In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA 16, Association for Computing Machinery, p. 2. 15, 39

[46] Lach, J., Mangione-Smith, W. H., and Potkonjak, M., 2000. "Efficient Error Detection, Localization, and Correction for FPGA-based Debugging." In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, ACM, pp. 207–212. 15

[47] Xilinx UltraScale Architecture and Product Data Sheet: Overview `https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf` Accessed: 1.24.2020. 16, 27

[48] UltraScale Architecture Configurable Logic Block User Guide `https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf` Accessed: 2.15.2020. 16

[49] UltraScale Architecture Configuration User Guide `https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf` Accessed: 2.15.2020. 18

[50] Mentor ModelSim `https://www.mentor.com/products/fv/modelsim/` Accessed: 4.7.2020. 23

[51] RapidWright Documentation `https://www.rapidwright.io/docs/` Accessed: 3.7.2020. 24

[52] Brigham Young University Office of Research Computing https://rc.byu.edu/about Accessed: 2.17.2020. 29

[53] Black, P. E., 2005. greedy algorithm https://xlinux.nist.gov/dads//HTML/greedyalgo.html Accessed: 2.21.2020. 39

[54] Skiena, S. S., 2008. *The Algorithm Design Manual*., 2nd ed. Springer Publishing Company, Incorporated. 40

[55] Vivado Design Suite User Guide: Using Constraints https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug903-vivado-using-constraints.pdf Accessed: 2.25.2020. 48, 58

[56] Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics https://www.xilinx.com/support/documentation/data_sheets/ds892-kintex-ultrascale-data-sheet.pdf Accessed: 2.25.2020. 57

[57] Vivado Design Suite User Guide: Programming and Debugging https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug908-vivado-programming-debugging.pdf Accessed: 2.25.2020. 57

APPENDIX A. INSTRUMENTING DIME DEBUG CIRCUITRY WITH RAPIDWRIGHT

This appendix will provide in-depth detail about the process of instrumenting FPGA circuitry with the RapidWright backend CAD tool. Common RapidWright objects will be referred to with capitalization, such as a Tile or Design.

A.1 Devices, Designs, Netlists, and RapidWright

This section will describe a few important aspects concerning how the FPGA and the logic to be programmed upon it is represented in RapidWright. The three layers to be aware of are the Device, the Design, and the Netlist (or Logical Design).

Device. A RapidWright Device is the representation of the FPGA itself, independent from the Design (below) that will be programmed onto it. The Device is aware of the physical layout of the FPGA, such as coordinates of individual tiles that contain programmable fabric.

Design. A RapidWright Design represents the physical part of the circuit that is to be programmed onto the FPGA. For example, the Device will contain Site objects, and the Design will contain SiteInstance objects. That is, a portion of the design that has been implemented and placed onto a particular Site. The Design object is typically the base of any RapidWright project. A Xilinx Design Checkpoint (DCP) is converted directly into a Design and the entirety of the FPGA design is contained within it. The Design is aware of both the Device and the Netlist (below).

Netlist/Logical Design. A Netlist, or EDIFNetlist, represents the logical aspect of the FPGA design. For example, the programming of a LUT to be either an SRL or a 2-to-1 MUX is determined at the logical level in the Netlist. The majority of design manipulations must happen at both the physical (Design) and logical (EDIFNetlist) level. The EDIF portion of the name is based on the Electronic Data Interchange Format file that Xilinx uses to represent their netlists.

A.2 Instrumenting DIME Circuitry

This section will go over the specifics of how the DIME Debug tool is instrumented into the FPGA design using RapidWright.

A.2.1 Import Design

The first step is to draw all of the design data into RapidWright with an object in the Design class. This is done in a single line of code:

```
1    Design design = CheckpointTools.readCheckpoint(args[2], args[3]);
```

Note that command line argument [2] is the original DCP, and command line argument [3] is an EDIF file drawn from Vivado alongside the DCP. At the point in RapidWright's development when this tool was written, the EDIF file needed to be imported alongside the DCP since the EDIF included within the DCP is encrypted.

One specific piece of information that is needed from the design before any instrumentation can complete is the list of design Nets that are going to be probed. For most of the experiments in this dissertation, this was done by determining the total number of nets in the design and selecting 200 of them at random (ensuring nets within the debug system or clocking system are not selected). However, for application purposes, this would be done by finding all nets that are marked for debug. The getNetsMarkedForDebug function within the RapidWright ILAInserter class handles this task by checking both the properties of each EDIFNet and the design constraints:

```
2    /**
3     * This method will examine a design for any nets marked for debug
4     * and return the list of names of those nets.
5     * @param design The design to examine
6     * @return A list of net names marked for debug.
7     */
8    public static List<String> getNetsMarkedForDebug(Design design){
9        // Nets can be marked for debug as a netlist property
10       ArrayList<String> debugNets = new ArrayList<>();
```

```
11          for(Entry<String,EDIFNet> e : design.getNetlistNetMap().
                entrySet()){
12              EDIFPropertyValue p = e.getValue().getProperty(
                    EDIF_MARK_DEBUG);
13              if(p == null) continue;
14              String etv = p.getValue();
15              if(etv.equals(ETV_true) || etv.equals(ETV_TRUE)){
16                  debugNets.add(e.getKey());
17              }
18          }
19
20          // Nets can also be marked for debug in XDC
21          ArrayList<ArrayList<String>> xdcLines = new ArrayList<ArrayList
                <String>>();
22          if(design.getEarlyXDCConstraints() != null) xdcLines.add(design
                .getEarlyXDCConstraints());
23          if(design.getXDCConstraints() != null) xdcLines.add(design.
                getXDCConstraints());
24          if(design.getLateXDCConstraints() != null) xdcLines.add(design.
                getLateXDCConstraints());
25          for(ArrayList<String> file : xdcLines){
26              for(String line : file){
27                  if(line.equals("") || line.startsWith("#")) continue;
28                  if(line.contains(XDC_MARK_DEBUG) && line.contains(
                        XDC_SET_PROPERTY)){
29                      String[] tokens = line.split("\\s+");
30                      if(tokens[0].equals(XDC_SET_PROPERTY) && tokens[1].
                            equals(XDC_MARK_DEBUG) && tokens[2].equals("true
                            ") && tokens[3].equals("[get_nets")){
31                          String netName = tokens[4].substring(tokens[4].
                                indexOf('{')+1, tokens[4].indexOf('}'));
32                          debugNets.add(netName);
```

```
                        }
                    }
                }
            }
            return debugNets;
    }
```

When finding a random set of nets for debug, the Net itself is referenced in an ArrayList called netsToProbeNets.

A.2.2 Create Trace Buffers

Armed with the complete Design and a list of Nets needing probing, the Cells that will represent the two parts of DIME Debug trace buffers can be created. As mentioned previously, many design manipulating operations in RapidWright will need to be done in two steps, physical (Design) and logical (EDIFNetlist).

Logical Cell Creation

For the logical side, an EDIFCell must be created. This will contain the information concerning what sort of cell is being created and how it is to be programmed. For the MUX, a LUT3 type cell is needed, and we want to program it as a 2-to-1 MUX:

```
    EDIFNetlist myEdif = design.getNetlist();

    // See if the LUT3 type exists. If not, create and add it.
    EDIFCell newECell = myEdif.getCell("LUT3");
    if(newECell == null){
        newECell = new EDIFCell(myEdif.getLibrary("hdi_primitives"),"
            LUT3");
        // Now add all the ports.
        newECell.addPort(new EDIFPort("O",EDIFDirection.OUTPUT,1));
        newECell.addPort(new EDIFPort("I0",EDIFDirection.INPUT,1));
```

```
48          newECell.addPort(new EDIFPort("I1",EDIFDirection.INPUT,1));
49          newECell.addPort(new EDIFPort("I2",EDIFDirection.INPUT,1));
50      }
51
52      // Now we have the "type," create our new EdifCellInstance
53      EDIFCellInstance newECI = new EDIFCellInstance(lutName, newECell,
            myEdif.getTopCell());
54
55      // Give the cell the appropriate properties.
56      // This is the part that will actually make the LUT implement the
            LUT equation I want (in this case, a 2 to 1 mux with I2 as the
            select line, I1 as the value when I2 is 1, I0 as the value when
            I2 is 0)
57      // This results in LUT equation O = I0 & !I2 + I1 & I2
58      newECI.addProperty("INIT","8'hCA", EDIFValueType.STRING);
```

A similar process creates the SRL, but an "SRL16E" type of EDIFCell, with the appropriate set of properties, is used instead.

Physical Cell Creation

Following logical cell creation, the physical cell must be created. A Cell will be created, referencing the EDIFCell, and all of the logical ports will be associated with physical Pins:

```
59      Cell newLUT3 = new Cell();
60      newLUT3.setType("LUT3");
61      newLUT3.setName(lutName);
62      newLUT3.setEDIFCellInstance(newECI);    // The ECI created earlier
            with edif tools
63      newLUT3.addPinMapping("A1", "I0");  // Hard coded pin mappings.
64      newLUT3.addPinMapping("A2", "I1");
65      newLUT3.addPinMapping("A3", "I2");
66      newLUT3.addPinMapping("O6", "O");
```

```
67      newLUT3.setLocked(false);
```

This process is repeated for the SRL Cell with the appropriate Pin mappings.

Greedy Placement

After the MUX Cell and SRL Cell are created they are immediately assigned an initial placement. A location is sought out that is physically close as possible to the source of the debug net the Cell will be tied to. For the MUX, this is the source of the net that is being probed. For the SRL, this is the MUX. Beginning at the tile of the source, a search pattern spirals outward until a suitable, unused location is found. At this point in instrumentation, the Cells are added to several hashmaps and arrays to maintain a record of their creation and initial placement. Their creation and placement will be finalized and inserted into the Design after a final placement is decided on with the simulated annealing algorithm.

A.2.3 Simulated Annealing Based Placement

At this point the Cells representing all DIME trace buffer MUXs and SRLs have been created and are organized into both hashmaps (that affiliate each Cell with the preliminary Site it is to be placed on) and parallel arrays. The next step is to optimize the placement of these Cells by using simulated annealing. Additional arrays are created that contain all *possible* locations that an SRL or MUX could be placed on, respectively. Possible locations include all unused LUTs as well as LUTs that have an SRL or MUX preemptively placed on them. Memory and non-memory LUTs are distinguished for SRLs and MUXs. These arrays are randomly indexed to select an SRL or MUX that will attempt a location swap. Then the arrays of possible new locations are randomly indexed and the outcome of the swap is determined, based on the new Manhattan distance and the current temperature of the simulated annealing algorithm. If swaps are accepted, the hashmaps affiliating Cells with Sites are updated. The implementation of simulated annealing is already described in Chapter 6.

A.2.4 Finalize Placement

Now that placement has been conclusively decided, the SRL and MUX Cells must be inserted into the Design. The hashmap of Sites that will be occupied by trace buffer Cells is iterated over. A SiteInstance, based on the Site, is instantiated and included in the Design. Each Cell is assigned to one of the eight LUTs of the Site (represented as ElementType) and added to the SiteInstance, bringing the Cell into the Design as well:

```
68      // Iterate over every Site that we will be using, adding the cells
           to it, and adding them both to the design.
69      for(Entry<Site, ArrayList<Cell>> siteEntry : myUsedSites.entrySet()
           ) {
70          Site thisSite = siteEntry.getKey();
71          ArrayList<Cell> cells = siteEntry.getValue();
72          // Make a new SiteInstance (creation will also add it to the
               design)
73          SiteInstance thisInstance = new SiteInstance(thisSite.getName()
               +"_instance", design, thisSite.getTypeEnum(), thisSite);
74
75          // Select a LUT element for each cell.
76          int index = 0;
77          for(ElementType ETO : thisSite.getSiteType().getElements()){
78              // Ensure it is an element that works (a 6LUT)
79              if(ETO.getName().contains("6LUT")){
80                  // Get the next cell in line.
81                  Cell thisCell = cells.get(index);
82                  // Give the cell the element.
83                  thisCell.setElement(ETO);
84                  // Add the cell to the instance (which will add it to
                       the design)
85                  thisInstance.addCell(thisCell);
86
87                  index++;
```

```
88                  // If we have assigned all the cells of this Site an
                       element, break out of the element loop.
89                  if(index>=cells.size()) {
90                      break;
91                  }
92              }
93          }   // End element loop
94      }
```

Since placement is a purely physical operation, this step does not require an additional logical counterpart.

A.2.5 Stitch Trace Buffers

Now that all trace buffer SRL and MUX Cells have been created and their placement finalized, we must tell the Design which Nets should be tied to which Pins on those Cells. This step must be done at both the physical and logical level. A helper function is used called connectPin that, when given all of the necessary parameters, locates the net at both levels and ties it into the correct design pins:

```
95  public static void connectPin(String net_name, String phys_pin, String
       logic_pin, Cell cell, SiteInstance site_ins, EDIFCellInstance
       e_cell_ins, EDIFCell e_cell, Design design, boolean create_net,
       boolean pin_is_source) {
96
97      //
98      // Physical - Connect Pins to Cells
99      //
100     Pin newPin = new Pin(pin_is_source, phys_pin, site_ins);
101     Net newNet = design.getNet(net_name);
102
103     if(create_net){
104         newNet = new Net(net_name);
```

```
105             design.addNet(newNet);
106         }
107         newNet.addPin(newPin, true, true);
108 
109         //
110         // Logical - Connect EdifPortRefs
111         //
112         EDIFPort newEPort = e_cell.getPort(logic_pin);
113         EDIFNet newENet = newNet.getLogicalNet();   // Get equivalent
                logical net
114 
115         if(create_net){
116             newENet = new EDIFNet(net_name, e_cell_ins.getParentCell());
117         }
118         newENet.addPortRef(new EDIFPortRef(newEPort, newENet, -1,
                e_cell_ins));
119 
120 }
```

Note that this does *not* route these nets. This 'stitching' step only informs the Design, both logically and physically, which Cells must be tied into any given net. Which specific physical routes on the FPGA these nets will be placed on is decided later by the router in Vivado.

A.2.6 Export

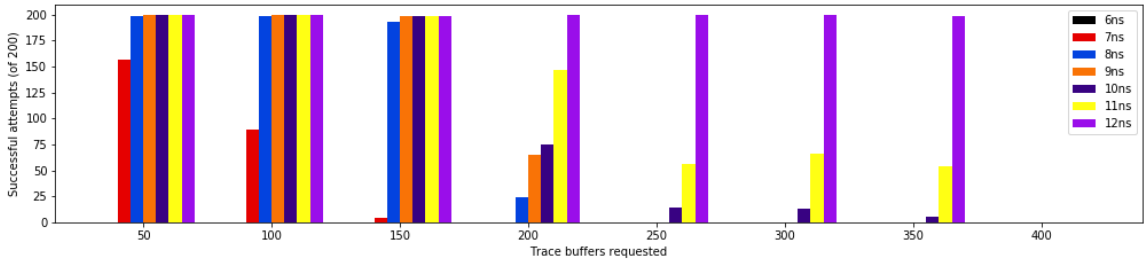
At this point, the Cells that make up each DIME Debug trace buffer have been created, given finalized placement, and assigned to be connected to signals. All of this information has been included within the Design. The Design can now be pushed back into DCP format in order to be exported back to Vivado:

```
121 CheckpointTools.writeCheckpoint(design, output_name+".dcp");
```

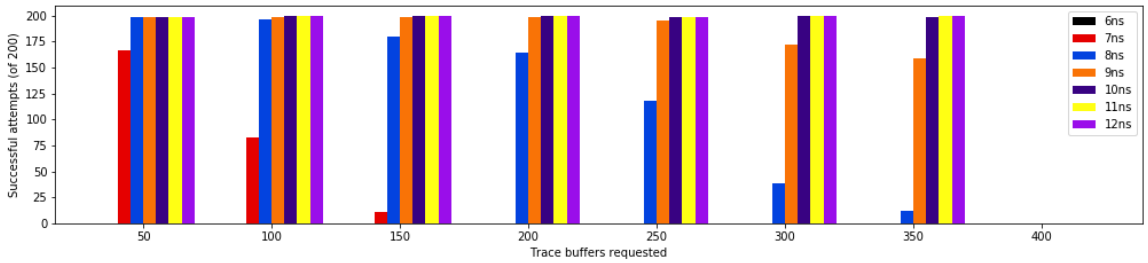
A new EDIF is not created as it will be included with the DCP, unencrypted, with this command.

APPENDIX B. IMPLEMENTATION SUCCESS RATES

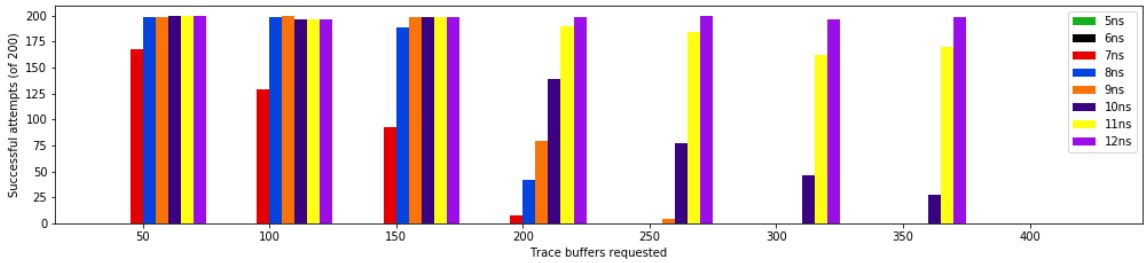
The implementation success rates documented throughout this dissertation typically built on one another, with each successive experiment also utilizing the enhancements from previous experiments. In this appendix, the implementation success results of each experiment *without* prior experiments in place are given. These results show how enhancements to DIME Debug impact each benchmark on their own. Note that all of these experiments implement 16-bit trace buffers and utilize the greedy algorithm for placement.



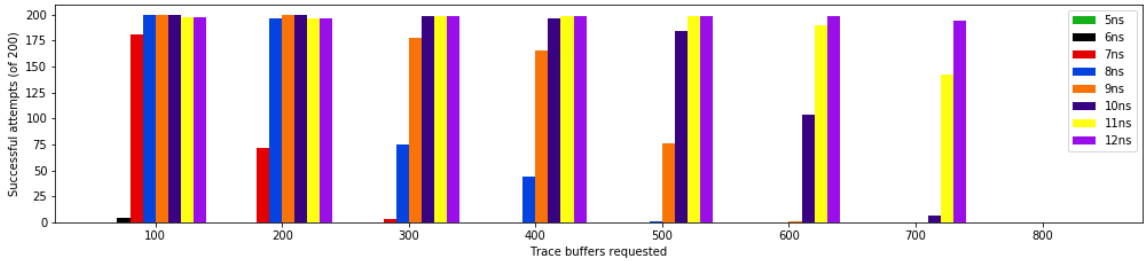
(a) No Optimizations



(b) Simulated Annealing Placement

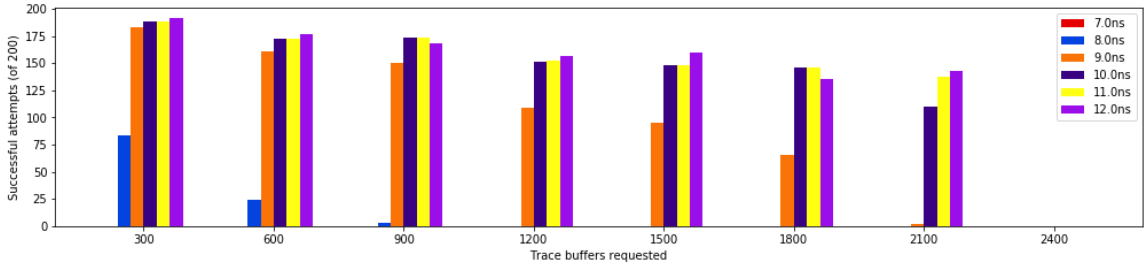


(c) Multicycle Path Constraints on Buffer-to-Buffer Paths

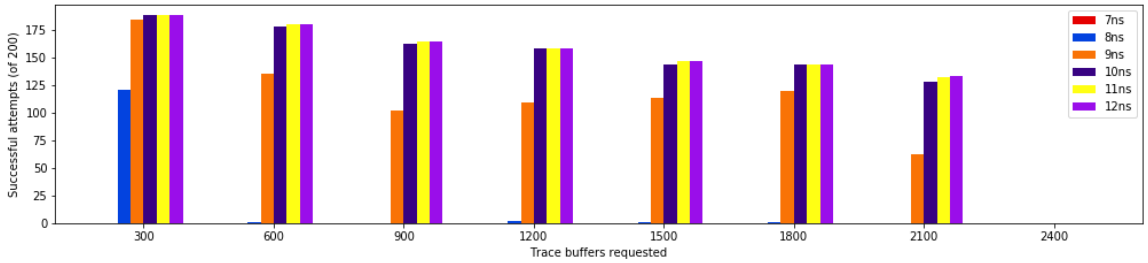


(d) 1% of LUTs Preallocated for Debug (note change to probe counts on x-axis)

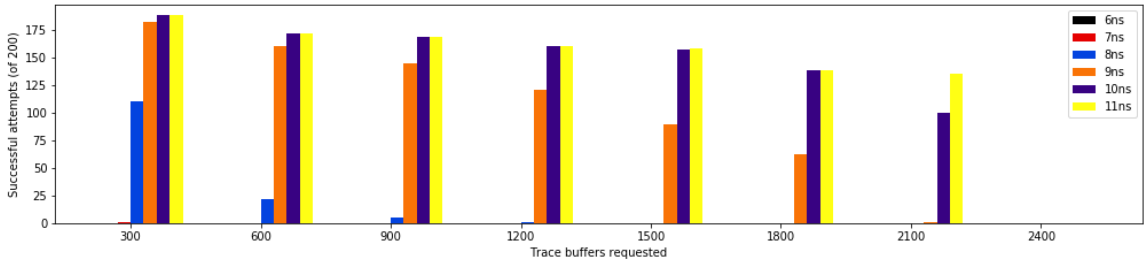
Figure B.1: Implementation success rates for 90% utilized LC3 benchmark with various enhancements implemented one-by-one.



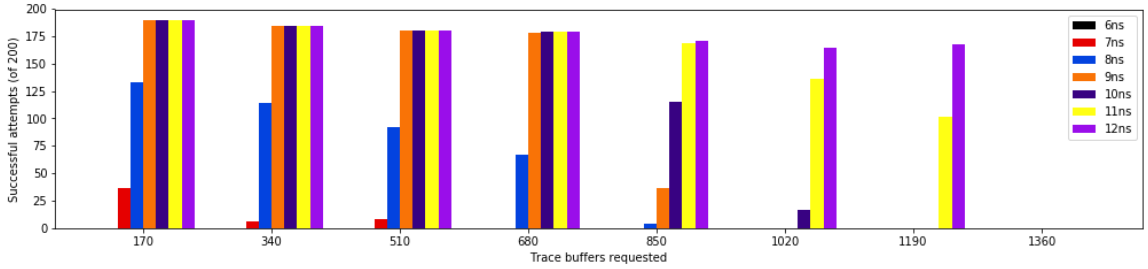
(a) No Optimizations



(b) Simulated Annealing Placement

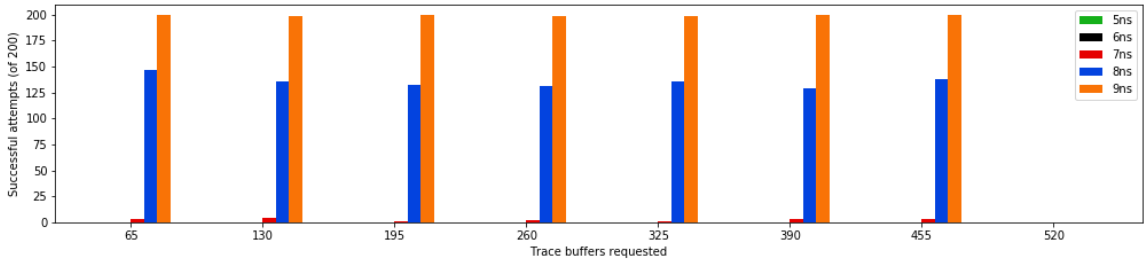


(c) Multicycle Path Constraints on Buffer-to-Buffer Paths

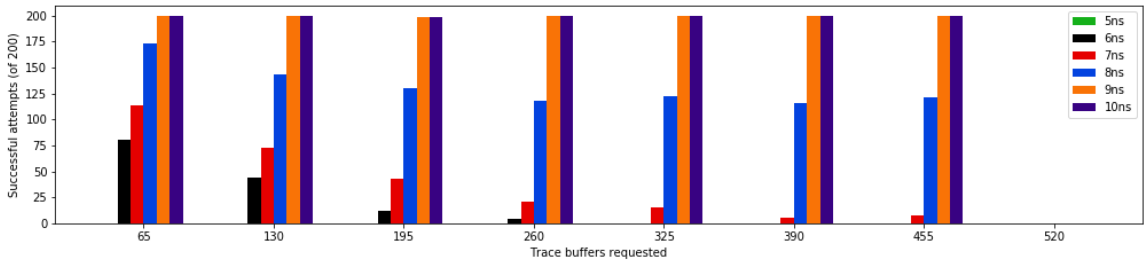


(d) 1% of LUTs Preallocated for Debug (note change to probe counts on x-axis)

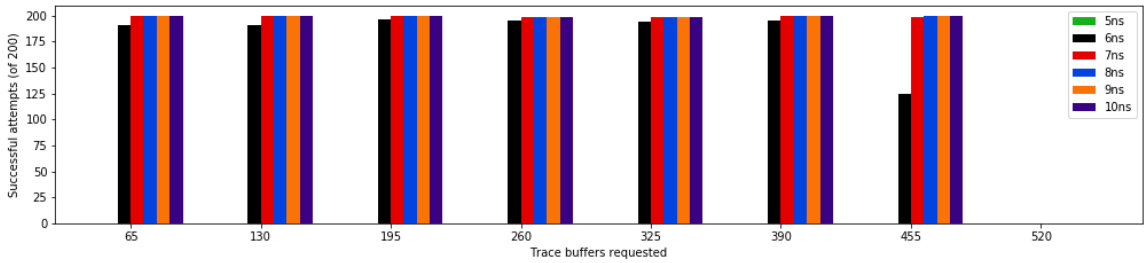
Figure B.2: Implementation success rates for 94% utilized sudoku benchmark with various enhancements implemented one-by-one.



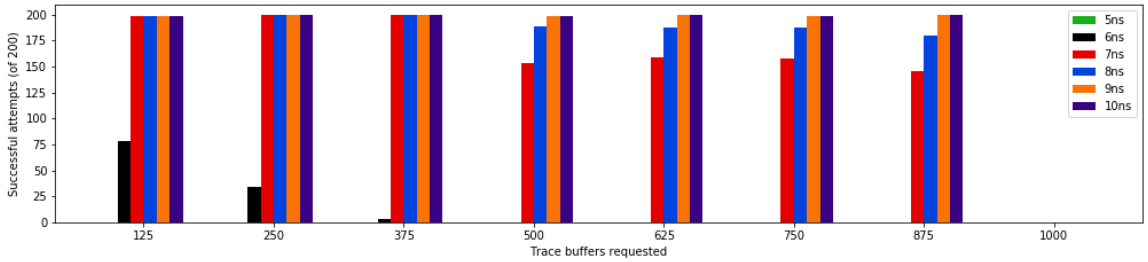
(a) No Optimizations



(b) Simulated Annealing Placement

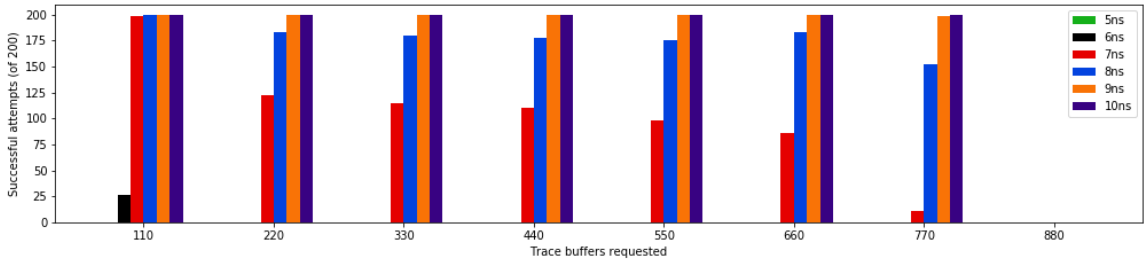


(c) Multicycle Path Constraints on Buffer-to-Buffer Paths

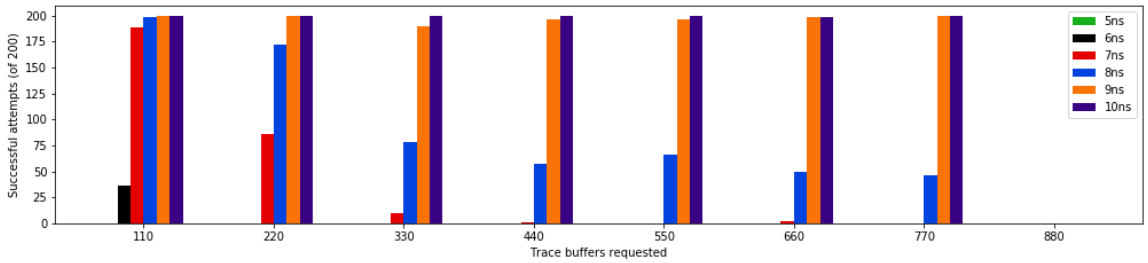


(d) 1% of LUTs Preallocated for Debug (note change to probe counts on x-axis)

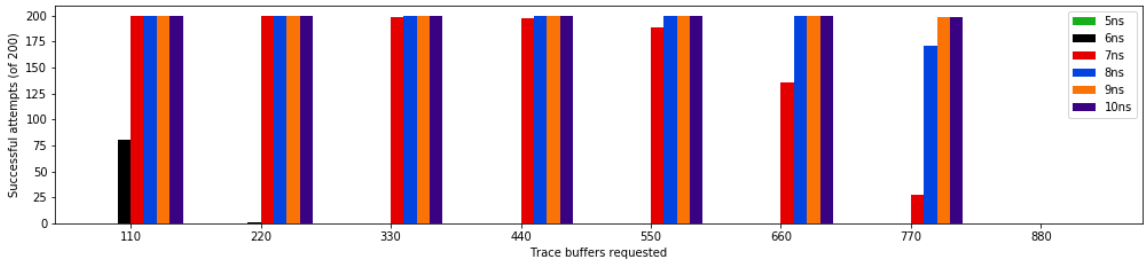
Figure B.3: Implementation success rates for 90% utilized RNG benchmark with various enhancements implemented one-by-one.



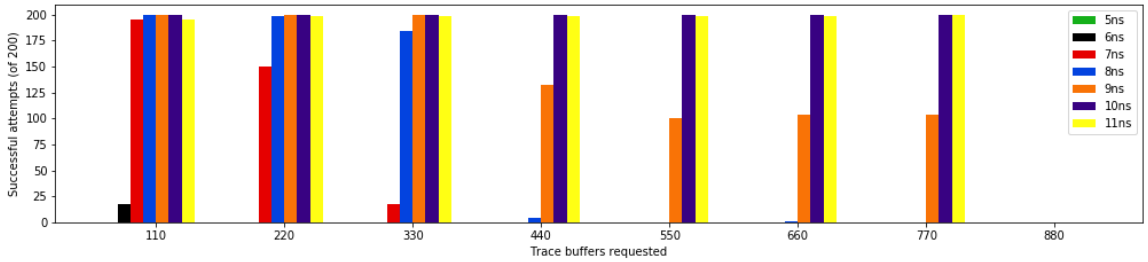
(a) No Optimizations



(b) Simulated Annealing Placement

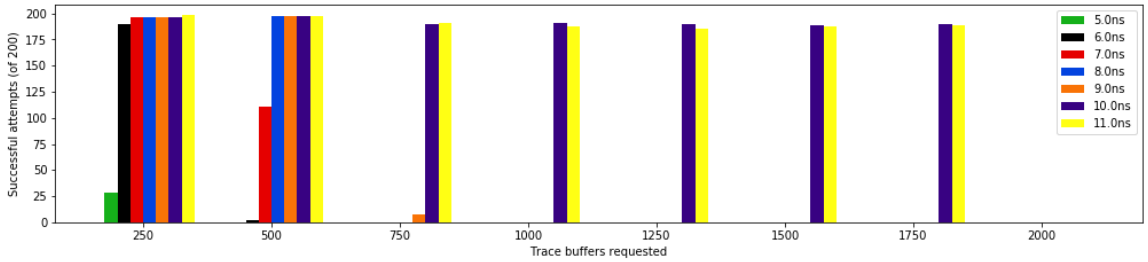


(c) Multicycle Path Constraints on Buffer-to-Buffer Paths

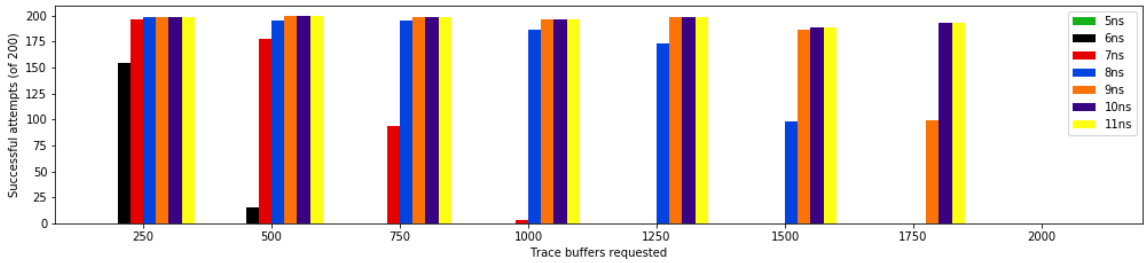


(d) 1% of LUTs Preallocated for Debug

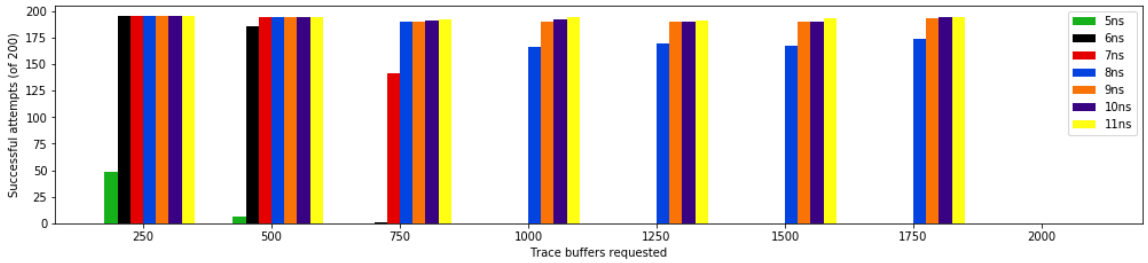
Figure B.4: Implementation success rates for 90% utilized uFIFO benchmark with various enhancements implemented one-by-one.



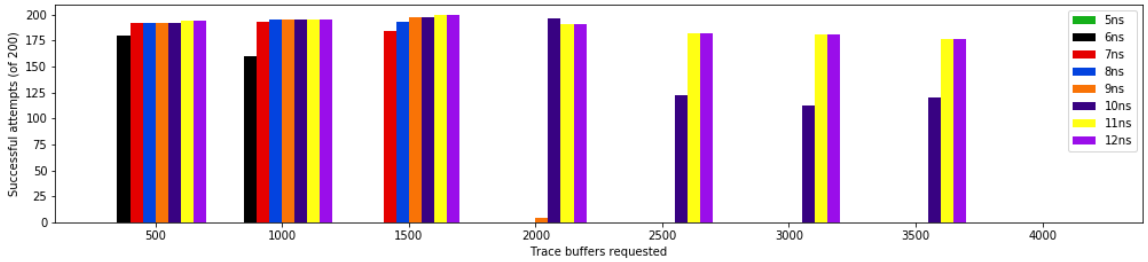
(a) No Optimizations



(b) Simulated Annealing Placement



(c) Multicycle Path Constraints on Buffer-to-Buffer Paths



(d) 1% of LUTs Preallocated for Debug (note change to probe counts on x-axis)

Figure B.5: Implementation success rates for 90% utilized RPulseG benchmark with various enhancements implemented one-by-one.