Blame Passing for Analysis and Optimisation

Charlie Brej
Dept. of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK.

cbrej@cs.man.ac.uk

Abstract

Potentially, asynchronous circuits can execute faster than their synchronous counterpart because of their average-case, rather than worst-case, performance. In practice, such an advantage is difficult to achieve. A major reason is the difficulty in identifying timing-critical regions of the circuit and analysing the results of changes to the system. The problem arises because static critical path extraction tools used by synchronous designers do not work with asynchronous circuits.

This paper introduces a novel, pragmatic dynamic timing analysis approach to determine bottlenecks in asynchronous circuits. This approach evaluates the behaviour of a circuit within a specific test-bench designed to exercise the circuit in a manner typical of its final application.

Extracted information can then be used to determine which optimisations should be applied, and where those optimisations should be applied. Circuit behaviour information can also be fed back to the designer to allow circuit bottlenecks to be visualised.

1. Introduction

There are many circuit design methodologies which do not use a global clock as a timing reference to mark the completion of operations[1]. Of these, the most relevant in this paper are circuits where the timing of each operation is not bounded but rather is implicit in the data encoding, and in particular circuits with Delay Insensitive (DI)[1] data encodings. Circuits with data dependant timing (e.g. operand dependant matched delays) or data dependant control sequences (in systems made with languages such as Balsa[2]) are also amenable to the approach described in this work, but will not be considered in depth.

Circuits with DI data encodings can have non-uniform timing which is dependant on the operation executed. This is different from synchronous circuits where every operation's execution time is bounded by the predictable clock period. This bounding leads to predictable timing which allow circuit analysis to take place in a static manner. In synchronous systems this takes the form of critical path extraction[4]. The predictability of synchronous systems, and the bounding which the clock provides, naturally leads to systems with worst case performance, irrespective of the pattern of data processed.

Static timing analysis has become the method of choice for synchronous circuit analysis as it has the advantage of high speed of analysis and complete coverage of all significant paths. The lack of simple timing references across an asynchronous circuit can make static analysis difficult. This paper attempts to use simulation and dynamic analysis to exploit the potential for average case performance[3] in asynchronous circuits, where data-dependent timing exists.

One of the design methodologies which tries to exploit average case performance is *early output* logic[5]. In this paper early output circuits will be used to demonstrate the dynamic timing analysis method and the optimisation system.

1.1. Early Output Logic

Early output logic attempts to increase performance of a system by first decreasing the latency of each stage: Through the use of 1-of-n delay insensitive codes, the completion of computations can be determined through the use of completion detection logic on the data outputs of a stage rather than an estimation of stage timing using a worst case delay model. Bit-level pipelining allows the generation of partial results which can be forwarded to the next computational stage while the remainder of the outputs are still being processed. In cases where the inputs which have arrived to a function are sufficient to generate an output, the output generation is not synchronised with the arrival of the remaining inputs. The output is generated in parallel with the gathering of the inputs to the stage. This allows *early output* [5] generation, yet correctly acknowledges all inputs to the stage even if they were late (and so unnecessary for generating the output).

Figure 1 shows a segment of an early output circuit. The communication is done across 4 wires: request zero (R0) and one (R1), validity (V) and acknowledge (A). The early output OR gate is constructed from an AND/OR pair which generate the two data output signals. The validity output from the gate is formed by gathering all validity inputs. The latch cannot acknowledge until the validity becomes high. It asserts its validity output once it is outputting data becomes valid. This style of early output circuit construction is described in more detail elsewhere[5].

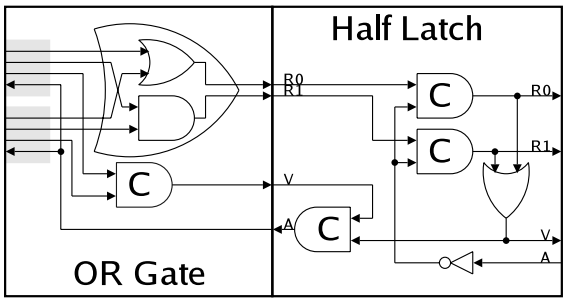

Figure 1: Example early output circuit segment

1.2. Asynchronous Circuit Construction

Most asynchronous circuits are constructed in a manner very similar to that of synchronous circuits. The circuit is composed of computational logic which takes inputs and generates outputs based on those inputs. Latches are used to store data and keep it stable while it is being processed by the combinatorial logic. The main differences in the numerous asynchronous design methodologies come from the use of differing handshaking protocols and data encodings used to co-ordinate transfer of data between latches. Each approach has its advantages and disadvantages. In particular, it is advantageous to ensure that control signalling happens with the same set of signal transitions for each transfer. The power and speed attractive two-phase protocols[6] make this difficult. The use of a 'reset' phase with four-phase protocols[8] leads to simpler circuits, but considerable effort is requires to hide the latency of the reset phase by overlapping it with other circuit activity. Encodings such as four-phase 1-of-4 encoding are popular as the circuits produced are simple and the energy efficiency of the code is good[7].

1.3. Asynchronous Circuit Properties

Early output logic tries to tackle the overheads of the four-phase 1-of-n codes when used in combinatorial logic. Unfortunately, due to the use of the four-phase protocol there remains the reset period problem. A generally accepted method of reducing the effect of the reset period is to doubling the pipelining in the system while keeping the number of tokens the same. This allows half of the logic stages to compute while the other half resets, ready to accept new values.

Fine grain pipelining is not always benefial. It can lead to an increase in the latency of data flowing through the pipeline. Other optimisations such as C-element tree balancing improve response time for each input equally. This often does not take into account the case where inputs arrive in sequence and so balancing the tree can shift the last input to arrive from a position where it was close to the output to a position further from the output.

In order to determine where these optimisations should be applied, the circuit's performance must be analysed when performing 'typical' operations.

2. Static Timing Analysis

There are some static methods which can be adapted to asynchronous circuits.

2.1. Slack matching

Slack matching [9] allows a crude balancing of the level pipelining between two paths with the same start and end points. This method adds additional pipelining latches into the path with the lower pipelining. This ensures that at the start of the fork the two paths are capable of accepting an equal number of data tokens and a stall due to one pipeline being full becomes less likely. This system makes many assumptions such as a equal execution time of each stage, a bundled data system (no bit level pipelining) and no data dependant delays. Another limiting factor is only optimisation which can be applied using this method is pipelining latch insertion.

2.2. Critical path extraction

Synchronous circuits use static timing analysis to extract the critical path and the optimisations rely on making this shorter.

The extraction of the critical path from synchronous designs uses an algorithm which finds the route and the length of the critical path [4]. This process is made simpler because of the assumption implicit in the use of clocked latches that all inputs are applied at the same time.

The algorithm marks the time of arrival of data at each point in the circuit. This is done by determining the latest arriving input to each gate and marking the output time as that time plus the delay of the gate. The outputs of the latches after the active clock edge are marked as occurring at time zero. Once this has been performed on all signals in the circuit, the signal with the latest arrival time can be found and its route between latches can be determined.

This method has several limitations: combinatorial logic loops are not permitted due to the cyclic dependencies produced, and the critical path can include more than one mutually exclusive path which gives a critical path which cannot occur.

This method is sufficient for simple synchronous circuits. Unfortunately most asynchronous circuits do not have predictable and cyclic timing and the static timing approach is not applicable.

3. Blame Passing

A method to analyse asynchronous circuits is crucial to allow the asynchronous engineer to tackle system bottlenecks. As this cannot easily be done statically, it must be performed dynamically.

3.1. Simulation

The basis of the dynamic timing analysis approach is the ability to simulate the examined circuits. In order to observe realistic operation of the unit, the circuit must be placed into a test-bench emulating the environment in which the unit would be used. Because the delays and sequencing of the operations are data dependant, the test-bench must accurately reflect the environment, otherwise the optimisations applied will be optimising the circuit to execute operations or react to environment stimuli sequences which may never occur.

The absolute accuracy of the simulator used is not important (as long as relative delays are reasonably consistently represented) and any level of simulation between behavioural models and post-layout transistor-level analogue simulations could be used. In this paper, an example fixed delay gate level simulator has been used to demonstrate the methodology. A custom gate level simulator was implemented as it is fast and it does not rely on external tool suites to generate satisfactory results for all components.

Once the circuit and the test bench have been loaded into the simulator, the simulation begins with the release of the reset signal. The simulation then continues until the benchmark has been completed. The completion of the benchmark can be signalled by raising a specific signal or it can be time bound and instead of recording the time taken to perform a set number of operations, the number of operations executed in the set amount of time is recorded.

The simulation performs two tasks: 1) measuring the performance of a proposed circuit and 2) extracting information about its behaviour in order to improve the performance further. Even inaccurate simulators, where the exact delay of each component is not known, can be used to extract reasonable comparative performance results, giving a good idea if an optimisation would have a positive or a negative effect.

3.2. Slowest Path Extraction

This paper introduces the concept of a *slowest path*. The slowest

path follows the actual sequence of transitions which accumulated into the delay of the system during the full benchmark. This is different approach from determining the critical path in a synchronous system because the analysis required to find the slowest path need not be exhaustive. Such a restricted analysis allows it to rapidly observe average case performance (rather than worst case). A slowest path is allowed to pass through any unit as many times as is required. This enables the approach to extract the path from long multi-cycle operations and thus also observe the signal interactions in latches as well as logic. The method of extracting the slowest path is loosely based on static timing analysis.

In static timing analysis each wire in the system is marked with the arrival time of the data in a worst case scenario. Once the wire with the latest arriving time has been found the critical path can be extracted. This can be done in a single pass over the circuit (marking signal times can be done at the same time as identifying the currently most critical path). For the sake of simplicity, in the following example we will presume this is done using another pass.

To extract the critical path with a known end point, a path back from that point towards the previous latch for the gate with the latest arrival point at the end point must be found. This path passed back through gates with the latest input arrival times until the output of a latch is reached. The path follows a theoretical sequence of transitions which could happen and so require the clock to have a period longer than the delay of the critical path.

In the dynamic timing analysis, the end point of a simulation run is the benchmark completion signal. In fixed time simulations any signal which transitioned towards the end of the simulation run can be picked. This end point is the last point in the slowest path. The previous point in that path can be determined by finding which was the last input to arrive to the transitioning gate. If this input would have transitioned sooner the operation would have taken less time, and so this input bears the 'blame' for the delay of the circuit. Blame passes from gate input to gate input back through the path (hence "blame passing simulation") until the initial signal is reached (usually the release of the reset).

Unfortunately, if only a single time is recorded for each gate, the cyclic nature of the slowest path will cause that value to be overwritten on each cycle of the simulation. Instead, the proposed technique generates the slowest path forward rather than in reverse during the simulation. As the simulation executes, each transition of a gate is recorded, along with its cause, as the output of the gate could become a part of the slowest path. Should its transition not cause any subsequent gate outputs to change, the transition is counted as a *dead end*. This can now be forgotten as it can not form a part of the slowest path. The recorded transitions keep a reference counter in order to allow their removal should all transitions caused by them have reached dead ends. Discarding dead ends prevents the simulation memory footprint from growing out of control.

3.3. Slowest Path Analysis

The slowest path in any simulation represents the sequence of transitions which accumulated to the complete delay of the simulation. This shows the exact points where the optimisations should be applied as applying optimisations in areas not passed through by the slowest path would not effect the path and the operation will still take the same amount of time. There are exceptions to this rule: optimisations could cause a unit, not on the path, to become slower and become a part of the path, causing the whole system to operate slower. The other exception is in gates where a number of inputs which transition can cause the output to transition. Here the easiest target to focus on is to optimise the sequence of events which led to the transitioning of the first input which triggered the gate. It is possible to also shorten the slowest path by generating a new path through an optimised unit which feeds one of the other inputs which could trigger the transition of the gate before the original first input reaches it.

The optimisation to be applied to the units passed though by the slowest path can be determined by observing the route of the path through known constructions.

4. Optimisations

To demonstrate a number of optimisations, a simple example circuit was designed. The circuit takes a number from an internal constant source and decrements it (storing the result in a register) until it reaches zero, at which point it reads a new number from the constant source. Figure 2 shows the design which consists of a constant source (Const) which feeds a number to the unit each cycle, a register (Reg) which holds the current value, a decrementer (Dec) which reduces the number by one, an OR gate which tests for the number being equal to zero and a multiplexer which picks either the new value or the external constant to be written to the register.

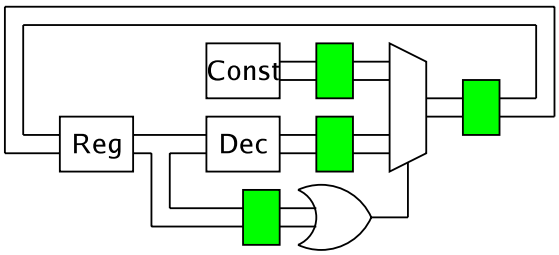

Figure 2: Decrementer circuit

The design was heavily pipelined to allow parts of the circuit to reset in parallel with others processing. The shaded blocks in the figure show the placement of half latches which increase the pipelining of the design. In addition to these, the decrementer was *vertically pipelined* to a single bit level (a half-latch placed on the carry path between each bit slice). This may seem excessive but these latches should be treated only as possible latch locations as any latches which restrict the performance of the design can be removed through one of the optimisations.

The circuit was simulated and its slowest path extracted. The simulation is set to run for a fixed time of 100 000 gate delays, after which a random signal which transitioned in the last time-slice is taken as the end point of the slowest path. The signal can be randomly picked as no matter which signal is chosen, the path, within a small number of gate delays, converges into the same route as with any other signal selected.

One of the methods the designer can use to observe the slowest path (in order to find the bottleneck in the system) is to annotate that path onto the schematic used to design the circuit or a diagram which represents the design. In figure 3 the slowest path is placed on top of a representation of the design. The arrows represent the transitions in slowest path. As the simulation executes many operations of the unit, the paths often take the same route a number of times. In the figure, the thickness of the arrow represents the number of times a particular gate crossing had occurred. Not visible

on the diagram is the distinction between the rising and falling transitions.

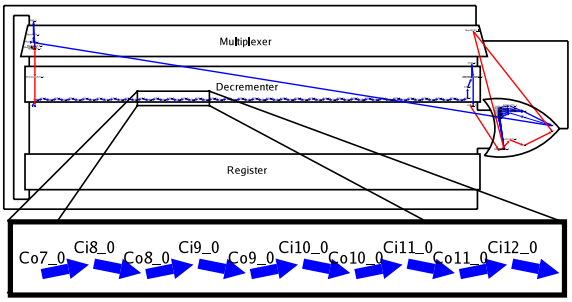

Figure 3: Slowest path in the decrementer design

The zoomed segment in the figure shows the a part of a sequence of transitions which occupy the majority of the slowest path (77%). These are falling transitions along the carry chain of the decrementer.

In this benchmark the constant, which is loaded and decremented, is large (2^{32}-1) with the simulation time comparatively small and so the decrementer never has a long carry chain dependency. The use of early output logic allows a fast generation of a result as the carry signals can be generated locally rather than needing to propagate the full length of the unit.

The generation of the result is not the bottleneck in this benchmark. Instead, the circuit takes a very long period of time to release the signals on the carry chain despite the fact the chain is broken up into small one-bit segments. The root of the problem is the construction of the half latches on the carry chain which prohibit their outputs from dropping while their inputs remain high (after the acknowledge has been applied). This dependency causes the full carry chain to reset sequentially from the bottom and ripples the release of the data signals through the entire unit. This problem can be remedied using an early-drop latch[5]. This latch drops its data outputs upon receiving an acknowledge even if the data inputs remain high.

4.1. Early Drop Latch

The application of optimisations can be described by tables. A positive effect of applying an optimisation can be predicted through the observation of a frequently occurring sequence in the slowest path passing through the element to be optimised. This path is shown in the “Pos” column in each optimisation table. As each optimisation has a possibility to cause a lengthening of the slowest path, the “Neg” column depicts the path which, if observed in the pre-optimised design, is likely to cause the optimisation to decrease the performance of the system. The “Apply” column in each optimisation table presents the optimisation to be applied. Figure 4 shows the table for the early-drop latch optimisation.

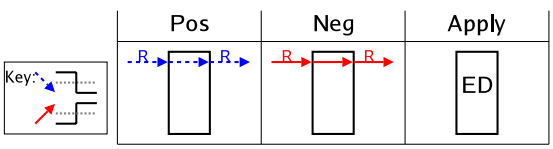

Figure 4: Early drop latch optimisation

The early drop latch releases the request (data) signal as soon as the acknowledge has been released rather than waiting for the request on the input to fall. This optimisation is particularly effective in circuits with the slowest path passing through many latches on the falling data signal transitions (an example of which is the carry chain reset in the decrementer example). The pattern to be matched for the optimisation to be effective is the down-going transition (dashed arrows) of a request out signal being dependant on the request in signal. Replacing the latch with an early drop latch would allow the release of the request out signal to be done concurrently (before the request in signal is released).

An early drop latch does have an additional delay on the rising transitions (solid arrows) of the data signal propagation and should not be used in situations where this frequently occurs in the slowest path.

4.2. Latch Removal

As mentioned before, the number of latches placed in the example design is high and many of them will have a negative effect on the performance of the design by adding latency to the slowest path. Removing a latch can reduce the cycle time by two gate delays (if the latch is on the slowest path in both the set and the reset periods). The danger in doing this is the latch may have been adding pipelining crucial to make the system free flowing. There is little way to determine which latches add pipelining which is useful to the system from the slowest path and this is why the table in figure 5 has that entry missing.

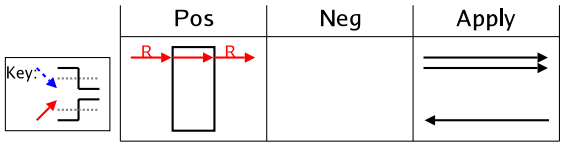

Figure 5: Latch removal optimisation

Instead the optimisation system has to rely on a simulation of the system along with the proposed optimisation to determine if it has a positive effect.

4.3. Latch Insertion

In situations where insufficient number of pipelining latches were placed in the design the optimisation system can spot where a latch is separating two regions which are unable to store two different tokens due to the latch not providing enough decoupling. Only once the stage in front has completed its phase can it allow the stage behind to enter its next phase e.g. the stage in front must complete its reset phase and accept the data from the stage behind before the stage behind can enter the reset phase and release the data. Such situations can be avoided by inserting an additional pipelining latch where the stage behind can commit its data (and enter the reset phase) before the stage in front is not ready to accept it. This can be seen in figure 6.

The danger of inserting latches is the addition of latency. Should the slowest path pass through the latch data signals, the path will become one gate delay longer for every pass.

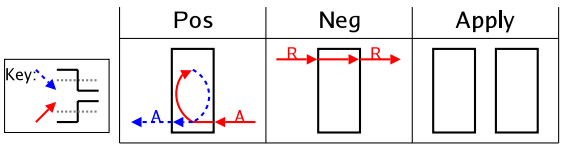

Figure 6: Latch insertion optimisation

The combination of latch removal and latch insertion effectively reproduces the effect of slack matching. The areas concentrated on by the slack matching techniques [9] (unbalanced pipelines) show up in the slowest path as recommendations to remove latches from

the over-pipelined path and to insert latches in the under-pipelined path.

4.4. Anti-Token Latch

In early output circuits, it is often possible to generate the result of a function without the presence of all inputs. Unfortunately, the late unnecessary input must be synchronised with and acknowledged to correctly group inputs. The anti-token latch [5] allows a stage to acknowledge an input which has not arrived yet. The latch then effectively holds an anti-token which propagates back through the pipeline and removes the undesired token.

The slowest path in situations where a stage waits for the token to arrive before acknowledging it passes through a latch from the data input arriving to the validity output rising, signalling the latch is ready to accept an acknowledge. The anti-token latch asserts the validity signal before any data has arrived which exposes it to receive an acknowledge before it holds any data to remove. Here an anti-token is formed and the stage becomes free to process a new set of inputs. Figure 7 shows the table for the anti-token optimisation.

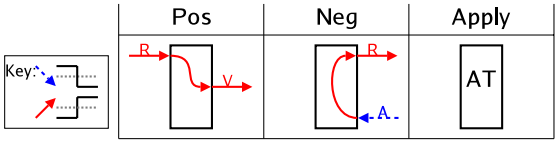

Figure 7: Anti-token latch optimisation

The anti-token latch is larger and slower in a number of transition sequences than an ordinary half latch. The negative effect box in the figure shows just one of the routes which if exist in the slowest path would gain an additional gate delay.

5. Results

To demonstrate the effectiveness of these optimisations they were applied to three circuits and the performance improvement due to each one was recorded. The performance of the original early output design is presented (labelled “Early None” in the graphs), along with the performance after the latch insertion and removal optimisations (Early Half), early drop latch optimisations (Early Drop) and anti-token optimisations (Early Anti). To give a good comparison of the performance of the resultant circuits they are compared with the synchronous equivalent (Synchronous) for which the timing is determined by extracting the critical path. The delay of the latching element and margins for clock jitter are not factored in.

Also presented is a DIMS implementation generated from the same design specification (DIMS None). The DIMS design cannot take full advantage of the early-drop and anti-token latch optimisations but the latch removal and insertion rules apply equally to this design style and the result of these optimisations is also presented (DIMS Half).

Each benchmark was run for 100 000 gate delays and the number of operations executed in that time was recorded.

5.1. Decrementer Benchmark

The decrementer circuit has already been shown, and forms one of the circuits upon which the optimisations will be demonstrated. Because the circuit has behaviour dependant on the data being processed, it is benchmarked with two different internal constant values. The ‘Zero’ benchmark sets the constant to zero, which causes it to continuously reload the constant. The ‘Full’ benchmark decrements from the maximum (32 bit) integer which causes it to never load the number form the constant. The circuits were not only benchmarked for use with the differing constant values but also optimised with them. The results for the circuit are given in figure 8.

The insertion and removal of latches optimisation in this circuit removes many of the latches in the carry path since they do not add to the pipelining of the circuit and instead add latency. This yields a 50% improvement in performance in both circuits. In the ‘Full’ benchmark, another large increase in performance is gained through the use of early-drop latches. The same effect was not seen in the ‘Zero’ benchmark as it does not suffer from the same problem. Instead, a lot of additional performance was gained through the use of anti-token latches which were able to pass anti-tokens to the decrementer once the value was loaded from the constant. This decreased the reset time of the decrementer which was a bottleneck in the performance.

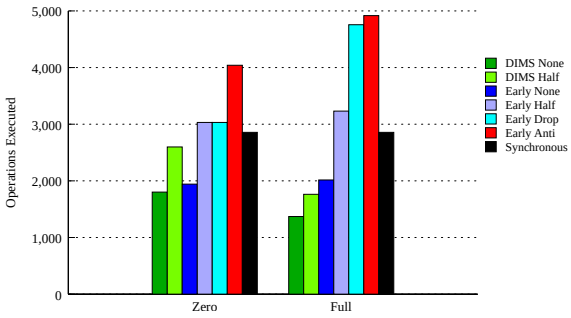

Figure 8: Decrementer benchmark results

5.2. GCD Benchmark

The GCD benchmark determines the greatest common denominator of two numbers. To keep the design simple, the numbers are restricted to 8 bits. The design comprises two dividers which only generate the remainder while discarding the result of the division, two registers to record the current number pair being worked on, a comparator which determines either of the two numbers have become zero and a pair of internal constants. The unit loads a pair of numbers from the constants and then repeatedly divides them by each other each time recording the remainder. Eventually one of the numbers reaches zero and the result is the other number. In this benchmark, the result is discarded and a new pair of numbers is loaded from the constants. The two modes of operation the design is worked on are: two numbers in the Fibonacci sequence, and two zeros. The Fibonacci sequence numbers (223 and 144) require a large number of operations before the result is generated and a new set of numbers is loaded. The zero test loads new numbers on each cycle as the greatest common denominator of two zeros cannot be determined.

The results of the optimisations on this circuit are shown in figure 9. The zero benchmark received a reasonable increase in performance due to the use of anti-token latches. These were effective at removing the results of the unnecessarily executed divisions and allowing the circuit to progress to the next phase. Because the placement of half latches was good, little performance gain is attributed to the removal and insertion of half latches.

5.3. CPU Benchmark

The CPU benchmark uses the datapath of an open source microprocessor [10]. The control signals are attached to pseudo

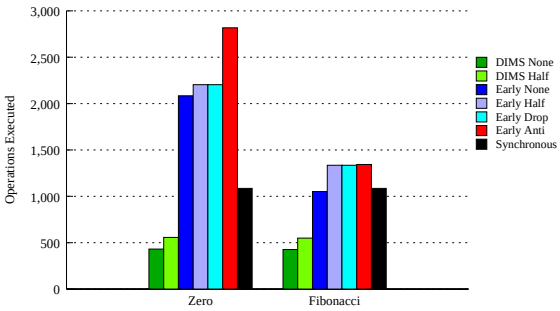

Figure 9: GCD benchmark results

random number generators which cause it to execute random instructions. The memory stage is formed from a delay which is triggered once all address inputs are present. The result of all memory operations is always zero. The delay of the memory stage is either zero for the 'Zero' benchmark or 50 gate delays for the 'Long' benchmark.

The results are shown in figure 10. Because the circuit was already relatively balanced and tuned, in the Zero benchmark none of the optimisations had a great effect. The DIMS circuit gained a reasonable performance increase due to the latch removal. In the Long benchmark, the anti-tokens were able to keep the design executing during memory accesses, the results of which were not requested by the register forwarding multiplexers. Instead, the circuit continued to execute while allowing the memory access to perform a delayed write to the register bank. The anti-token latches placed in the register bank effectively generated a register locking system where the registers which were not being written to generated anti-tokens on their inputs and continued with the next cycle of operation.

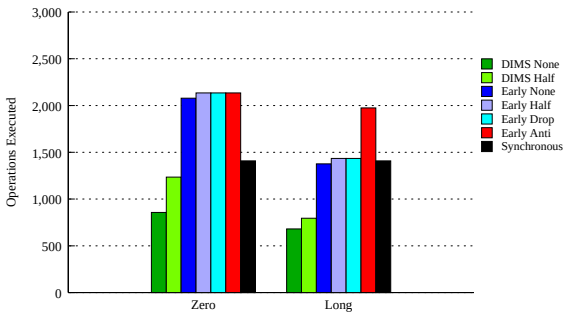

Figure 10: CPU benchmark results

6. Conclusions

Blame passing dynamic timing analysis offers an insight into the operation of a system which allows designer to make decisions about the circuit based on actual system behaviour, rather than making educated guesses about the effect of each alteration. The optimization system automates this process and allows poorly designed circuits to be balanced and offers even good designs additional performance with its use of advanced latch designs.

The blame passing extensions to the custom gate level simulator increased the simulation execution time by 30%. This is relatively small and allows the system to simulate, optimise and re-simulate in short cycles (about 3 seconds per cycle for each of the example designs).

The optimisations performed on the example designs showed cases where designers would be unaware of the real bottlenecks or of possible optimisations. The decrementer circuit was very inefficient due to its long reset time. Engineers tend to concentrate on the processing periods rather than reset periods and so an inefficiency like this would be easily overlooked. On the CPU example circuit, the addition of a register locking scheme by the designer would require a lot of additional work. A simple version of this was constructed by the optimisation system with no designer input. The optimisation system, by replacing a small number of latches, managed to construct a conceptually complex scheme which increases the system performance.

6.1. Future Work

The optimisation system is currently very specific. Only early output and DIMS designs which have been specified in a custom netlist format are allowed. Only a few optimisations have been specified and applied and the simulator can only execute in a fixed gate delay level setup. Future extensions to the system will allow different design methodologies such as bundled data pipelines (such as Micropipelines[6]) and non-pipelined approaches (such as handshake circuits [12]) to be exploited.

The simulator will be extended to read more accurate delay models of components and allow extraction of the slowest path from the event logs of other simulators. Additional optimisations will be added to the current set (stage retiming [11] and tree reshaping).

7. References

[1] J. Sparsø and S. Furber, "Principles of Asynchronous Circuit Design", Kluwer Academic Publishers, 2001, (ISBN 0-7923-7613-7)

[2] A. Bardsley, "Implementing Balsa Handshake Circuits", Ph.D. Thesis, University of Manchester, 2000.

[3] S. B. Furber, J. D. Garside, S. Temple and J. Liu. "AMULET2e: An Asynchronous Embedded Controller", Proceedings of Async 97, pp. 290-299, IEEE Computer Society Press, 1997.

[4] R. B. Hitchcock, G. L. Smith, D. D. Cheng, "Timing Analysis of Computer Hardware", IBM Journal of Research and Development, Vol. 26, 1, pp. 100-105, 1982

[5] C.F. Brej, "Early Output Logic using Anti-Tokens", Twelfth International Workshop on Logic and Synthesis (IWLS 2003), May 2003.

[6] I.E. Sutherland, "Micropipelines", The 1988 Turing Award Lecture, Communications of the ACM, Vol. 32, No 6, pp 720-738, January, 1989.

[7] W.J. Bainbridge, S. Furber, "Delay Insensitive System-on-Chip Interconnect Using 1-of-4 Data Encoding", Proceedings Async 2001, pp. 118-126, IEEE Computer Society Press, March 2001.

[8] D.E. Muller, "Asynchronous logics and application to information processing", Switching Theory in Space Technology, Stanford, University Press, Stanford, CA, 1963.

[9] Andrew M. Lines. Pipelined Asynchronous Circuits. MS Thesis, Caltech-CS-TR-95-21, 1995.

[10] C.F. Brej, "Yellow Star: A MIPS R3000 microprocessor on an FPGA", 2001

[11] S. Hassoun, C. Ebeling, "Architectural Retiming: An Overview", TAU95, November 1995.

[12] K. van Berkel, "Handshake Circuits - An Asynchronous Architecture for VLSI Programming", 1993.