In *12th Asynchronous UK Forum*, London, June 2002

Pipeline Occupancy Control for Power Adaptive Processors

Aristeidis Efthymiou
efthym@cs.man.ac.uk
Department of Computer Science, University of Manchester

May 2002

1 Introduction

This communication describes a mechanism for controlling the pipeline occupancy of an asynchronous processor, aiming to produce a power adaptive processor using only micro-architectural modifications.

A power-adaptive processor scales its power consumption and performance according to the task it is executing and the available power in the system. If there is enough power to run the processor in its full capacity and the task being executed has hard completion deadlines, there is nothing to do. But if the available power is relatively low, the processor should work as fast as it can with the power it has. Alternatively in an operating environment where the processor has plenty of *slack* time, the execution can be slowed down, by reducing the supply voltage or the switching activity to save more power than what could be achieved by just slowing it down. Dynamic voltage scaling [Fle00],[BPSB00] is a suitable candidate for this work, but it suffers from very slow transition times from state to state and requires specialised DC-DC converters for its power supply and also modified clock generators. For this reason architectural techniques are being investigated. The method presented here is to control the occupancy of the processor's pipeline as a method of controlling how much it "speculates" and thus save the energy spent in fetching and decoding instructions that are not executed.

The main idea is described in the following section and the changes to AMULET3[GFC99] to incorporate this mechanism are shown in section 3. Circuits to add/remove tokens are presented in section 4 and simulation results for a number of benchmarks in section 5.

2 Pipeline occupancy control

Controlling the occupancy of a purely serial pipeline, like that of figure 1, is easily done using a FIFO of tokens placed next to the original pipeline. The first stage, must acquire a token before it can operate and send the data to the next stage. On the other end of the pipeline, the token is returned to the FIFO when the operation is complete. The number of tokens that are present in the system determine the maximum occupancy of the pipeline (assuming that there are not going to be more tokens than the actual number of pipeline stages). So this mechanism gives the opportunity to vary the number of concurrent operations in the pipeline from none (stalled) to its maximum capacity.

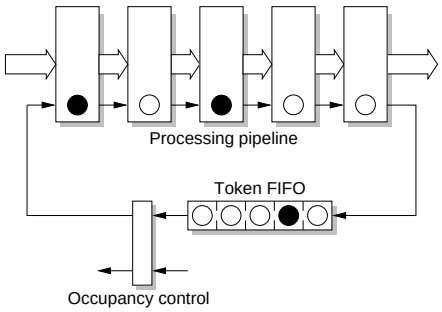

Figure 1: Token controlled serial pipeline.

3 Token FIFO in AMULET3

If the token controlled pipeline is applied to a single issue processor, like AMULET3, the stage that releases the tokens is the execute stage. Although there is one extra stage after execute (or more for multiple data transfers) there is no need to delay returning the token until the very end of the pipeline. The main reason is that conditional instructions and, most importantly, branches are resolved at the beginning of the execute stage, so returning the token later than this stage will save no extra energy. For a lower *power* consumption, the tokens could be returned at the register write-back stage, but this idea is not pursued further here.

For correct operation, mostly in the single occupancy configuration, the token delay through the token FIFO, should be longer than the delay for the branch target to reach the fetch unit. Other-

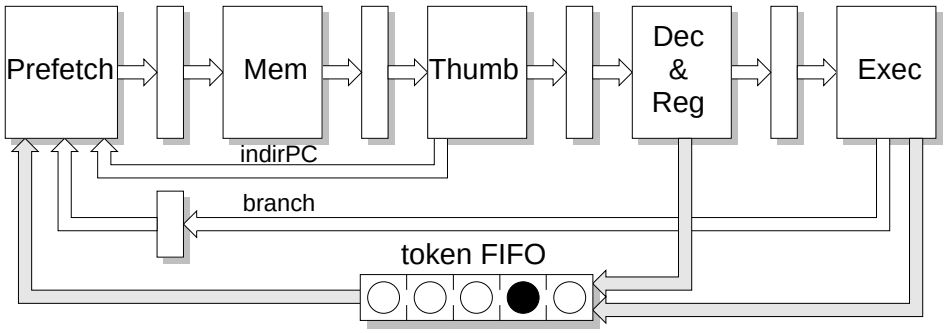

Figure 2: Token controlled AMULET3 processor core.

wise, another instruction from the shadow of the branch will be fetched instead of the branch target as would have been expected for a pipeline with an occupancy of one. To ensure this, the execution stage is made to complete its handshake with the fetch stage, before the token is released.

The implementation of pipeline occupancy control using tokens in AMULET3 posed some complications in three areas: handling the tokens taken by instructions that are discarded before reaching the execution stage, multi-cycle instructions, and the processor's peculiar indirect PC load method. All of these were easily solved with a low implementation cost.

Discarded instructions

The case of discarded instructions is quite interesting. In a synchronous processor, the common practice is to add a "valid" bit with the data and pass it from pipeline stage to the next. If this signal is available early enough, the pipeline register can be clock-gated to save energy. In an asynchronous processor, discarded instructions are never passed on to the next stage, which can be more energy efficient. With the introduction of the token mechanism, this is no longer possible, because the token acquired at the fetch stage must be returned, or the processor will eventually starve of tokens. In AMULET3, the earliest that an instruction can be discarded is at the decode stage, so the solution was to add a new handshake channel between the decode and execute stages to pass on the tokens of the discarded instructions.

Multicycle instructions

In AMULET3 there are two types of instructions that take more than one cycle to execute: long multiplications and multiple data transfers. Multiple data transfers, use the execution unit only for one cycle, to check their condition and calculate the first address. The PC can be one of the registers to be loaded, causing a branch. This is determined at the first cycle, and the appropriate address, if any, is given to the fetch unit as a special case of branch (see also indirect PC loads, below). Although the instruction may take a number of memory accesses to finish, the only time it can change the instruction flow is at this first cycle. Thus the token is returned at this time.

Long multiplications take two cycles in the existing processor. A multiplication cannot change the instruction flow of the processor (the PC is not allowed to be the destination register) so the token is returned at the first cycle, without risking a speculative fetch. The second execution cycle is not allowed to pass another token.

Indirect PC loads

The implementation of loading the PC from the memory is peculiar to AMULET3. Instead of performing a standard data load and then pass the result to the fetch unit, the address is given to fetch, which does the read from the instruction memory port, and then reuses the data value as an address to load the next instruction. This was implemented mostly to speed up multiple load instructions, which are frequently used to restore the registers and the PC from the stack, when returning from a function call. The PC can be loaded concurrently with the data loads and the instruction flow from the target address can be restored quickly.

For the token FIFO method, only the first fetch "cycle" acquires a token. The second cycle reuses the previous token, which then escorts the instruction in the usual manner. This is implemented by deferring the acknowledgement of the token until the second memory cycle is complete.

Figure 2 shows a simplified block diagram of the processor with the token FIFO. The block arrows are handshake channels, and the direction of the arrow is the same as the direction of the request

signal. The only change to the simplified linear pipeline example of figure 1, is the channel between decode and execute stages. In addition, some minor changes in the control units of fetch, decode, and execute were made to incorporate the new handshake channels.

4 Adding/Removing tokens

The abstract figure 1 describing the token FIFO concept showed a block to add and remove tokens in series with the token FIFO itself. Figure 3 shows the circuit that removes tokens. Adding tokens is done similarly.

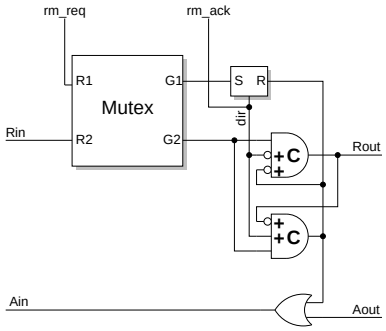

Figure 3: Token removing circuit.

In order to remove a token, all that is needed is to acknowledge an incoming request without passing it on to the next stage. Because the request to remove (or add) a token can coincide with the arrival of the input request a mutual exclusion element is included in the circuit. When the remove request (rm_r) wins the arbiter, it sets a "direction" flip-flop (dir). The flip flop selects which direction the incoming request will follow: either to the output request ($Rout$) or to the input acknowledge (Ain). In normal operation the output request direction is followed. When a token is to be removed, the flip-flop is set and the next request will be routed to the input acknowledgement wire. When this happens the direction flip-flop is reset, so that the following requests will pass through normally.

5 Evaluation

The pipeline depth control methods were evaluated by simulating a mixed behavioural/structural Verilog model of the processor, using NC-Verilog. A custom VPI module was linked with the simulator to count the toggles of a large number of nodes. The number of toggles was then multiplied by the capacitance of each node including both parasitic and transistor gate/drain capacitances, which were extracted from the original processor. As the modifications made to the base processor are minor, their own effect on the energy consumption is ignored in the measurements.

In addition to the usual speed, power and energy metrics, the notion of "extra work" defined by Manne *et al.* [MKG98], is used. This shows in each pipeline stage how much extra work is performed, measured in instructions processed at the pipeline stage, relative to the number instructions that were finally executed, i.e. those that passed their condition code check. This metric was added to give an extra indication of how much energy can be saved, given that the energy/power measurements are done with high level simulation, and thus cannot be accurate.

The benchmarks used comprise of Dhrystone, a FIR filter taken out of an implementation of GSM encoding, a DES encryption program and two SPECInt95 programs: compress and ijpeg. All are written in C and compiled with speed optimisations enabled, using the compiler provided with the ARMtools 2.51. The input size for all benchmarks was small, so that the simulation can complete in reasonable time: up to 24 hours in a Sun Ultra 5. For the same reason ijpeg was only allowed to do one cycle of compression and decompression.

Figure 4 shows the normalised power, energy, and execution time for each pipeline occupancy, with disabled branch prediction, for a number of benchmarks. The pipeline occupancy was kept constant for the whole duration of execution of each benchmark. The energy results show how much extra power was saved compared to simply slowing the execution without any architectural adaptation. Figure 5 shows the extra work per pipeline stage, for each occupancy.

It is clear that only when the pipeline occupancy is low (1 or 2) there are reasonable energy savings. DES encryption has only a few branches, so the extra work in any pipeline stage is insignificant and the same holds for the energy benefits through pipeline adaptation. The GSM filter was optimised by the compiler so that many branches were replaced by conditional instructions, which is why there is a lot of extra work at the execute stage: about 30% of the instructions fail their condition test. The other benchmarks have a significant amount of extra work, which was successfully removed by lowering the pipeline occupancy and this is reflected in the energy graphs as well.

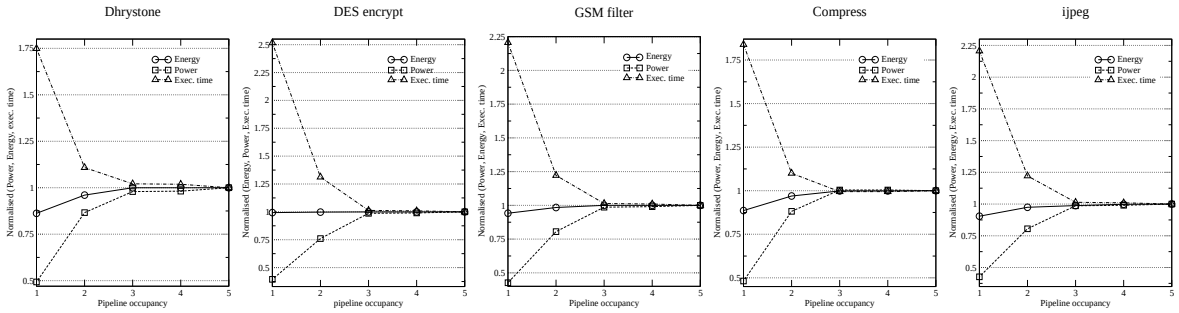

Figure 4: Results (energy, exec. time) of token based pipeline occupancy control

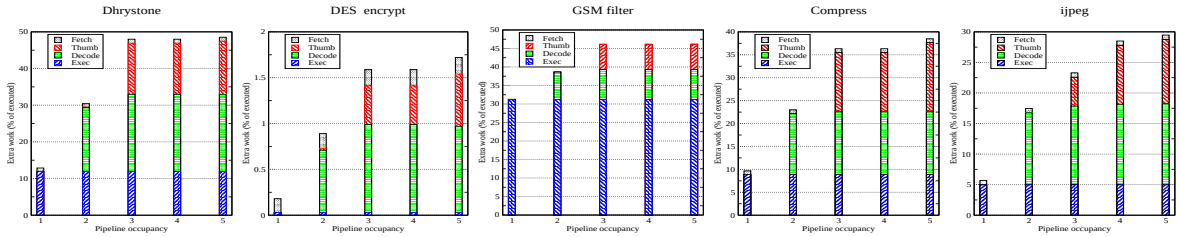

Figure 5: Results (extra work) of token based pipeline occupancy control

6 Conclusions

A novel method of managing the power consumption of a single-issue processor, by controlling its pipeline occupancy, is proposed. Decreasing the pipeline depth slows down the processor, but also saves energy because fewer 'speculative' instructions are fetched and decoded. In addition, hardware that supports pipelining, like branch prediction, can be turned off as it is not necessary in shallow pipelines. The power saved with this method is more than that which would be saved by merely slowing down the processor to achieve an equivalent performance, because of the lower switched capacitance per cycle.

A technique was presented to control the pipeline depth, using an existing asynchronous processor as a base. It uses token passing to control the pipeline occupancy and, indirectly, the pipeline depth. Simulating processor models employing the proposed token FIFO technique shows an energy reduction of up to 16% compared to the base system, using a collection of five benchmarks.

7 Acknowledgements

Thanks to Jim Garside for his valuable help in many aspects of this work. The author is funded by a scholarship from the Department of Computer Science, University of Manchester; this support is gratefully appreciated.

References

[BPSB00] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. In *Dig. of Technical Papers International Solid-State Circuits Conference*, pages 294–295, February 2000.

[Fle00] Marc Fleischmann. Crusoe power management: Cutting x86 operating power through LongRun. In *Symposium Record Hot Chips 12*, August 2000.

[GFC99] 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, April 1999.

[MKG98] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the International Symposium on Computer Architecture*, pages 132–141. ACM Press, June 1998.