Adaptive Pipeline Structures for Speculation Control

Aristides Efthymiou   Jim D. Garside

Department of Computer Science, The University of Manchester,
Oxford Road, Manchester M13 9PL, UK
E-mail: {ae,jdg}@cs.man.ac.uk

Abstract

Pipelining is a common method for improving the throughput of a system, especially when the majority of the processing is sequential. Unfortunately when the sequentiality is broken, a pipelined system suffers additional delay and, most importantly for this work, energy waste which is roughly proportional to the pipeline depth. Standard pipelines cannot be modified once they are built so their depth is fixed. This paper proposes a method that allows the dynamic adaptation of the structure of an asynchronous pipeline, so that pipeline stages can be merged and split at run-time, allowing greater flexibility. It is based on novel latch controllers that can be configured dynamically as 'normal' or 'collapsed', i.e. keeping their latches permanently transparent. Using these controllers a model of AMULET3 was designed that is capable of changing its pipeline depth dynamically when branches are anticipated, in order to alleviate the energy loss when the branch finally arrives.

1. Introduction

Pipelining has long been used by engineers as an inexpensive method to improve the performance of systems such as processors. Its major disadvantage is control hazards, when the sequentiality of its operation is broken by a (taken) branch, when the pipeline is drained and later refilled. Apart from the well-known speed implications, this wastes energy for the instructions that have already entered the pipeline and are discarded.

Because branches happen quite frequently it has become standard, even for embedded processors, to employ branch prediction. Branch prediction generally predicts correctly most of the branches and, as it is usually not on the critical path, it has a positive impact on speed. However, the branch prediction hardware consumes energy in every cycle, so it is questionable if the total energy consumption for the execution of a program is reduced.

Generally pipelining can be considered as a form of speculative execution: instructions are fetched into the processor pipeline before knowing that they are going to be executed. For energy consumption, speculation is always wasteful compared to the ideal situation where the only instructions allowed in the pipeline are those that are executed. Primarily, energy is wasted in instructions that are fetched and processed, but are later discarded due to a branch. Prediction techniques can reduce this source of energy waste, but at the expense of a per instruction-fetch energy penalty, which is also wasteful.

Asynchronous pipelines [11] are capable of energy efficiency equivalent to perfect clock-gating without the problem of power supply ringing due to large variation in switching current found in synchronous circuits [7]. Despite their energy efficiency, the above mentioned problem of wasting energy for every branch is still present in asynchronous pipelines. Moreover the replacement of global synchronisation with local communication makes it harder to broadcast the occurrence of a branch from the stage that determines the next PC to its upstream stages, so that these can discard the instructions they are currently processing. To transmit this information would require separate handshakes with all upstream pipeline stages and arbitration (or synchronisation) in each one of them; as a result the pipeline throughput could drop. Thus the common practice is to let the upstream stages continue normally and discard the 'unwanted' instructions at the stage that resolves the branch. This clearly wastes more energy compared to a synchronous equivalent, since all instructions fetched after the branch will be processed in all pipeline stages up to the one that resolves the branches.

In summary, every instruction flow change incurs a significant energy cost for a pipelined processor. This cost is greater in asynchronous processors and it increases as the pipeline gets deeper.

This paper presents a method that can change the configuration of an asynchronous pipeline dynamically so that its effective pipeline depth can be controlled. It is based on selectively making some latches transparent which, in ef-

Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)
1522-8681/03 $17.00 © 2003 IEEE

fect, joins pipeline stages together. Shortening the pipeline depth, means that less energy is wasted when branches happen, but this comes at the expense of speed loss. The method presented here allows tradeoffs to be made not at the design time, as in common pipelines, but at run-time.

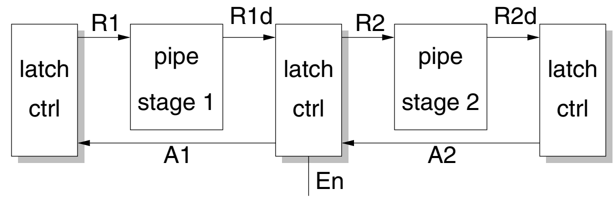
The next section presents the idea of dynamically making pipeline latches ‘permanently’ transparent. Section 3 describes common latch controller types and how their collapsed equivalents could be operating, while section 4 provides detail in the specifications for the new latch controllers. An application of the method is given in the remaining of the paper. A variation of AMULET3 [5] that uses the new latch controllers is described in section 6 and a method for determining dynamically when to reduce the pipeline depth is shown and evaluated in section 7. Finally section 8 concludes the paper.

2. Adapting the pipeline structure

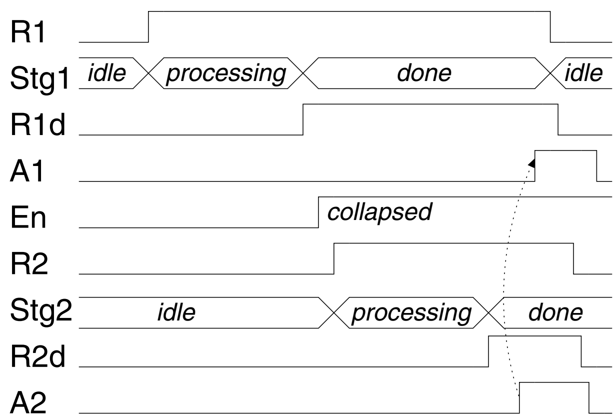
A pipeline is defined by the position of the pipeline registers/latches in the circuit. As there is no way to move these registers after the circuit is fabricated, the only alternative is to control when they are transparent or opaque. When a pipeline latch is made ‘permanently’ transparent, its two neighbouring pipeline stages are effectively joined in one stage. With this method the pipeline structure can be altered dynamically to suit the power and performance levels required by the system at any particular time.

The ability to join pipeline stages simply by making the latch between them always transparent is unique to asynchronous pipelines because the pipeline stages are self-contained and all communication is explicit. It is not possible to do the same in a synchronous circuit; control is global and correct operation relies on the delivery of results after a predetermined number of clock cycles. Even if this is managed — a tricky but soluble problem — a synchronous pipeline is normally balanced so that pipeline stages would have to be collapsed in (for example) alternate pairs accompanied by a simultaneous halving of the clock frequency. Thus collapsing in synchronous circuits would not be as general as in asynchronous ones.

In micropipeline-style asynchronous circuits, in order to be able to merge pipeline stages, reconfigurable latch controllers, that can be either ‘permanently transparent’ (*collapsed*) or ‘normal’, are needed. A collapsed latch controller must ‘pretend’ that it only connects the inputs and outputs together, as if they were the same wire, while always keeping the latch transparent. So, an input request will be passed on to the output, and will only be acknowledged when the output side has been acknowledged. In a series of collapsed latch controllers, the first will have to wait for the last one to be acknowledged (and propagated back) before it can give its acknowledgement.



(a) A 2-stage pipeline



(b) Collapsing pipeline waveforms

Figure 1. Timing of pipeline collapsing

The pipeline latches cannot be collapsed at any time. Figure 1 shows a two stage pipeline where the latch controller in the middle becomes collapsed and the signal waveforms at the time of collapse. The correct time to change from normal mode to collapsed is when a new input request has been received and the latches are about to be loaded. At this time the downstream pipeline stage (*stg2*) has finished processing the previous data and it is safe to merge the two pipeline stages. After the collapse the upstream stage is not acknowledged (*A1*) immediately, as it would normally be, because the new, joined pipeline stage has not finished processing the current data; its second half still has work to do. When the output acknowledgement (*A2*) is received, which means that processing at the second half of the stage has finished and the output data are safely stored at the next latch, the upstream latch controller is acknowledged, notifying the upstream stage that the whole pipeline stage is finished.

Splitting a previously merged pipeline stage also happens when a new request is received. After the latches have loaded the input data, the load enable signal is deasserted, making the latches opaque and acknowledging the input side. Concurrently the output request is issued as usual. The pipeline stages are now split and the first stage is free to receive the next data item for processing.

As the operating mode of a latch controller can only change at the time of new requests, the signal that sets the latch controller mode, *collapse*, must be locally synchronised with the input request signal. This makes it hard to have a global *collapse* signal if the latch controllers are to be reconfigured while the pipeline is operating. The solution is to have *collapse* bundled locally with the rest of the data signals that are latched at the pipeline stage. Section 5 investigates further on this issue.

3. Latch controller types

Latch controllers are an important element of the control part of an asynchronous processor. So it is not surprising that there is a great variety of latch controller types that can be built. This section considers the set of four-phase (return to zero) controllers implementing protocols used in AMULET3 [8].

Depending on how long the data are kept valid when the handshake signals return to zero, there are a number of handshake protocols and corresponding controllers. Two of the most commonly used protocols are discussed here (see figure 2):

Broad Data are kept valid until the acknowledgement becomes low.

Broadish Data are kept valid until the request becomes low.

Broadish is generally faster but assumes that the return to zero of the acknowledgement is 'dead' time for the downstream circuits. Some circuits need the data to be held for this time, in which case a broad protocol must be used. As different parts of a system have different use for the acknowledgement return to zero time, generally both types of controllers are used in a system.

For each of the above protocols there can be two variations depending on when the latches become transparent [8]:

Normally closed After the input request has been received and (obviously) before the input acknowledgement is asserted.

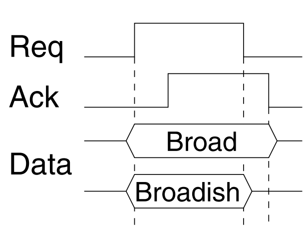


Figure 2. Broad, broadish protocol timing.

Normally open When the output side is not busy (which depends on the protocol) the latch becomes transparent, regardless of an input request.

Normally-closed latches isolate the downstream circuits from any spurious transitions while new data is expected but they suffer from the extra delay of having to open the latches after the data is ready. Thus normally-closed latch controllers lead to more energy efficient pipelines, while normally-open controllers lead to faster pipelines. To enable this trade off at run-time, reconfigurable latch controllers that can be either normally-open or closed have already been developed [8] and are incorporated in the design of AMULET3.

Another type of latch controller, early-open [9], uses a 'pre-request' signal, becoming valid some time before the input request, to open the latches just in time. As such a signal might not always be available, early-open controllers are not considered here. Nevertheless, when the pre-request signal is available, it is straightforward to apply the early-open concept to the latch controllers presented here.

3.1. Collapsed latch controller types

This section determines if the protocols described above are still meaningful when the latch controller is collapsed. Since the broad and broadish protocols differ only in the way the return to zero of the acknowledge signal relates to the data validity, the following discussion focusses only on this phase of the protocol.

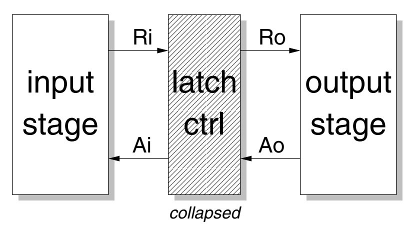


Figure 3. Pipeline with collapsed broad latch controller.

For broad protocol, the output stage expects the data to be kept valid until it has returned the input acknowledgement (Ao, figure 3) to zero. As the latch controller is collapsed, its latch(es) will always be transparent, so it cannot directly guarantee the above condition. Thus this requirement must be guaranteed by the upstream stage. If the upstream stage adheres to the broad protocol, it will not change the data until *its* output acknowledgement (Ai) has returned to zero. Consequently, for correct operation, the collapsed latch controller must not allow its input acknowledgement (Ai) return to zero until the output acknowledgement (*Ao*) has; in other words the controller behaves as if *Ai*, *Ao* were the same wire.

For the broadish controller type, the return to zero of the input acknowledge (*Ai*) can happen at any time after the input request becomes low, as its return to zero is not related to the data validity. So the collapsed, broadish controller can take a short-cut and allow the return to zero of the input acknowledgement right immediately the input request has fallen regardless of the state of the output acknowledgement. In this case the behaviour is different: *Ai* and *Ao* do not have to appear as if they were on the same wire.

3.2. Normally open/closed controllers

By definition a collapsed latch should be always open to let data pass through as if it weren't there. So it may seem that the normally-closed variation is of no use when the latch controller is collapsed.

In reality the unwanted energy consumption caused by glitch propagation [1] — which triggered the idea for normally closed latches — still happens with collapsed latches. Moreover it is well known that glitch-induced energy increases with the logic depth between latches [2], which is precisely what happens when collapsing a pipeline latch. So the benefit of reducing speculative operations by limiting the pipeline depth may back-fire because of the increase in the energy consumed by glitches.

Having a collapsed latch controller which only opens when the request input is ready would help, because the request is asserted only when the data are ready; all intermediate values are stopped from propagating down the pipeline. Obviously in this case the latch itself is not really collapsed, but the controller still gives this impression, so the term "collapsed" is still used here.

From the above discussion, all four types of latch controllers presented above are still meaningful when they are collapsed, thus circuits for all of them need to be built. The reconfigurable normally open/closed operating mode [8], is an attractive feature, so it is retained in the proposed latch controllers. The next section defines two (collapsible) latch controllers (for the broad and broadish protocols) that are configurable as collapsed or normal and as normally open or closed.

4. Collapsible latch controllers

A convenient way of describing the operation of asynchronous circuits is by using state transition graphs (STG) [10]. Figure 4 shows the STGs for normal (a-d), collapsed (e-h), normally open and closed configurations for both broad and broadish protocols.

A brief description of the operation of the broadish latch controller is given below in both normal and collapsed operating modes, based on the STGs. The broad protocol is the same with the exception that the latch enable signal is only allowed to rise after both output handshake signals have returned to zero.

4.1. Non-collapsed

In the quiescent state of normal, non-collapsed operating mode (fig.4(a,b)), all signals are low, except for *Na* which is high and *En* which depends on the normally open or closed condition. *Na* is low when a request has been received but not acknowledged yet. In normally-closed mode, an input request (*Rin*) causes *En* to rise, making the latches transparent. The rising *En* causes *Na* to fall, which turns *En* back low and makes *Rout* high, propagating the request downstream. After the latch has closed, *Ain* is asserted to acknowledge the input. *Rin* could then fall which resets *Na* to its quiescent value of 1, causing *Ain* to return to zero. At the output side, the raised *Rout* will eventually be acknowledged by a rising *Aout*. Then *Rout* falls re-enabling *En* to rise, when the next input request arrives. The only difference in normally-open mode is that *En* is set back to 1 whenever *Rout* falls, regardless of the state of *Rin*.

4.2. Collapsed

In collapsed, normally-closed mode (fig.4(f)) the operation is the same as above except for the rising transition of *Ain*. This happens whenever *Aout* rises. At that time *Rin* is still held high, so *Na* is low. *Rin* can now fall which makes *Na* high, turning *Ain* low again. The high *Na* combined with *Aout* will make *Rout* low. *Na* cannot fall again in response to a new input request until *Aout* has returned back to zero. In normally-open mode (fig.4(e)), *En* is forced high continuously, so *Na* does not depend on it. All other operations are exactly the same.

The STGs of the collapsed controllers have only two added arcs compared to the normal controllers. By the definition of the collapsed controller, arc *Aout*+ $\rightarrow$ *Ain*+ is added, so that the input can only be acknowledged when the output is acknowledged. The corresponding arc for the falling edge is only needed when the protocol is broad. For the broadish version an arc from *Aout*- to *Na*- is added instead. This makes sure that if a new input request is received (*Rin*+) while the output acknowledgement has not fallen yet, *Ain* will not be set high, mistaking *Aout* for a new acknowledgement. When the latch is collapsed and normally-open, the load enable is always on, so it is not shown in the STGs. It should be noted that a number of arcs in the collapsed STGs (4(e-h)) are not necessary but are

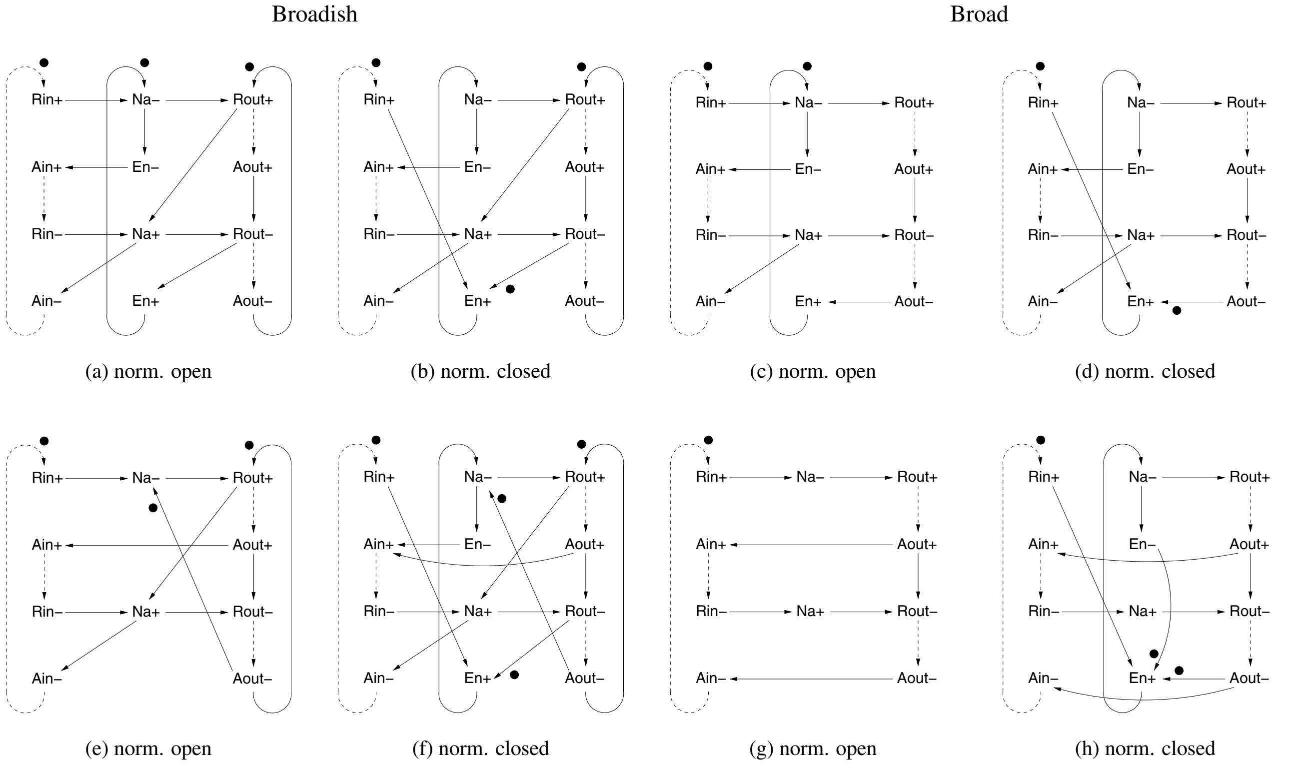


Figure 4. STG's of all latch controllers: (a-d) non-collapsed, (e-h) collapsed.

retained to show the similarities to the equivalent normal STGs (e.g. *Aout*+ → *Rout*-).

The collapsible latch controller circuits are described below. They are based on the existing latch controllers by Lewis *et al.* [8].

Figure 5 shows the proposed latch controller for the

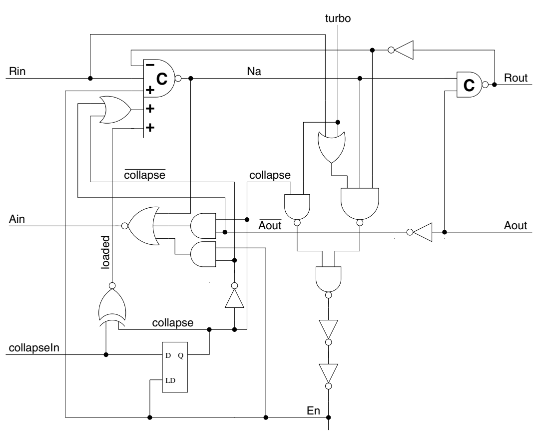


Figure 5. Collapsible broadish latch controller.

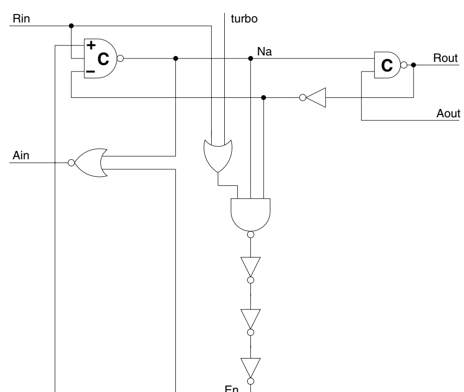


Figure 6. Existing broadish latch controller.

broadish protocol (figure 6 shows the existing controller). It is designed by combining the synthesised STGs shown earlier. The only circuits added to the synthesised circuit are the latch holding the *collapse* signal, the XNOR gate that drives *loaded*, and the extra + input of the C element producing *Na* that is driven by *loaded*. *Loaded* becomes low when a change in the configuration is going to happen and rises again when *collapse* is loaded with the new value.

Its connection to the C element, prevents *Na* from falling — and thus any other action in the controller — before *collapse* and the gates it drives are ready.

Figure 7 shows the proposed circuit for the broad protocol (fig. 8 shows the existing circuit). As with the broadish circuit, the latch and the XNOR producing *loaded* are used. In this case a C element is added to make sure that *Ain* is only allowed to rise when *collapse* has been loaded and the gates it drives are ready.

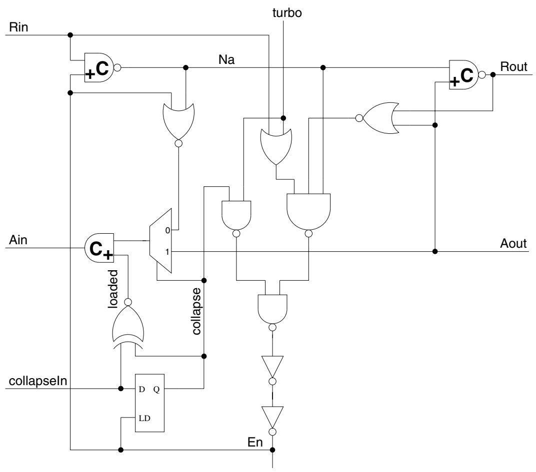


Figure 7. Collapsible broad latch controller.

The STGs in figure 4 were synthesised independently using Petrify [3] and the resulting equations were modified slightly to be useful in all operating modes. The final circuits are speed independent in either operating mode. During the transitions between the collapsed and normal modes though, speed independence cannot be guaranteed. For this reason a few gates are added to ensure smooth transition between the operating modes with some timing assumptions. These proved easy to guarantee and transistor-level simulation in a 0.18um process showed that the latch controllers are operating (and switching operating modes) correctly in all process corners. The extra circuits added have an effect on the speed of the controllers which have a 14% to 40% slower minimum cycle time (in non-collapsed mode) compared to the existing controllers.

5. Controlling the pipeline collapse

As explained earlier, *collapse*, the control signal setting the operating mode of the collapsible latch controllers, must be locally synchronised, i.e. bundled with the latch input data. Thus a global *collapse* signal cannot be used. A possible implementation would be to have the first pipeline stage produce the *collapse* signal for the first latch controller, which is then propagated to subsequent controllers

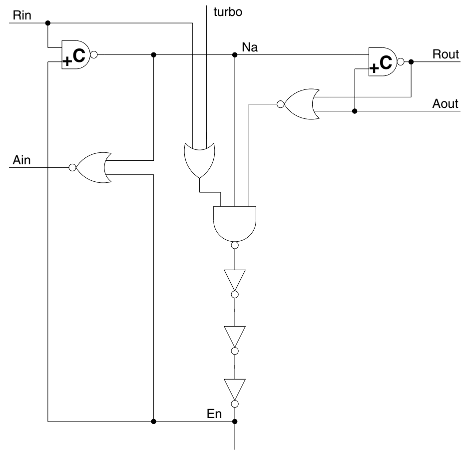
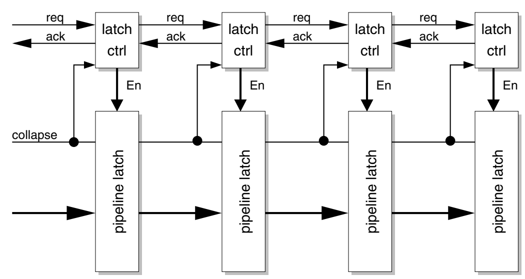
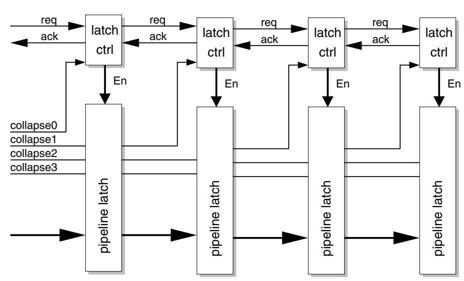


Figure 8. Existing broad latch controller.

down the pipeline (figure 9(a)). On route, depending on local conditions, the collapse signal can be changed by the pipeline stages. Alternatively, for independent control of each latch controller, the first stage can produce a *set* of collapse signals which are latched and propagated to the appropriate latch controller (figure 9(b)). In both cases the first pipeline stage is the most suitable to generate the signals since, in asynchronous pipelines, it is easy to transmit signals following the pipeline flow but considerably harder in any other direction.



(a) Serially controlled



(b) Individually controlled

Figure 9. Collapsible signal distribution.

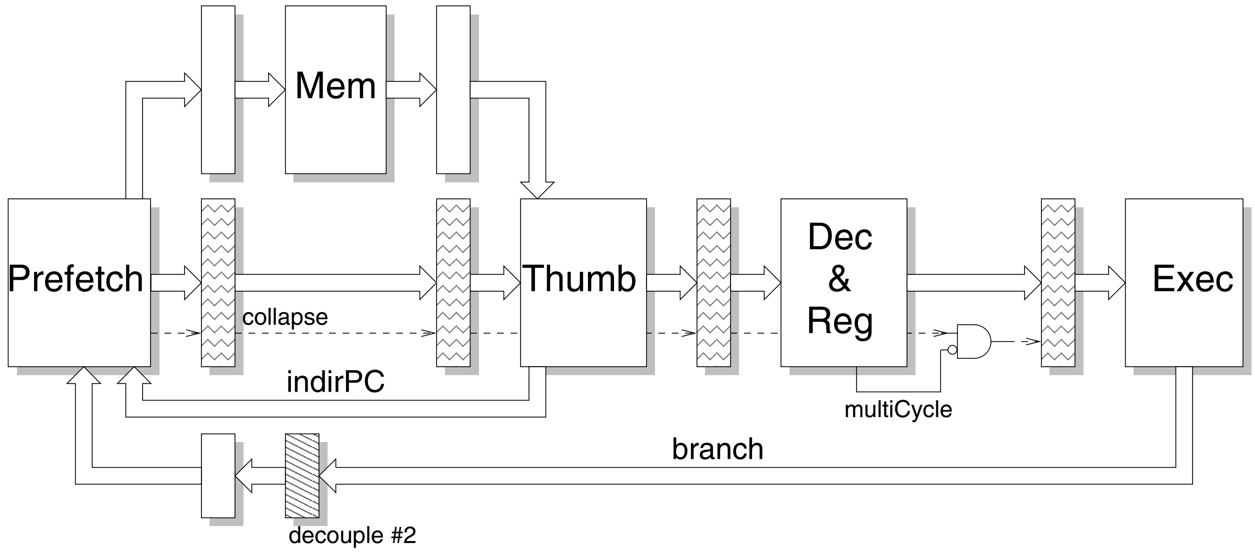


Figure 10. Block diagram of AMULET3 with collapsible pipeline latches.

6. Integration in the processor

To evaluate the new latch controllers, AMULET3 was modified to use them. This section describes the modified processor core and the issues that were taken into consideration.

Figure 10 shows an abstract block diagram of the AMULET3 core showing which pipeline latches were changed to use collapsible controllers. Only the latches marked in the figure with grey waves need to be made collapsible; all others, including those not shown in the figure, are unaffected.

As can be inferred from the way the collapse signal is drawn in the block diagram, there is only one such signal propagating from the prefetch stage down the pipeline to the execute stage, in the same fashion as explained earlier in figure 9(a).

Nearly all the pipeline latches of the processor are modified to be of the collapsible type. A notable exception is the decoupling latch, on the branch path from execute to prefetch, which must not be collapsed. If it were, in a configuration where all the latch controllers are collapsed, branches would deadlock: the decoupling latch would try to write the target address into the PC in the prefetch block, but that would not be allowed until the instruction, i.e. the branch, is acknowledged (completed). To break the deadlock, a second decoupling latch is added in the branch channel. This latch can be a standard latch, but then the branch latency will be increased, which is undesirable. Alternatively it can be a collapsible latch, which works in the opposite way to the other latches in the pipeline: it is collapsed when there is at least one other latch in the pipeline that is not collapsed. In other words, it works as a latch only when all the other latches are collapsed, which is when the above mentioned deadlock could happen.

In AMULET3 some instructions (e.g. long multiplications) require multiple execution cycles to complete. When the pipeline stage between decode and execute is collapsed, these instructions could not be executed. This is where the flexibility of generating a local *collapse* signal is valuable. For these instructions the collapse signal for the decode-execute pipeline latch is temporarily disabled using the AND gate in figure 10. This re-instates the pipeline stage between decode and execute for the duration of the multi-cycle instruction, so that multiple execution cycles can be performed as required.

With the applied changes the processor can be configured to have any pipeline latch collapsed and the operating mode of each latch controller can be changed for each new input request if needed. The processor can operate in the full range from top-speed, fully-pipelined to energy-conserving single-stepping each instruction. An evaluation of the processor's speed and energy consumption for each pipeline occupancy level is given in [4], where a set of benchmarks was run at each pipeline occupancy level. Collapsing the whole processor gives up to 15% of energy savings, at the cost of approximately halving the speed compared to the fully-pipelined processor.

7. Dynamic pipeline-depth adaptation

Using the collapsible latch controllers to set a constant processor pipeline depth for the duration of a program execution, as shown above, is the most conservative way to use this new technique. In this section a dynamic approach is used, where the whole pipeline is collapsed when there is an indication that a branch may be imminent, so that the energy waste of prefetching instructions beyond it is eliminated.

Since over 80% of the branches are conditional [6], the instruction that generates the branch condition could be used as a hint that a branch is approaching. AMULET3 implements the ARM architecture which uses condition codes to specify the branch condition. Thus a technique that can be applied is to start single-stepping the processor when an instruction that changes the condition codes is detected, as there is a high probability that it is closely followed by a conditional branch. When the first instruction from the branch target is fetched, the processor will resume its normal operating mode. For processors that do not use condition codes, comparison-type instructions could be detected instead.

Detecting instructions that change the condition code is simple in the ARM instruction set architecture as there is a specific bit in the data-processing instruction format which controls this. It is important that the detection occurs as early as possible in the pipeline; the number of stages between *prefetch* and the detecting stage determine how many instructions following the one that sets the condition code have already been fetched and thus how much energy might be wasted. In AMULET3 the detection can be done at the first decoding stage (Thumb in figure 10), so only one instruction may be already fetched before the detection happens. This is very useful as over 50% of branches in the benchmarks used here, immediately follow the instruction (usually compare - CMP) that sets the condition codes.

In order to detect the condition-setting instructions as early as possible, a more general opcode pattern could be used for the detection which matches more instructions than just those setting the conditions. This will obviously put the processor into single-stepping mode for more of the program execution, slowing it down but, if it allows the detection to be done at an earlier stage, it could save an extra instruction fetch per taken branch.

As an alternative to detecting the condition-setting instruction as early in the pipeline as possible, an optimising compiler could be used to insert some 'neutral' instructions between the branch and the condition-setting instruction. This would make the hardware implementation simpler and could allow enough time to detect the condition-setting instructions in instruction-sets with complex encoding. Modifying the existing compiler was not possible for this work, since the source code is not available. Thus the following design of this technique tries to detect condition-setting instructions as early as possible.

7.1. Design

When an instruction that changes the condition code is detected at the Thumb stage, the whole processor should become just one pipeline stage, by collapsing all the pipeline latches. As explained in section 5 this can only happen gradually; as the condition code setting instruction moves down the pipeline it collapses the pipeline latches on the way. That would still leave the processor a two stage pipeline, *prefetch* and the rest of the stages joined, so one instruction following the branch would be fetched and wasted if the branch is taken.

In order to save this instruction too, the information that a condition-setting instruction has been detected must be sent back (e.g. counter-flowed) to the prefetch stage, so that the remaining latch controller can be collapsed as well. The simplest and safest way would be to pass this information the next time the two stages 'synchronise', when the next instruction is passed from *prefetch* to *Thumb*. This will collapse the *prefetch* to *Thumb* latches (when they next get loaded), making the whole processor a single pipeline stage. Thus *prefetch* will not fetch another instruction until the execution of the one following the detected, condition-setting instruction is complete. Consequently, even if a branch follows immediately after the condition-setting instruction, no extra instructions will be fetched.

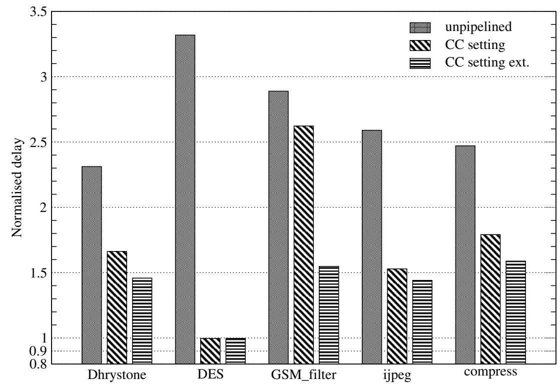
When the first instruction from the branch target is fetched, it resets the latches to their normal operating mode, as it travels down the pipeline, returning the processor to its normal, fully-pipelined operating condition.

In the ARM instruction set architecture all instructions can be conditional (not only branches), so the above design might be too conservative. For this reason a variation (CC setting ext) was designed and evaluated that detects instructions that are conditional but not branches and forces the processor to operate in fully-pipelined mode. Thus when the condition code is set, the processor does not have to operate in single-step mode until a branch occurs. This method is expected to improve the execution delay without compromising the energy savings achieved.

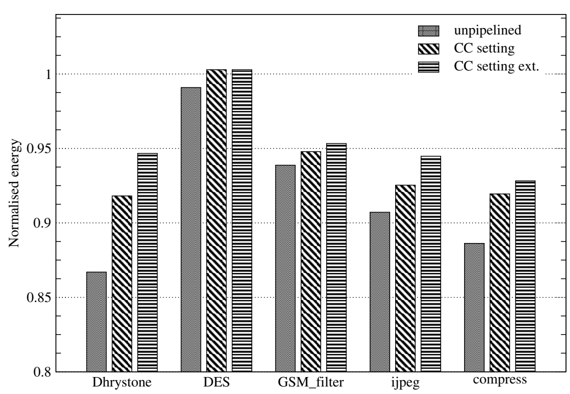
7.2. Evaluation

The condition-code setting detection techniques require sufficiently small changes to the processor that can be evaluated using a simple evaluation methodology. Two Verilog models were produced and simulated running a set of five benchmarks: Dhrystone, DES encode, GSM filter, compress, ijpeg. Energy estimation is achieved by counting the number of toggles for almost all of the circuit nodes and multiplying with the capacitances extracted from the AMULET3 netlist.

Figure 11(a) shows the execution delay of each benchmark for the two variations described above and the unpipelined version (all pipeline latches are collapsed), normalised relative to the fully-pipelined version. Figure 11(b) shows the normalised energy consumption for the same programs and processor variations.



(a) Delay



(b) Energy

Figure 11. Evaluation results for condition code setting.

For *DES encode* there are insignificant energy savings with any speculation control technique, as there are very few branches and thus almost no speculative instruction fetches. Thus, for this benchmark, the interest is in the execution delay overhead of the method tested. In this case both condition-setting detection variations managed to keep the execution delay to the levels of the fully-pipelined version. In contrast the unpipelined version is over three times slower.

The difference in the two variations can be seen in a benchmark like *GSM filter*. The execution delay of the first variation is very close to that of the unpipelined processor, while the execution delay of the second variation is almost half of the first. As the *GSM filter* code has many conditional data-processing instructions, it clearly benefits from restoring the fully-pipelined mode early, as is done by the second variation.

Generally, both variations managed to reduce the execution delay compared to the unpipelined processor, for a small increase in energy consumption. This shows that the dynamic approach was quite successful. The second variation is consistently faster than the first, but it also consumes more energy. It is very useful for benchmarks that have a large number of data-processing conditional instructions, such as *GSM filter*.

8. Conclusions

A novel method for dynamically adapting the structure of a pipeline is presented. It is based on new latch controllers that can be configured to be collapsed, i.e. behave as if they do not exist, while keeping the latches they control 'permanently' transparent. These collapsible latch controllers can be used to control the occupancy and the throughput of a pipeline and reduce the wasted energy when branches occur.

As an application, AMULET3 was modified to use the new latch controllers. Whenever an instruction that sets the condition code is detected, the whole pipeline is collapsed to a single stage, so that if there is a conditional branch following that instruction, no energy will be wasted fetching and processing instructions following the branch.

Experiments illustrate that this strategy can save significant energy if timing requirements are not strict. The behaviour depends strongly on the type of code, as evidenced by the variation in the behaviour of different benchmarks; thus the facility to allow dynamic control of the pipeline structure — even partially under software influence — may be very valuable.

References

[1] L. Benini, M. Favalli, and B. Riccò. Analysis of hazard contribution to power dissipation in CMOS IC's. In *Proceedings of the International Workshop on Low Power Design*, pages 27–32, May 1994.

[2] E. Boemo, S. Lopez-Buedo, C. Santos Perez, J. Jauregui, and J. Meneses. Logic depth and power consumption: A comparative study between standard cells and FPGAs. In *Proceedings of the XIII Design of Circuits and Integrated Systems (DCIS) Conference*, Nov. 1998.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous con-

trollers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, Mar. 1997.

[4] A. Efthymiou and J. D. Garside. Adaptive pipeline depth control for processor power-management. In *Proceedings of International Conference on Computer Design*, pages 454–457. IEEE Computer Society Press, Sept. 2002.

[5] J. D. Garside, S. B. Furber, and S. Chung. AMULET3 revealed. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, Apr. 1999.

[6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[7] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12. IEEE Computer Society Press, Apr. 2002.

[8] M. Lewis, J. D. Garside, and L. Brackenbury. Reconfigurable latch controllers for low power asynchronous circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 27–35, Apr. 1999.

[9] P. Riocreux, M. Lewis, and L. Brackenbury. Power reduction in self-timed circuits using early-open latch controllers. *IEE Electronics Letters*, 36(2):115–116, Jan. 2000.

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

[11] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

1522-8681/03 $17.00 © 2003 IEEE