Automating the Area-Delay Trade-off Problem

Haven Skinner
Dept. of Computer Engineering
hskinner@ucsc.edu

Rafael Trapani Possignolo
Dept. of Computer Engineering
rpossign@ucsc.edu

Jose Renau
Dept. of Computer Engineering
renau@ucsc.edu

ABSTRACT

An ongoing challenge in digital architecture design is the problem of replicating functionality across systems with different timing behavior. Since the maximum clock frequency of a system is based on the longest path between registers, creating systems with a high clock frequency requires that architects divide logic between many pipeline stages, adding additional costs in power and area. This trade-off makes it difficult to share functionality between similar architectures which differ only in desired timing behavior. Fluid Pipelines are a type of latency-insensitive system which can leverage latency-insensitive pipeline transformations as part of the synthesis flow to automatically generate designs with identical behavior but different timing attributes. In this paper we take demonstrate how several four and five stage RISC-V processors can be transformed into three, two, and one stage processors, and compare those against open-source RISC-V processors with similar features.

ACM Reference Format:
Haven Skinner, Rafael Trapani Possignolo, and Jose Renau. 2018. Automating the Area-Delay Trade-off Problem. In *Proceedings of Workshop on Computer Architecture Research with RISC-V (CARRV).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A trade-off which has remained at the core of digital architecture design is the one between power/area and clock speed. Creating a system with a high maximum clock speed requires more hardware, adding costs in power and area. At the code level, this is accomplished by dividing the system's logic between register blocks, which comes with the unavoidable downside of very closely tying the functional behavior of the architecture to it's timing. This means that unlike in software development, it is much more difficult to share code between similar projects.

Fluid Pipelines are an alternative, latency-insensitive, design pattern, where functionality is based only on the order which data is received, not on its timing. This development model separates the behavior of the system from the timing. This also provides a high-level mechanism for manipulating the system's clock frequency: pipeline transformations [10]. Pipeline transformations allow for inserting or removing fluid registers from the design as part of the compilation flow, without changing the code base. This is further discussed in Section 2.

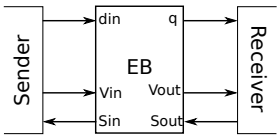


Figure 1: FReg are the basic construct blocks of Fluid Pipelines, they can be viewed as queues of limited size.

In this paper we apply pipeline transformations to several four and five stage RISC-V processors in order to remove registers, transforming the system into one with fewer pipeline stages, without changing the behavior. We compare the three, two, and one stage processors derived through transformations against open-source RISC-V processors with similar features.

Section 2 discusses fluid pipelines and fluid pipeline transformations in more detail. Section 3 describes the design of the fluid pipelined RISC-V processors, showing their internal structure untransformed, and when several example transformations. Section 4 compares the fluid pipelined RISC-V processors against several widely used open-source RISC-V processors with similar attributes. Finally we offer some concluding thoughts and future plans in Section 5.

2 FLUID PIPELINES

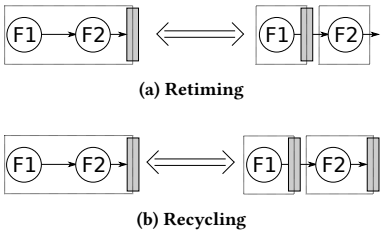
The clock synchronous pipeline remains the dominant method of implementing digital architectures for synthesis. In this paradigm, the system is kept in sync by one or more global clocks. A side effect of this approach is that the behavior of the system is closely tied to its timing.

Latency-Insensitive Systems are an alternative where the behavior is solely based on the order that data is received, and not on the timing. Clock synchronous systems can be automatically transformed into latency-insensitive ones [7–9]; however, exposing the latency-insensitive backend to the designer provides improved performance and flexibility [10, 11, 13]. Prior work has also shown that Fluid Pipelines can be synthesized with with minimal to negligible costs to timing, area, and power [11, 12].

There are a variety of ways to implement a synthesizable, latency-insensitive pipeline; Fluid pipelines [10, 11] do so by replacing the clock synchronous registers used in traditional pipelines with fluid buffers, depicted in Figure 1. The control signals, *valid* and *stop* manage a simple handshake protocol to signal when valid data is available and handle back pressure.

Workshop on Computer Architecture Research with RISC-V (CARRV),
,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Retiming

(b) Recycling

Figure 2: Retiming and Recycling can be applied to fluid systems to improve the clock frequency.

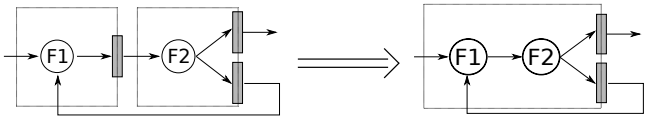
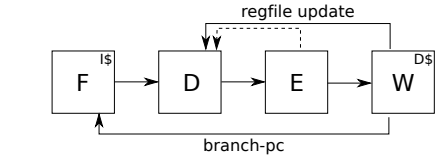
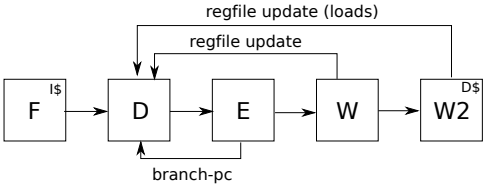


Figure 3: When two stages are collapsed together, registers between them can be removed. Each arrow represents a *fluid connection*, with *data*, *valid*, and *stop* signals.



(a) 4-stage Cliff CPU (c4), which optionally includes a forwarding path, shown with a dotted line (c4+fwd).



(b) 5-stage Cliff (c5)

Figure 4: The 4 and 5 stage "Cliff" cores, which implement the RISC-V 64i instruction set.

Developing a latency-insensitive or fluid system places real constraints on the designers. For example, in traditional digital design, it is possible to assume that if a signal happens a given cycle, a few cycles later something else will happen. For a latency-insensitive system, however, the designer can not assume any time dependencies. From a high level point of view, the design should function properly even if any fluid register in the system adds a random delay before producing output.

Fluid pipelines provide performance improvements over automatically generated latency-insensitive systems by going even further and not assuming any particular completion order between stages. This allows the designer to leverage out-of-order computation, which can improve overall performance [10].

2.1 Pipeline Transformations

Prior work on Fluid Pipelines has shown that since their functionality does not depend on the cycle that events happen, it is possible to change the number of pipeline stages. This operation is referred to as ReCycling [6] (Figure 2b). ReCycling is usually performed to increase the number of pipeline stages, and therefore, increases the maximum frequency of a circuit [6, 10]. In this paper, we are interested in reducing the number of pipeline stages, to automatically synthesize designs with less pipeline stages than present in code. This can be thought of as ReCycling in reverse, alternatively it can be described as reducing the latency between two or more stages to zero.

We model pipeline transformations as an operation done on pairs of stages, "collapsing" one stage into another. This is depicted in Figure 3. When this operation is performed, the "forward" connections between the stages are removed, while the "back" connections remain. Each of the arrows in Figure 3 represent a *fluid connection*, which includes a *data*, *valid*, and *stop* signal. All of these connections are maintained after collapsing stages together, ensuring correct functionality, though many may be simplified away later on in the synthesis flow.

3 CLIFF CPUS

In this paper, we demonstrate how pipeline transformations can be applied to remove pipeline stages from a design to generate different timing variation of the same architecture, without changing the codebase. To evaluate this we developed several four and five stage RISC-V CPUs, which we refer to as "Cliff" cores. Figure 4 shows the cores used for this paper, each of which implement the RISC-V 64i instruction set. The four stage RISC-V Fluid cores have the topology: fetch, decode, execute, and write back, the five stage RISC-V Fluid core breaks write back into two stages.

Each RISC-V core includes instruction and data SRAMs. The Fluid RISC-V cores were developed with a HDL and compiler designed for creating Fluid architectures [12]. The HDL works on the RTL level. The compiler can implement the system in either C++ or Verilog, and can apply pipeline transformations as part of the compilation flow.

Transformations are applied as a set of mappings, mapping the name of a new pipeline stage to a list of stages which will be merged to form it. Figure 5 shows several possible sets of transformations which could be applied to the c4+fwd CPU. These are guaranteed to maintain functionality as long as the design conforms to the fluid pipeline protocol [10–12], and does not contain any implicit time-latency assumptions.

Table 1 compares the Cliff cores to the open-source RISC-V CPUs that they are evaluated against. VScale [4] is a 32-bit, 3-stage CPU. PICORV32 [1] is a 4-stage, also 32-bit. RI5CY [3] is a 4-stage 32-bit core which has additional features such as a prefetch buffer. Zero-riscy [5] is a smaller 2-stage version of RI5CY.

The automatically Fluid transformed cores keep the same original name and add the generated number of pipeline stages. E.g:

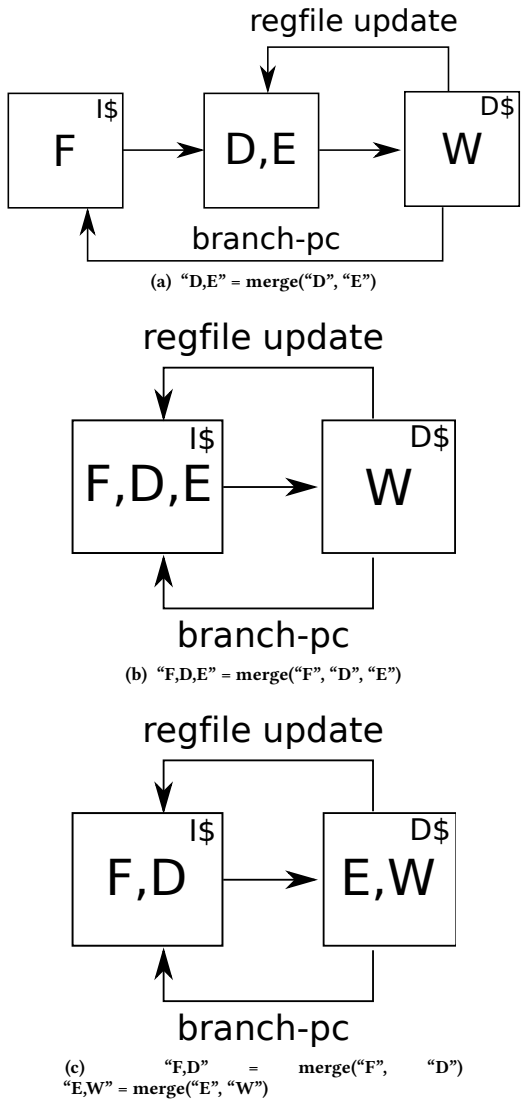


Figure 5: Since the Cliff cores are fluid pipelined, they can be transformed, as shown above.

Table 1: Cores used in the evaluation.

Name	Fluid	Main Characteristics
c4	Yes	4-stage
c4+fwd	Yes	4-stage with forwarding
c5	Yes	5-stage
c5+fwd	Yes	5-stage with forwarding
c6+fwd	Yes	6-stage with forwarding
Zero-riscy	No	2-stage [5]
VScale	No	3-stage [4]
PICORV32	No	4-stage [1]
RI5CY	No	4-stage [3]

c4+fwd 2-stage means that the original c4+fwd Fluid core was automatically transformed to become a 2 pipeline stage core.

For synthesis, we make sure that all the SRAMs associated for data and instruction are not included in the synthesis results by having a similar size and excluding the SRAM blocks from the synthesis itself. When possible we also select configuration parameters to match the Fluid RISC-V cores. For all the cores, we use a commercial synthesis flow that has topological information during synthesis. For each core, we select a target frequency 5% under the maximum achievable frequency for that given core.

4 EVALUATION

This section shows that fluid pipelines can be used to automatically generate efficient cores with less pipeline stages than present in code, and to show that Fluid cores are competitive against existing RISC-V cores.

We do a competitive analysis between all the Fluid cores and several non-Fluid RISC-V cores. For the Fluid RISC-V cores and the available Non-Fluid RISC-V cores, we perform synthesis with a commercial tool that has a topographical aware synthesis, and report the time and area for each core. Figure 6 shows the resulting pareto plot with the y-axes showing the normalized area against c5+fwd, and the x-axes showing the delay in nanoseconds.

One observation is that Fluid RISC-V cores have approximately similar delay-area trade-offs to Non-Fluid RISC-V cores. This is important because it shows that Fluid does not introduce additional design overheads.

Another important observation is that automatically transformed Fluid cores (c4 2-stage, c4+fwd 3-stage…) consistently save area at the cost of pipeline frequency. This is the expected result, but even more interesting is that the generated/transformed cores are consistent with non-Fluid cores like VScale (3-stage) and Zero-riscy (2-stage) RISC-V cores.

One wrong or incorrect observation would be to assume that PICORV32 and VScale are the best because they are in the pareto frontier. Nevertheless, this does not take into account CPI. For Dhrystone, PICORV32 has a 4 CPI, while the c4 has a 2.5 CPI, if the pareto plotted performance, it would show that c4 is ahead of PICORV32.

An interesting observation is that one of the non Fluid RISC-V core (RI5CY) seems to have a significant area and frequency overhead. The area overhead is due to the extra prefetcher available in the model and a multiplier unit that does not exist in the other cores. These two blocks account for approximately 25% of the RI5CY area. The Zero-RISCY is a 2 stage pipeline version of the RI5CY, this more area optimized core is very close to the Fluid transformed c4 2-stage core.

Our RI5CY vs Zero-RISCY area synthesis results are consistent with PULP [2] published results were the Zero-RISCY has approximately 30% of the RI5CY area. In our case, Zero-RISCY has higher frequency, but in the previously published results, the target frequency was a conservative 100MHz for both cores. In this work, we always target the maximum achievable frequency and then decrease 5% the target frequency to avoid complex timing overheads. The same methodology is applied to all the cores.

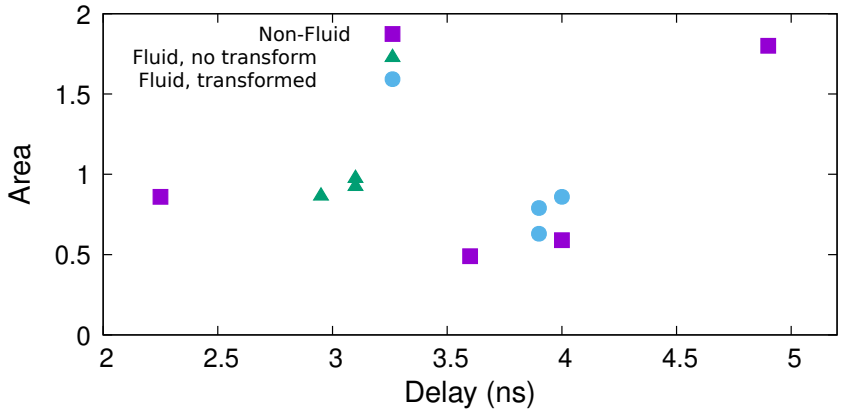


Figure 6: Fluid cores are competitive against manually generated Non-Fluid RISC-V cores.

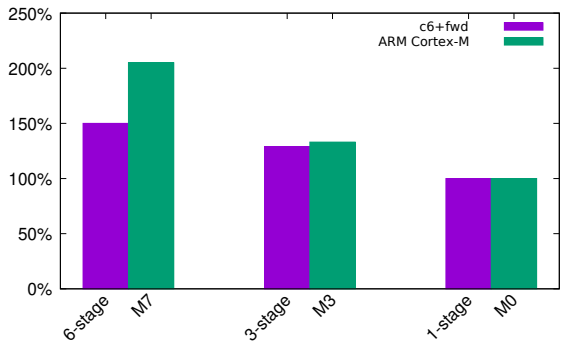


Figure 7: Automatically transformed Fluid cores have similar performance as well tuned ARM Cortex-M cores.

One last interesting observation is that the delay of the c4 does not change going from three to two stages. This means that the merge from three to two did not change the critical path. Considering that Retiming (Figure 2a) allows developers to change the timing of the pipeline automatically, the ability to merge and rebalance may prove a useful addition to this tool.

Besides comparing against RISC-V cores, we also compare a larger 6-stage Fluid RISC-V core (c6+fwd) and its automatically generated shorter pipeline RISC-V cores against ARM Cortex-M series.

To understand the automatically transformed results, we compare against the ARM cores. Figure 7 shows the c6+fwd and the automatically transformed 1-stage and 3-stage performance normalized against the ARM Cortex-M series. The Cortex M7 outperforms the c6+fwd in 6-stage form, likely because its pipeline is simply more balanced. Nevertheless, as we collapse the designs the imbalance is not as important, and the performance change between the collapsed RISC-V and ARM core is lower.

5 CONCLUSION

In this paper we show how fluid pipelines can aid in a common problem for hardware designers: sharing code between similar architectures with different timing properties. This is possible because fluid pipelines separate the behavior of the system from its timing, enabling pipeline transformations. In this paper we show how pipeline transformations can be applied as part of the compilation flow to remove pipeline stages, increasing the delay in order to use less area. In our evaluation we show that the automatically generated cores are comparable to similar cores implemented by hand.

Fluid pipelines, a type of latency-insensitive system, attack a scalability problem inherent with clock synchronous pipelines, derived from the fact that a clock synchronous pipeline's behavior is closely tied with its timing attributes. This makes such systems inherently brittle, as any change in timing, both in terms of critical path and clock cycle latency, can have difficult-to-manage effects on other parts of the system. These scalability issues will only grow worse and digital architectures continue to grow in size and complexity.

Though developing with latency-insensitive systems is not without its challenges, they provide a model for managing larger systems by viewing its components in isolation, and provide a high-level mechanism to control timing with pipeline transformations. A fluid pipeline, which removes any implicit assumption about completion order, can be viewed as a fully distributed, synthesizable system.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] [n. d.]. PicoRV32: A Size Optimized RISC-V CPU. https://github.com/cliffordwolf/picorv32. ([n. d.]). Accessed: 2017-05-25.
[2] [n. d.]. PULP Platform: Parallel Ultra Low-Power Platform. http://www.pulp-platform.org/. ([n. d.]). Accessed: 2017-11-20.
[3] [n. d.]. RI5CY: A Small, 4-stage RISC-V Core. https://github.com/pulp-platform/riscv. ([n. d.]). Accessed: 2017-11-20.
[4] [n. d.]. VScale. https://github.com/ucb-bar/vscale. ([n. d.]). Accessed: 2017-4-4.
[5] [n. d.]. Zero-riscy: A Small, 2-stage Core derived from RI5CY. https://github.com/pulp-platform/zero-riscy. ([n. d.]). Accessed: 2017-11-20.
[6] D.E. Bufistov, J. Cortadella, M. Galceran-Oms, J. Julvez, and M. Kishinevsky. 2009. Retiming and recycling for elastic systems with early evaluation. In *46th Design Automation Conference*. 288–291.
[7] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. 2000. Performance Analysis and Optimization of Latency Insensitive Systems. In *Proceedings of the 37th Design Automation Conference*. ACM, New York, NY, USA, 361–367. https://doi.org/10.1145/337292.337441
[8] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky. 2010. Elastic Systems. In *Proceedings of the 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign (MEMOCODE '10)*. 149–158.
[9] J. Cortadella, M. Kishinevsky, and B. Grundmann. 2006. SELF: Specification and Design of Synchronous Elastic Circuits. In *Proceedings of the ACM/IEEE International Workshop on Timing Issues (TAU 06)*.
[10] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. 2016. FluidPipelines: Elastic Circuitry meets Out-of-Order Execution. In *Computer Design (ICCD), Proceedings of the 34th International Conference on*.
[11] Rafael T. Possignolo, Elnaz Ebrahimi, Haven Skinner, and Jose Renau. 2016. FluidPipelines: Elastic Circuitry without Throughput Penalty. In *Logic Synthesis (IWLS), Proceedings of the 2016 International Workshop on*.
[12] Haven Skinner, Rafael Possignolo, and Jose Renau. 2017. Liam: An Actor Based Programming Model for HDLs. *International Conference on Formal Methods and Models for System Design* (2017).
[13] M. Vijayaraghavan and A. Arvind. 2009. Bounded Dataflow Networks and Latency-Insensitive Circuits. In *Proceedings of the 7th IEEE/ACM Int'l Conf. on Formal Methods and Models for Codesign*. IEEE Press, Piscataway, NJ, USA, 171–180.