Wagging Logic: Implicit Parallelism Extraction Using Asynchronous Methodologies

Charlie Brej
School of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK
Email: cbrej@cs.man.ac.uk

***Abstract*—Asynchronous circuits have a number of potential performance advantages over their synchronous equivalents due to the ability to exploit average case performance. These advantages are offset by the loss of performance caused by the handshaking overheads which causes designs to be throughput bound. This paper investigates the nature of the throughput problem and proposes a novel automatic approach to overcome its effect. The designs generated using the method not only cease suffering from a throughput bottleneck, but also attain the parallel computation properties despite their original sequential specification. The method is then demonstrated on a processor design. The processor demonstrates the ability of the method to implement a seven gate delay per operation superscalar microprocessor with: register locking, instruction reordering, simultaneous multi-threading, cache-banking and other complex techniques, all automatically or with minor design effort. Such a design can be constructed in days rather than the hundreds of person years required by conventional methodologies.**

I. Overview

This paper examines the performance properties of asynchronous circuits and presents the reset phase of the circuits as a bottle neck. A solution is presented which attempts to overcome this bottleneck, akin to loop unrolling the design. When combined with early evaluation, this solution is then shown to have properties of extracting instruction level parallelism from circuits which were originally designed to be sequential. This property is then demonstrated on a processor design along with descriptions of a series of methods used to overcome some new adverse effects. The work establishes a methodology of rapidly designing systems where parallelism is automatically extracted, thus yielding high performance designs in a fraction of the design effort.

A. Background

Modern high performance processor designs require a vast engineer effort to implement the many features which are necessary to enable the expected high degree of instruction level parallelism. Due to the high level of complexity, many designers abstract the behaviour of the system to a data-flow/token-based representation. This allows the operations to stall while awaiting data or executing more demanding operations. In such systems, during each clock-cycle, each stage executes a handshake with its upstream and downstream neighbours to agree the transfer of data. Not only does this allow a intuitive representation of the system, but also permits advanced features such as early evaluation, where an operation can generate results before all inputs are present, then stall in order to consume the late arriving data.

These features are already implicitly present in many asynchronous methodologies. The removal of the clock allows further advanced elements such as delay-insensitively encoded bit-level pipelining. The simplest delay insensitive code (one which presents its own validity) is a 1-of-2 (dual rail) code where rising one wire signifies a zero and rising the other signals a one. When both signals are low the data deemed to be not ready, and thus the presence of new data can be detected. These codes are also usually used in high speed domino/pre-charge circuits, but gate level logic implementations are also common. Because both the inputs and the outputs of the logic blocks are encoded using delay-insensitive codes, no external timing sources are required to delay the propagation of the data. Instead, individual bits of the output are evaluated as soon as sufficient inputs have arrived. No design effort has to be expended on making circuits early evaluating, the simplest automatically generated logic implementations produce the outputs early.

II. Asynchronous Circuit Performance

Progress in single threaded performance of modern processors is becoming stagnant due to the design complexity of very large scale systems, the design challenges presented by globally distributed clock nets and increasing variation in component delays. Asynchronous circuits have the potential of achieving substantially higher performance targets than synchronous equivalents. Most advantages have been demonstrated and exploited [1], [2], [3], [4], yet these fail to yield the expected performance boost.

A. Advantages

The advantages of asynchronous circuits can be broken down into two groups, those brought in by using local delays rather than a global clock and those achieved by using implicit timing when computing (shown in figure 1)[5]. The clock overheads consist of unbalanced stages and clock skew. Bundle data designs can alleviate clock skew by generating a local stage clock signal by handshaking with neighbouring stages. Although unbalanced stages can cause a degradation in performance in synchronous systems by running at the speed of the slowest stage, the bundled data approach does not automatically solve this problem as

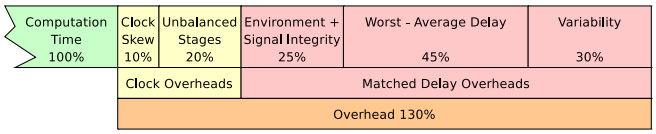


Figure 1. Clock Overheads

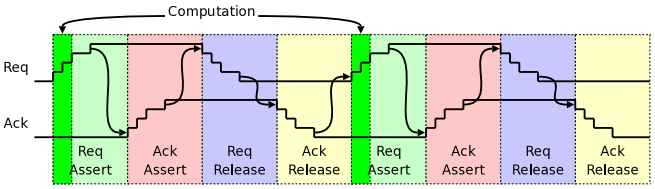


Figure 2. Four Phase Handshake

the design will still have a cycle time equal to the slowest executed stage. So design effort is still required to lessen this effect and it will never be fully removed by selectively executing/bypassing stages.

Matched delay overhead is a much greater component of the clock overheads. These generally cannot be overcome by using bundled data approaches, (although data dependant delays can alleviate some of the worst-average delay effect). The basis of these overheads is the worst-case delay assumptions made by estimating the delay of a computational stage. The worst case delay is calculated by taking the slowest operation within a stage, assuming the worst case voltage and temperature and the most pessimistic distribution of poor transistors and wires.

B. Disadvantages

Judging by the overheads of synchronous systems, it should be easy to implement asynchronous designs with 130% higher performance. Yet rarely do asynchronous implementations come close to the speeds of synchronous designs. There are a number of minor effects which contribute to this (area overhead, overly conservative delays, no speed binning...), yet these do not account for the performance decrease removing all advantage gained by the shift to the asynchronous methodology in the first place.

The one major overhead is the inability of data to progress to a stage which is going through the reset phases due to the previous value passing through it [6]. Many designers seriously under-estimate the granularity of pipelining required to allow tokens to flow freely without colliding with the reset phase of the previous operation. The length of the reset phase is dependant on the protocol and arrangement of the stage.

In the four-phase protocol, the computation happens in parallel with the request assert phase (as shown in figure 2). The protocol then goes through three more phases: acknowledge assert, request release and acknowledge release. In early output designs [6], the output is generated even before the request collection has completed.

Not all the phases are of equal length, for example, bundled data circuits have coarse grain data bundles which require little synchronisation and well constructed asymmetric delays mean the request release phase is much shorter than the computation phase. To demonstrate the latency and cycle delays of different design methodologies, a simple circuit was constructed in a assortment of design styles. The test design chosen was a 32 bit 2:1 multiplexer. Table I shows the latency and cycle delays of the design in eight popular design styles: two phase bundled data micropipelines[7], phased logic using LEDR[8], Mousetrap[9], four phase bundled data[10], DIMS[11], NCL-X[12], early output[6] and the Caltech QDI pre-charged half buffer style[13]. The delay values are given in gate delays and the table is designed to only give very rough estimations of the actual delays. This is because there is a multitude of other factors (delay margins, non-inverting gates etc.) which make a completely fair comparison impossible. Despite this, all design styles show the cycle time being far greater than the latency, with four-phase circuits being particularly affected.

Logic Style	Latency	Cycle Time	Cycle/ Latency	Phases
2 Phase BD	4	8	200%	2
LEDR	4	8	200%	2
Mousetrap	3	6	200%	2
4 Phase BD	3	8	267%	4
DIMS	5	18	360%	4
NCL-X	3	14	467%	4
Early Output	3	16	533%	4
Caltech QDI	2	12	600%	4

Table I
DELAYS OF VARIOUS LOGIC STYLES

The cycle time over latency fraction gives the number of dummy pipeline stages which would need to be added in order to avoid tokens having to wait for a stage to complete its reset phase before accepting the new data. Adding five empty stages to a Caltech QDI style stage would give a stage five latency delays before it was used again. By then it should have completed its reset and be ready to accept new data.

This fraction makes several assumptions which are practically impossible to uphold. These are: that all stages are the same depth, all inputs arrive at the same time and all acknowledgements arrive at the same time. If these assumptions are not upheld, the cycle time becomes longer, making the cycle time 43 times greater than the latency of the stage and causing the stage to require additional pipelining.

One example of a circuit which has a disproportionally long cycle time is an incrementer when implemented in early evaluating bit level pipelined design styles. The average latency of a 32 bit incrementer, implemented in early output

logic, is little more than three gate delays as the inputs bits arrive in a skewed wave-front and the carry chain is usually short. The worst case path of the stage is 30 gate delays with 30 wire forks. The cycle time of the design is two times the 30 C-element delays to gather the requests of the stage, two times the 30 C-element delays to collect and propagate the acknowledge signals and eight additional delays for latching and completion detection. The total is a cycle time of 128 gate delays which is 43 times greater than the latency of the stage.

C. Solutions

There are a number of methods to alleviate the problem of logic stages taking too long to reset. One method is to use a two-phase protocol which has a lower cycle time to latency ratio. This cuts the ratio down but two-phase logic relies on timing assumptions which require additional overheads. Another possibility is to increase the pipelining of the design. This increases the latency of the stage. Slack matching[14], [15] automatically inserts the correct number of latches into circuits but the technique does not take into consideration data dependant stage delays and often generates designs with more latches than gates. The additional latency added by the latches adds a substantial overhead over the pure computational delay. Other methods such as anti-tokens and early drop latches[6] reduce the reset phase, yet these also add latency and are only beneficial in specific circumstances.

III. Wagging Logic

Wagging buffer structures[16] have been shown to be beneficial when constructed in large tree structures[17]. Wagging buffers are an attempt to increase the capacity of FIFO structures without increasing the latency. A wagging buffer alternates the writes to one of two (or more) latches on the input and on the output it alternates the reads from the latches. This structure doubles the capacity of the FIFO, but adds latency of a demultiplexer and a multiplexer. The added latency makes the structure only beneficial when the wagged FIFO already has a capacity of two or more. The technique also increases the cycle time of the input and output stages which are the bottlenecks of the structure.

While wagging buffers aim to duplicate buffer structures in order to increase the FIFO capacity, wagging logic attempts replicate entire logic stages to allow them to reset in parallel. This is done by copying the logic of each stage and cycling which copy of the logic the data should go through. This allows one of the stages to compute while the others are resetting.

Each stage has a *wagging level* which signifies the number of copies of logic the stage contains. Each copy is called a *slice* and has a *slice number* associated with it. The inputs and outputs of each slice are connected to *mixers* which collect the data from the outputs of one wagging logic stage and then distribute the data to the slices of the next stage. The mixers also latch the data to allow the connected slices to work independently. Figure 3 shows an example pipeline with a combination of non-wagging and wagging logic stages. These are connected using a selection of mixers.

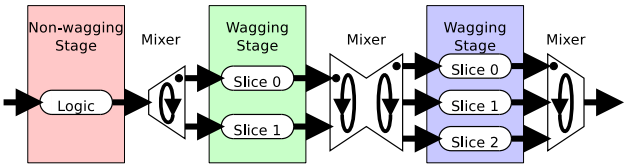
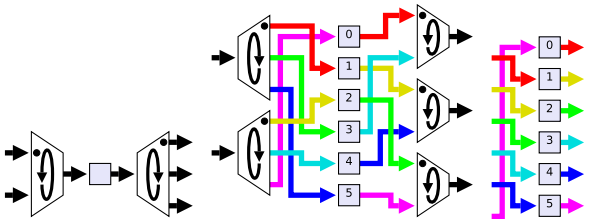


Figure 3. Example Wagging Pipeline



(a) Single channel mixer (b) LCM channel mixer (c) Direct mixer

Figure 4. Mixer designs

A. Wagging Mixers

To connect two wagging stages, it is possible to multiplex the data of the input stage to a single channel and then demultiplex it again to the second stage, in a similar way to wagging buffers. This construction, shown in figure 4(a), has a performance limit of the cycle time of the single pipeline latch and the multiplexing/demultiplexing logic (about 20 gate delays in most four-phase design styles).

An alternative is to demultiplex first to a set of intermediate channels and then multiplex again. The number of intermediate channels is the LCM (lowest common multiple) of the level of wagging of the two stages, shown in figure 4(b). An initial token, at reset time, is placed in latch 0. This is then passed to the first output slice. The value from the first input slice is written to latch 1, then passed to the second output slice. The sequence continues and after six transactions it repeats. The structure does not have a bottleneck stage which will limit the performance, but it still has the added latency of the mixers.

When connecting two stages with an equal level of wagging, multiplexing becomes unnecessary as the lowest common multiple of any number and itself is itself. The output of stage X is passed as an input to stage X+1 (mod the wagging level), shown in figure 4(c). If a single level of wagging is used in an entire design, the latency overhead is limited to the interfaces between wagging and non-wagging logic.

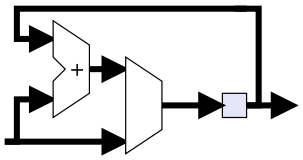


Figure 5. Example Accumulator Design

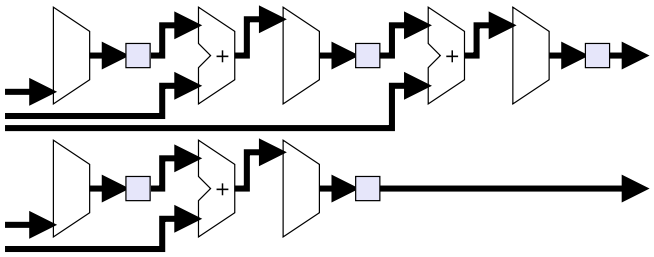


Figure 7. Flattened Accumulator Operation

B. Example Wagging Design

Figure 5 shows a design of a simple accumulator circuit. The circuit has two operations: "Load" reads a new value into the register and "Accumulate" adds the value in the register to the input value and writes the result back to the register. The type of operation executed is declared in the input channel, and it directs the multiplexer to pick the appropriate value. The contents of the register is also passed out each cycle. The worst case delay of the stage is the delay of the adder and the multiplexer.

Figure 6 shows the design and example operation in a level six wagging logic implementation of the accumulator. The sequence of operations passed through it is:

0) Load
1) Accumulate
2) Accumulate
3) Load
4) Accumulate
5) Load

In the figure, the data dependencies can be seen as the black units and arrows (results of grey units were discarded). There are the two sequences of accumulates which have no data dependencies (the black line regions are unconnected). Because the second data dependency region does not rely on the results of the first, the second set of accumulations can be executed in parallel (subject to the arrival of inputs).

This can be seen in figure 7, where the operations conducted by the circuit are shown flattened. Unless there is a conflict over hardware resources, which in this case there is not (thanks to the high level of wagging), the timing of operations is only dependant on the arrival of inputs. If data of the first self contained sequence is late, or the computation is slow, the second sequence can complete before the first has generated a result. This allows proceeding stages to try conducting as much processing as possible before they strictly require the data from a late arriving input.

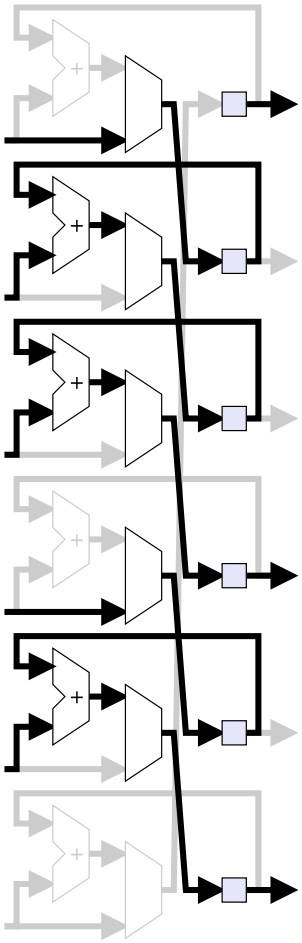


Figure 6. Wagging Accumulator

Because early output logic uses bit level pipelining, each bit of the result progresses to the next stage as soon as it is calculated. This is highly advantageous when using units such as adders or incrementers which have an ordered sequence of desiring input values and generating output values. This greatly reduces the delay of two adders placed in series as parts of the result generated early by the first adder are the parts desired first by the second adder.

C. Results

To examine the performance advantages of wagging logic, a system was constructed which allows the user to create wagging circuits and analyse their performance. The system takes designs written in synchronous structural verilog as an input, allows them to be desynchronised into early output dual-rail circuits and selectively waggs parts of the circuit to a chosen level. The final implementation can then be simulated and have its performance evaluated. The performance is, by default, measured by counting the number of results generated within the time of 10,000 gate delays (deemed sufficiently long to extract an accurate performance figure) in a simulation. As a rule of thumb, the number of operations executed within 10,000 gate delays is equal to the number of MIPS if the design had been implemented in 300nm technology. The tool also outputs the slowest path[6] of the simulation which can then be used to improve the performance of the design.

The result of the 32 bit incrementer stage, described in section II-B, can be fed back to the input of the stage, creating a self-sustaining circuit which continuously increments a value. Without any wagging, this circuit manages to execute 78 operations within a 10,000 gate delay simulation run. This gives a cycle time of 128 gate delays (as predicted). When wagging the stage there is an option of which wagging mixer should be used. If a direct mixer is used (see "Inc.

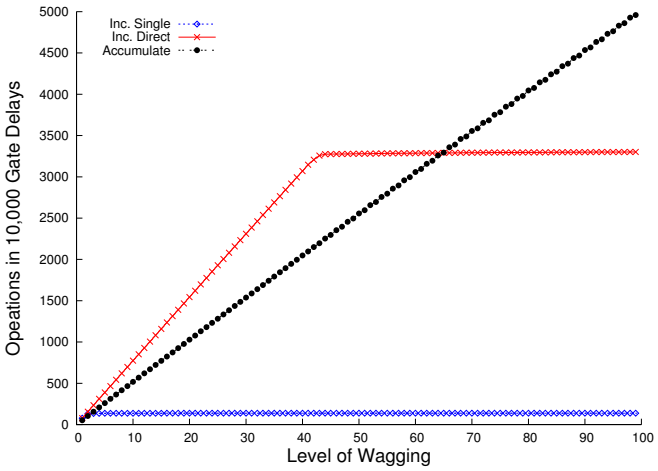


Figure 8. Performance of Wagging Logic Stages

Direct" in figure 8), each wagging slice added to the design adds 78 to the number of operations able to execute in the simulation time. As the design is very throughput bound this can continue up to a wagging level of 43 (executing at three gate delays per operation), when the design becomes latency bound and the performance cannot be increased with the addition of extra logic slices. If a single channel mixer was used in the design, the performance could increase to 138 operations, within the simulation time (72 gate delays per operation), at which point the bottleneck of using a single stage becomes dominant (see "Inc. Single" in figure 8).

In the incrementer design, each operation was dependant on the result of the previous calculation. When this is not the case, a new execution cycle can begin before the result of the previous cycle has been generated. An example of this is the accumulate circuit described in the previous section. The design was constructed with inputs to each wagging slice supplying operations infinitely fast so as to only measure the performance of the stage itself (and not the input logic). A repeating sequence of one load and one accumulate was executed. The results in figure 8 (labelled "Accumulate") show the increase in performance does not reach a ceiling. If the graph was extended, the number of operations would continue to increase to infinity (assuming a sufficiently fast supply of inputs and collection of outputs).

The same technique was applied to a selection of ISCAS benchmarks. The inputs were connected to data generators, which would exercise the circuit, and the outputs were accepted and acknowledged. When the circuits were wagged, nearly all gave results similar to the accumulator example, where the performance could be increased indefinitely by adding more logic slices. Some circuits with specific data input patterns exhibited a sequenced operation where each computation cycle was dependant on the data generated by the last (and thus had a performance ceiling), but without a clear understanding of the function of the circuits it is difficult to see if the designs were executing in a sensible mode of operation. With all ISCAS benchmark circuits and input data patterns examined, the wagging designs took less than ten gate delays per operation. Because of a lack of suitable large scale benchmarks to demonstrate the feasibility of the technique, a new design was constructed to demonstrate the wagging logic design methodology.

IV. Example Processor Design

"Red Star" is a simple processor, based on the MIPS[18] instruction set, designed to explore the possibilities of wagging logic. Using wagging logic, and other techniques outlined below, the design becomes very large and it is not suggested that this is a reasonable alternative to common place practises at this point in time. What is presented is a set of techniques which will become increasingly viable as the price of increasing single threaded performance (in both area and design effort) continues to rise. The target of the design is to implement traditionally complex architectural features with little or no design effort and combine this with the fastest computation possible. The issues of area and power consumption are not addressed at this point.

A. Datapath

The datapath, shown in figure 9, was designed in a synchronous style with a small alteration. The design is fully functional in the synchronous form. The alteration made is the addition of two stages before committing the results back to the register bank or updating the PC to a branch target. This increases the branch shadow by two instructions. This is unnecessary in the synchronous version. In the asynchronous wagging design this increases the number of instructions prefetched. The penalty of prefetching a greater number of instructions is not felt in the wagging version as the resources wasted executing these operations are not shared with those of instructions fetched at the branch target address. Nor is there a large delay before the branch target instructions begin to be fetched, as the new PC value quickly progresses through the two additional empty stages and informs the target slice the address to fetch the next instruction from.

B. Register Bank

All storage, in early output logic, is constructed using FIFO latches. Each cycle, the value is removed from the latch and a new value is written to it. Register bank latches have enable inputs which select whether the new data input should be written or the old value should be recycled. This kind of construction is wasteful of both energy and area, but it fits well with wagging logic. In wagging circuits, the contents of the whole register bank is copied to the register bank in the next slice. If the data, a register is to be written with, arrives to the register bank late, the register value will

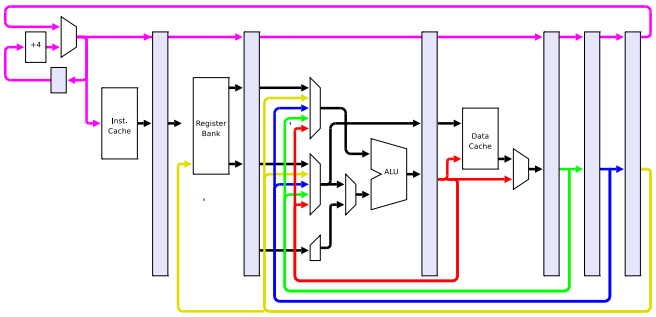


Figure 9. Red Star Datapath

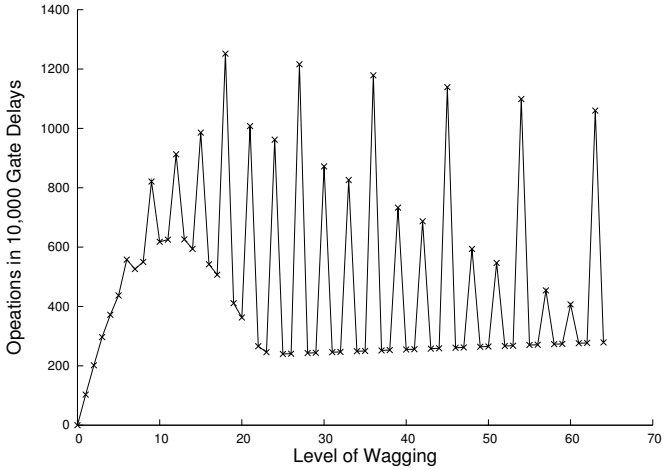


Figure 10. Performance Executing a Nine Instruction Loop

arrive at the next slice some time later than the others. Unless this register is read in the next slice, there is no need to wait for its arrival before reading one of the other registers. Once the value is entered into the register bank in one of the previous slices, it can propagate through the slices and quickly catch up the computation wavefront. This enables the register bank to block while waiting for the value to be written. Additionally this allows the register bank reads and writes to be executed in parallel and even out of order. If the value, to be written to a register, is still being computed, there is nothing stopping the following instructions from writing to the next slice of the register bank (to any register).

C. Caches

The two caches (instruction and data) are the only connections between the processor and the environment. The position of the interfaces between wagging and non-wagging logic is likely to be a bottleneck unless a low bandwidth point is chosen. Having a single cache shared between all slices will cause the system to be bound by the performance of the cache. The alternative of treating the cache like a register bank and copying its entire contents to the next slice is impractical. Here, two different approaches are used to allow independent parallel access to a component, yet keep coherence between the instances.

1) Instruction Cache: A parallel access instruction cache can be constructed by adding a cache to each slice of the wagging design. Although the data contained within the caches of the different slices will be different, the processor does not insist that data, once fetched in one cycle, persists into the next cycle. The individual caches in each slice need not keep coherency. Figure 10 shows the performance of the Red Star processor (using individual instruction caches in each slice) executing a loop of nine instructions (including the branch shadow). The circuit starts with no valid data in the caches and all instructions have to be fetched from RAM. After executing nine instructions the loop restarts and the same nine instructions are re-executed.

If the chosen level of wagging was nine, each slice would be re-executing the instruction it executed the previous iteration, and thus that slice's instruction cache would already contain the relevant data. Because nine is not a sufficient wagging level to permit the circuit to execute in non throughput bound operation, level 18 wagging performance is greater, although it fetches the set of nine instructions twice.

If the chosen wagging level was 16, or another number which shares no common factors with the number of instructions in the loop, then each slice's cache will end up fetching every instruction in the loop. This is made worse by the large number of caches which have to be filled with data from RAM. Wagging levels with common prime factors with the number of instructions in the loop give higher cache hit rates.

A microprocessor which is only fast when executing loops of nine instructions is not desirable, so a method of increasing the cache hit rates in any size loop (or program structure) is needed. Instead of updating the PC to the branch target in whichever slice it gets delivered in, it is possible to delay the effect of a branch and start execution of the new instruction block in a more appropriate slice. For the computation to be aware which slice it is currently executing in, a slice number component can be used which outputs the slice number the data is passing through. This can be constructed to statically output a fixed number for each slice, or generate the number dynamicly by using a mod N counter (where N is the wagging level).

By comparing the branch target address with the slice number, it is possible to determine if the slice should fetch this instruction or pass the branch target unchanged to the next slice and execute a NOP. The easiest way to designate which slice should start the execution of a new instruction block is to compare the address modulo the wagging level to the slice number. Performing modulo operations becomes trivial if the divisor is a power of two (result is equal to the bottom bits of the dividend). For example a block of instructions at branch target address 0x01E8 should be

executed starting from slice 8 if the wagging level was 16 (0x01E8 mod 16 equals 8).

If a power of two wagging level was used, each slice would only execute instructions which have the slice number in the bottom bits of the address. The advantage of this technique is to break up the large instruction cache into smaller faster caches, allow their access to be parallelised, ensure the data is not replicated in multiple caches and allow the removal of several bottom bits of the address stored in the cache (as these are guaranteed to be equal to the slice number).

Once implemented, the 16 way wagged, branch delaying design executed at 8 to 20 gate delays per useful instruction executed (i.e. not counting the inserted NOP instructions). Loops of size 17 gave very poor results as each branch caused 15 stages to be skipped and execution to resume in a stage which had only just entered its reset phase. This was improved by wagging at level 16 but using a mod 8 slice counter to determine the branch start slice. This reduces the worst case penalty to 7 skipped stages which only sometimes propagates to a slice which is still resetting. This design run at 8 to 10 gate delays per useful instruction executed in all loop sizes above 4. Better results could be obtained by making the compiler aware of the penalty associated with specific branch distances.

2) Data Cache: The data cache requires coherence between caches in different slices, but the bandwidth to it is much lower than the instruction cache, so the option of having a single data cache is a possibility. The alternative is to implement independent write-through caches with a coherency network between them. This is an attractive option as there is correlation between the data addresses accessed and the instruction addresses they are accessed from. This could lead to increased cache hit rates.

The data cache can be constructed using a set of separate caches, along with a small level zero (L0) cache which is propagated the same way as the register bank. The L0 cache stores the write accesses of the previous N instructions. If there is no cache access in a slice, a value from the L0 cache is updated in the L1 cache and the committed flag is marked for that cache number in the L0 cache. Once a value has been committed in all L1 caches it can be removed from the L0 cache. This strategy is somewhat complicated and a routine to deal with an overflow of the L0 cache must also be designed. It is also possible to reduce the number of data caches from the number of slices down to a lower number by sharing a single cache between several slices. Although these possibilities are being examined, because the benchmarks used are not sufficiently memory access demanding, the data cache is rarely in the slowest path, and thus improvements in its performance have little effect on the performance of the design as a whole. This remains part of future work.

D. Simultaneous Multithreading

The latch inputs to slice zero are of a different design than the other latches in the system. Most latches are made with a half-buffer design but the slice zero latches must start with a token reset time which form the initial circuit state. These latches could start with two tokens at reset time causing two computational wavefronts. Because the data of the two wavefronts is carried with them, the circuit holds no state when the next wavefront reaches it. This allows the two threads to be fully independent yet be executed on the same hardware. This effectively replicates simultaneous multithreading, where a single set of resources is shared between two (or more) threads. There are still points where the design must be altered to assure the correct functionality. Elements which do not keep all their data in the wavefront (in this case caches) must be manually protected from one thread accessing or damaging the data of another. This can be achieved by adding a variable which holds the ID of the thread, carried in the wavefront. This ID can then be used by the caches to determine which data is accessible.

The second necessary change is in the interface with non-wagging logic. The interfaces now, rather than cycling through the slices in order, must interleave accesses from the two threads. This can be done in either a deterministic pre-calculated sequence, or using arbiters to service the requests in order of their arrival.

The latch controllers of slice zero could dynamically add or remove threads from the processor with the state of the whole thread being moved to or from the slice zero latches. This procedure could be controlled by an external process or by the design itself.

Experimental results show the strategy is easy to implement but requires the wagging level to be doubled to retain the same performance. Doubling the area in order to execute two threads could be attained by making two versions of the same processor, so there is no obvious advantage to using this technique. Possible advantages of simultaneous multithreading include sharing cache and other resources, direct communication between threads and distributing activity of a single thread over a larger area to avoid hot spots. Further investigation into this field will be conducted.

E. Optimisation

Extracting the slowest path[6] of a simulation run is a powerful method of finding the bottleneck in the system. In the Red Star design the slowest path was mainly in the ALU adder and the branch target adder. These were initially constructed using a ripple carry design. This design performs very poorly when adding a small positive to a small negative number. The delay before generating the result can be up to 64 gate delays (as much time as is allocated for 10 instructions). Using a standard fast carry or carry select adder to implement both adders would have been an

option but here an incremental optimisation approach gave favourable results.

The incremental method looks at the slowest path and tries to shorten it by rearranging the logic or adding *hinting logic*. Hinting logic is superfluous logic which is designed to generate a result much faster than the normal logic, but only in specific circumstances. In the case of the adders, the hinting logic detected long carry chains and propagated a carry value directly to a specific adder slice. This would only be activated when the long carry chain is present but when it does it saves many gate delays worth of latency. The biggest advantage of this method is to generate units with low latency on specific paths as concentrating on all paths will increase the average case delay.

When applied, this technique decreased the cycle-time per instruction down to seven gate delays even on subtraction and branch back heavy benchmarks. The ALU adder was implemented with a fast top bit result generation (for sign test operations). The branch adder was constructed with fast generation of the bottom eight bits, as these are used to as the index into the instruction cache with remaining bits of the result then having several additional gate delays before they are needed. With more effort it should be possible to reduce the instruction cycle-time to about five gate delays per instruction without making any architectural changes.

F. Results

Although the design contains gates which have a high fan in, these gates only exist in the reset logic and do not form a part of the slowest path. If they did then the issue could be resolved by increasing the wagging level or increasing pipelining in the problem stage. The current design executes a number of benchmarks (made from kernels of compiled C code segments) at an average of seven gate delays per instruction. If the design would be technology mapped, the delay per instruction would be 10 ± 3 inversion delays, depending on the success of the technology mapper and layout. Table II shows the performance of a number of synchronous and asynchronous designs[19], [20], [21], [22], [1], [23], [24], [25], [2], [26], [27]. To compensate for technology scaling, the performance is given in number of inversion delays per instruction (on their respective process technologies). The delays of inversions were taken from TSMC standard cell libraries[28] for each technology. There are factors which are not considered, such as the complexity of different instruction sets and the manufacturing speed of the benchmarked sample (some designs were manufactured on slow runs and the Pentium D and Core 2 chips were from slow bins), thus the results in the table should only be used as a rough estimate.

Processor	MIPS	Tech. (nm)	Inversion delay(ps)	Inversions/ Instruction
SPA	6	180	60	2778
Philips 80c51	4	500	170	1471
ARM996HS	83	130	45	268
nanoSpa	63	180	60	265
Amulet 2	40	500	170	147
TITAC-2	52	500	170	113
Aspro	115	250	85	102
Lutonium	200	180	60	83
Amulet 3	115	350	120	72
Fulcrum Vortex	475	150	50	42
MiniMIPS	180	600	200	28
Pentium 1 300	425	250	85	28
Pentium D 805	2746	90	30	12
Core 2 E6300	4502	65	22	10
Red Star				10 ± 3

Table II
PERFORMANCE OF A SELECTION OF PROCESSORS

V. FUTURE WORK

The work presented in this paper is still in its early stages and although the aim of achieving high performance computation appears on target, it is important to examine the design methodology's position and its future targets.

A. Power and Area

The work presented makes a fundamental assumption which must be true for the method to yield beneficial results. When generating parallel designs, the unit that is to be parallel must be duplicated. This trades a doubling of the area, in exchange for a possible doubling of the performance. This performance may be then decreased due to the increase in wire capacitance. Despite this, this is an assumption that all parallel system designers take for granted, and has been shown to be true through the numerous highly parallel high-performance designs present on the market today.

The energy consumption of wagging circuits is nearly identical to the original non-wagging circuit which forms each of the slices. The same number of logic gates transition for each operation, but these transitions are dispersed among the slices. In non-wagging designs, slack matching latches need to be inserted to improve the performance. These cause the additional energy consumption which is not present in the wagging version. Latches between slices are constructed using half-buffer designs (except for slice 0), rather than latches which hold a token at reset time, have adequate decoupling (to not deadlock the system) and thus consume more power. Finally, because the computation phase is quasi-delay-insensitive, the design has implicit voltage scaling compensation and resistance to environmental fluctuations.

Static power consumption is a problem which increases with area which increases with the wagging level. For low wagging levels the increase in performance is also linear with the wagging level, thus the static power per operation executed remains the same. If the slices of the design are not placed in a regular manner, there is a possibility of longer wires which may increase capacitance and delays. Effects such as this may become relevant with large designs and high levels of wagging.

It is important to consider all these issues when choosing the where, at what level and even if to apply the wagging. Wagging is not suitable in all instances and the paper explores the upper limit of the approach. There is no reason to apply the approach to the point of saturation, and if a doubling in performance is sufficient then a two way wagged design should be used.

B. Design Flow

Parts of the flow, such as desynchronisation to early output logic, wagging, simulation, timing extraction and dynamic timing analysis, have been implemented. This is a product of several years of effort. The flow still needs a technology mapper and an input language. The technology mapper will enable a more accurate comparison between the different technologies and even permit post-layout measurements. Desynchronisation is used to allow an easy design input. This is already showing its limitations as components such as additional buffering, conditional transactors, memory blocks etc. have to be added into the verilog parser and treated differently from other elements. The use of a language gives a more high level way of implementing these features. This protects against their misuse and allows verification which could assure non-deadlocking function of advanced components.

An input language also allows the design to be compiled into a network of course blocks (such as adders and multiplexers) which the optimisation system could be aware of. These blocks may have possible optimisation techniques described for them to allow the slowest path analysis to pick and apply the most suitable optimisation to overcome a specific bottleneck. This has already shown itself to be a very powerful technique to create custom components for their specific applications (in the branch and ALU adders of the Red Star design). These optimisations were done manually but an automatic approach is being worked on. Automatic latch removal and insertion has been implemented and the optimisation system will be expanded to alter the logic structure and shorten specific paths.

While the more advanced features of the design flow are finalised, a small experimental processor is being manufactured. Executing the 80c51 instruction set and manufactured in 130nm technology, the processor aims to demonstrate the ability of wagging logic to implement complex non-RISC multi-cycle instruction sets and explore new caching structures. The level of wagging was set at 11 to examine the point of saturation, although the performance does not increase above level eight. All slices are by-passable to allow the removal of any faulty slice, and test the performance at different levels of wagging. The returned silicon is expected in March 2010. The post layout results look positive, simulations yielding speeds roughly 100 MIPS unwagged, and 800 MIPS when wagged, although these are highly dependant on code executing. During 2010, it is hoped that both the completed tool-flow and the Red Star example design will be available to the public.

VI. Conclusions

This paper presented a powerful strategy to overcome the greatest obstacle in generating high performance asynchronous circuits. The application of the method and additional high-performance techniques are demonstrated on an example processor design. Implicit data dependency tracking allows the engineer to concentrate on higher level architectural improvements rather than worrying about blocking stages and sharing resources. Bit-level early output logic also allows average-case latency based computation, but the methods shown in this paper can just as easily be applied to any asynchronous logic style.

It is hoped that wagging designs could compete with current high performance cores, but with a fraction of the design effort. This can already be demonstrated, to a small extent, by the relative ease of constructing complicated features in Red Star. The processor took one person week of design effort, and optimisation to achieve seven gate delays per instruction took another three person days. This was deliberately short in order to research automatic optimisation of a rapidly implemented processor, but, even without an optimiser, this simple implementation reaches the performance of current cutting-edge designs. Within a very short period of time it would be possible to implement a world class processor with speeds unattainable by synchronous design methods.

References

[1] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *International Symposium on Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, 1997.

[2] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, D. W. Lloyd, S. Mohammadi, J. S. Pepper, S. Temple, J. V. Woods, J. Liu, and O. Petlin. Amulet3i - an asynchronous system-on-chip. In *International Symposium on Asynchronous Circuits and Systems*, page 162, Washington, DC, USA, 2000. IEEE Computer Society.

[3] Montek Singh, Steven M. Nowick, Jose A. Tierno, Sergey Rylov, and Alexander Rylyakov. An adaptively-pipelined mixed synchronous-asynchronous digital fir filter chip operating at 1.3 gigahertz. In *International Symposium on Asynchronous Circuits and Systems*, page 84, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *International Symposium on Asynchronous Circuits and Systems*, page 198, Washington, DC, USA, 2000. IEEE Computer Society.

[5] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. Coping with the variability of combinational logic delays. *ICCD*, pages 505–508, 2004.

[6] Charles Brej. *Early Output Logic and Anti-Tokens*. PhD thesis, University of Manchester, 2005.

[7] Ivan E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, 1989.

[8] Robert B. Reese, Mitchell A. Thornton, and Cherrice Traver. A fine-grain phased logic cpu. *isvlsi*, 00:70, 2003.

[9] Montek Singh and Steven M. Nowick. Mousetrap: Ultra-high-speed transition-signaling asynchronous pipelines. *iccd*, 00:0009, 2001.

[10] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Kluwer Academic Publishers, dec 2001.

[11] David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.

[12] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous cad tools. *IEEE Des. Test*, 19(4):107–117, 2002.

[13] A. Lines. Pipelined asynchronous circuits. Master's thesis, California Inst. of Technology, 1995.

[14] Peter A. Beerel, Nam-Hoon Kim, Andrew Lines, and Mike Davies. Slack matching asynchronous designs. In *International Symposium on Asynchronous Circuits and Systems*, pages 184–194, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[15] Piyush Prakash and Alain J. Martin. Slack matching quasi delay-insensitive circuits. In *International Symposium on Asynchronous Circuits and Systems*, page 195, Washington, DC, USA, 2006. IEEE Computer Society.

[16] A. Yakovlev A. Bystrov. Fast four-phase tree fifo. In *Asynchronous Forum*, 1999.

[17] Rudolf H. Mak. A taxonomy of maximally elastic buffers. In *CS-Report 04-26*. Dept. Math. and Comp. Sc., 2004.

[18] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[19] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, S. Temple, J. D. Garside, and Z. C. Yu. Spa - a secure amulet core for smartcard applications. *Microprocessors and Microsystems*, 27 (9):431–446, October 2003.

[20] Hans van Gageldonk, Kees van Berkel, Ad M. G. Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *International Symposium on Asynchronous Circuits and Systems*, pages 96–107, 1998.

[21] A. Bink and R. York. ARM996HS: the first licensable, clockless 32-bit processor core. *IEEE Micro*, 27(2):58–68, March 2007.

[22] L. A. Plana L. A. Tarazona and D. A. Edwards. Architectural enhancements for a synthesised self-timed processor. In *Asynchronous Forum*, 2007.

[23] Akihiro Takamura, Masashi Kuwako, Masashi Imai, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *International Conference on Computer Design*, pages 288–294, 1997.

[24] Marc Renaudin, Pascal Vivet, and Frédéric Robin. Aspro-216: A standard-cell q.d.i. 16-bit risc asynchronous microprocessor. In *International Symposium on Asynchronous Circuits and Systems*, pages 22–31, 1998.

[25] Alain J. Martin, Mika Nystrom, Karl Papadantonakis, Paul I. Penzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, and Ahmet Tura. The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller. In *International Symposium on Asynchronous Circuits and Systems*, pages 14–23, 2003.

[26] A. Lines. The Vortex: A Superscalar Asynchronous Processor. In *International Symposium on Asynchronous Circuits and Systems*, pages 39–48, 2007.

[27] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, 1997.

[28] Cadence Design Systems TSMC Standard Cell Libraries, 1997-2007.