Dynamic speed/voltage scaling for GALS processors

Shelley Chen Anand Eswaran

Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA 15213
Email: {schen1, aeswaran}@andrew.cmu.edu
http://www.ece.cmu.edu/~schen1/ece743

Abstract

Dynamic voltage scaling (DVS) has emerged as a successful and scalable solution to deal with the growing power consumption associated with increased chip complexity. We describe two schemes that allow the extension of DVS across multiple clock domains specific to GALS out-of-order superscalar processors. One scheme addresses the issues involved in the commonly shared front end of the pipeline. The other enhances the effectiveness of voltage scaling within the various functional units of a superscalar processor by addressing dependency issues. We plan to implement our design on simGALS [1], figure shown at right.

1 Introduction

Every generation, CMOS transistors are getting smaller in area, allowing processor designers to increase the complexity of a processor chip. A detrimental fall-out of this increasing complexity and integration in modern microprocessors is the fact that power density is rising at a significant rate. One significant component of the power budget is clock power.

As processors become more and more complex, the complexity of the interconnects increases significantly. Since most processors today are single clock driven, the clock signal must be propagated to the furthest parts of the chip without increasing clock skews. However, chips today have become so complicated and clocks frequencies are so high that the effects of clock skew, though small, would have a significant effect on the functionality of the processor. Fortunately, GALS, Globally Asynchronous, Locally Synchronous, processors have emerged as a good solution to the clock skew problem. The processor chip is divided into smaller clusters, each cluster having its own clock.

An added incentive of having the processor split into separate clusters, each running on their own clock, is that this gives each cluster the freedom to have its clock frequencies and voltage sources independently manipulated. These conditions will be discussed later in this paper.

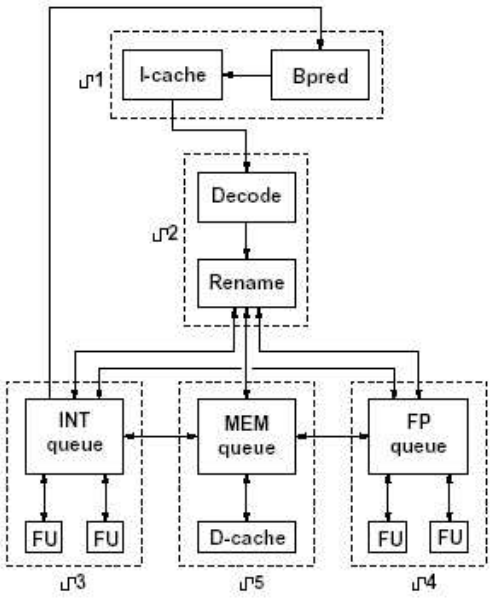


Figure 1: High-level block diagram of the GALS architecture. The sections encapsulated by the dotted lines represent independent clock domains.

We investigated two solutions for reducing power consumption in the pipeline. One solution targets the commonly shared front end of the pipeline (the fetch and decode stages). The second is an extension of [1], to accommodate issues like data dependencies among the functional units in the execute stage.

2 Previous or Related Work

In all data flow paths other than the critical path, the locally-synchronous blocks can be slowed down, thus producing significant power savings. This is because the energy consumption in CMOS is proportional to the square of the Vdd. Our work aims at the development of a GALS architecture that can dynamically adapt the supply frequency and

voltage delivered in a particular window of execution in order to achieve maximal utilization of power savings without significantly affecting performance of the machine.

The implementation issues involved in power saving mechanisms using dynamic clock management for high-performance GALS out-of-order superscalar processors have been discussed in [1] and [2] . There exists an inverse relation between supply voltage and logic delay due to switching capacitances involved. Thus any change in voltage needs to be accompanied by a proportional change in the frequency. Power dissipated in the chip is quadratically proportional to the supply voltage and hence reduction of Vdd would result in significant power savings.

The implementations of current architectures of GALS out-of-order superscalar Processors [1],[2],[3],[4] performs no voltage (and hence frequency) scaling for the fetch, decode and other front-end stages on the assumption that slowing down these stages might be detrimental to net throughput of non-blocking stages further down the pipeline. Most current implementations are based on some mechanism by which the queuing buffers and asynchronous FIFOs that lie in between two differently-clocked stages of a GALS processor informs the producer or receiver to slow down the clock frequency.

Baniasadi and Moshovos [5] discuss throttling of the fetch and decode stages accordingly with the commit rate. In addition, another technique that they introduce is throttling of the front end when the number of instructions waiting for dependencies exceeds a pre-defined threshold. These techniques are meant to complement existing speculation-based confidence methods.

3 Paper Overview

This paper is organized as follows. In Section 4, we talk about some implementation details concerning the different issues that we proposed to attack in this paper: front end throttling and data dependencies. Section 5 describes the simulator that was used for the testing and verification of the results. Section 6 describes the baseline model that we used for comparison to the modified GALS processor. In addition, it summarizes the results obtained from implementing the new techniques. Section 7 draws some conclusions about the effectiveness of the techniques introduced. Section 8 focuses on possible future work that can be done and how to extend our research further.

4 Implementation Details

This section describes the steps that were taken to implement the dynamic scaling of the fetch stage and the dependency check for the execute stages of the pipeline. The details about the algorithms implemented are described as well as the hardware requirements for each solution.

4.1 Front End Throttling of the Pipeline

A common problem that accompanies asynchronous designs is that the system becomes unbalanced. The more common case is that the fetch rate exceeds the commit rate. Usually, this problem can be averted by inserting queues between pipeline stages to handle short spurts of unevenness in the instruction flow. However, this is not ideal for low power or good performance. When the queues are empty, the system is still executing at the same clock rate and consuming the same power, but not doing anything productive. When the queues are full, performance drops significantly. Queues only work for short periods of fluctuations.

It would be ideal to incorporate some intelligence into the fetch and decode stages so that they will know when to slow down due to earlier bottlenecks in the system. For example, if the queues to the functional units are at capacity, then the fetch and decode stages should stop issuing instructions until the queues start to empty. On the other hand, if the system is empty, then there is inefficient usage of the available resources, which is just wasting power. Thus, the fetch and decode stages should readily adapt to this and begin issuing instructions at a faster rate if resources in the pipeline are underutilized and slower if there is congestion in the pipeline.

4.1.1 Fetch/Decode Adaptive Algorithm

The algorithm we developed keeps track of the number of instructions committed and fetched within a set window size. The window size determines the number of instructions executed between each sampling period. Every sampling period, the number of committed instructions and the number of fetched instructions are compared. If the number of instructions fetched exceeds the number of instructions committed by a

predetermined threshold, then the fetch and decode units are clocked down. On the other hand, if number of instructions fetched and the number of instructions committed is higher than a certain threshold, then that means that the instruction flow is once again pretty smooth and the fetch clock can be sped up again. The tricky thing is to determine the window size, the threshold values, and the amount to adjust the clock by. These issues will be addressed in the next section.

```
if (num_inst >= window_size) {

 // commit rate too high
 if (commit_rate – fetch_rate >
              threshold_high)
   // increase fetch rate
   clock_rate_fetch =  HIGH_MODE;

 // fetch rate too high
 if (commit_rate – fetch_rate <
       threshold_low)
    // slow it down
    clock_rate_fetch = LOW_MODE;

 // else do nothing.
}
```

Figure 2: Pseudocode of algorithm for adjusting the clock frequency of the fetch unit.

Ideally, we would just like to switch the clock between a high frequency and a low state. The amount of clock adjustment is very important because it is possible to continually overshoot and undershoot the desired clock rate. Thus, the system will be infinitely switching its clock frequency every sampling period, never reaching an optimal execution speed. However, if it is too small, then the system will not be able to adapt to the needs of the application quickly enough. Semeraro and Albonesi [3] suggest that the amount of change should be 3-12% to produce optimal results. However, our system cannot handle dynamic thresholds. We only support two clock frequencies for the fetch unit. Not only is this simpler to implement in hardware, but also more power efficient because complicated computation tends to drain more power.

The window size of the fetch stage determines the frequency of sampling that occurs. This affects the reactivity of the fetch stage to changes in the system. If the window size is too large, then there will be a significant amount of delay before the system adapts to changes. On the other hand, if the window size is too small, then the system will be excessively sensitive to changes and adapting superfluously. [3] uses a window size of 10,000 instructions, whereas [5] uses a window of only 1024 instructions. In Section 6, we will explain how we came to our optimal window size, 40,000 instructions.

The values of the high and low thresholds of the fetch stage determine the sensitivity of the system to performance changes. If the values of the thresholds are too narrow, once again, the system will be adjusting too frequently. If the values are too wide, then the system will not adapt to performance changes at all. [3] suggests that a threshold of 0.75 – 1.75% produces optimal performance and low power gains. We found that strategically chosen narrow values actually work best. We will go further in detail about the specifics of this in the Experimental Results section later on.

4.1.2 Hardware Requirements

To implement the above scheme, we basically need two instruction counters, one for the fetch unit and one for the commit unit, to keep track of the number of fetches or commits done between sampling periods. In addition, a counter is needed for determining when the end of a window size has ended. In addition, a shifter and an adder can be used to compare the values of the number of instructions fetched and the number of instructions committed within a sampling period. In [5], the optimal power-performance gain was achieved when the front end was throttled when three times the number of instructions was being fetched as were being committed. Our optimal threshold was much stricter. This will be discussed in more detail later on. Overall, the hardware needed to implement this algorithm is not that complex, making this implementation very feasible.

4.2 Data Dependencies among Functional Units

The queue algorithm for scaling the clock frequencies of the functional units mentioned in [1] and [2] determines operating frequency of the functional units based solely on queue length. This stems from the assumption that there is a direct correlation between the number of entries in the issue queue for each clock domain over an interval of instructions and the desired frequency of that particular domain.

The queue length is indicative of the rate at which instructions are flowing through the instruction core. If the queue length increases, it implies that instructions aren't flowing into the functional units fast enough.

However, this approach might be unacceptable in conditions where there are dependencies across various functional units. For example, the memory unit might be stalling on an operand that to be generated in the integer unit. In this example, all memory operations subsequent to the memory instruction that is stalled will queue up in the memory unit issue queue. If the clock frequency uses the queue length of the memory unit as an estimator for frequency scaling, it would speed up the clock of memory unit. However this is unnecessary, considering that the dependency resides in the integer unit. Thus increasing the frequency would result in no significant effective increase in performance of the memory unit.

Thus we believe that when there are inter-dependencies between the functional units it might be more sensible to base the frequency scaling upon the number of ready instructions in the queue, those which have no dependencies at all associated with them. Thus, it would be more appropriate to base the frequency on the number of uncommitted instructions starting from the head of the queues that have no dependencies associated with them.

In addition, we need to keep count of the number of dependencies there are for each functional unit. This would help for situation such as if the memory issue queue is full of instructions that are waiting for an instruction executing in the integer queue. Ideally, the integer queue operating frequency would be sped up, reducing the wait time for the instructions in the memory queue.

We plan to integrate this with our earlier mechanism that aims at minimize the difference between the fetch and commit rates. We believe that corrective actions based on fetch-commit rate should take place over a larger window. Thus there should be no need to adjust the clock rates of the front stages of the pipeline unless the adjustment cannot be sorted locally within a particular clock domain, especially considering the fact that commit-rate based throttling is more coarse-grained and affects the propagation speed through all the pipe stages.

4.2.1 Dependency Tracking Algorithm

We describe below the modified implementation of the dependency-tracking algorithm. The pseudocode for the dependency checking algorithm is shown in Figure 1 below. The implementation of the Dynamic Voltage scheme has not changed significantly from the original algorithm used by [1] and [2]. The difference is that we use the number of ready instructions rather than the occupancy in the queue.

```
if (state == HIGH_STATE) {

  if (ready inst in queue < threshold)
    increment count;

} else {

  if ((ready inst in queue < threshold)
      || (dep_cnt > dep_threshold))
     increment count;

}

if (count > CLOCK_INTERVAL) {

  state = !state;  //switch states
  count = 0;       //reset counter

}
```

Figure 3: Modified dependency-checking algorithm for dynamically adjusting the operating speed and voltage of a functional unit.

We associate two counters with each functional unit. One of these tracks the number of independent instructions associated with that unit. The other keeps track of the number of dependencies that are waiting for the completion of an instruction with that particular functional unit.

When an instruction is first dispatched to the queue of a functional unit, its operands are checked. If the operands are ready, then the independent counter for that functional unit is incremented. This increment signifies that there is another ready instruction in the queue to be issued. However, if the instruction's operands are not ready yet, then a linear search through the execution units of each functional unit is executed to find the instruction that this current waiting instruction is dependent on. When such an instruction is found, the dependent counter is incremented. This conveys to the other functional unit that there is a dependant instruction waiting for an executing instruction.

When an instruction completes execution, it must notify all dependent instructions that its result is ready. Thus, it must

once again linearly search through all the queues of the functional units and wake all instructions that are dependent upon its result. At this point, for each instruction woken up, the dependent counter for its associated functional unit is decremented and the independent counter for the functional unit of the waiting instruction is incremented. Thus, when all dependencies are resolved, the dependent counters for all functional units should be zero.

Finally, when an instruction commits, the independent counter of the associated functional unit is decremented.

/*

Unfortunately, dealing with synchronizing the counters for the functional units spanning different clock domain proved to be a more daunting task than it seemed. Synchronizing the counters correctly was difficult when writebacks were done at different times (each functional unit writes back to its own queue first, then buffers the writebacks to the queues of the other functional units). In the end, we simulated the counters by searching through the queues whenever we needed an independent count and a dependency count for each functional unit. Although this made the simulator run much slower, the correctness of the counters is still ensured.

*/

4.2.2 Hardware Requirements

The hardware needed to implement this algorithm is not substantial. Two counters are needed for each functional unit to keep track of the number of independent instructions and the number of dependent instructions. In addition, there needs to be a counter for keeping track of the instruction count in the window size for each functional unit. In addition, two comparators are needed for each functional unit, one each for signaling when the sampling period of the algorithm has been reached and another one each for determining if the number of ready instructions is larger than the thresholds. In addition, one more comparator for each functional unit is needed for determining if the number of dependent instructions is larger than the dependent instruction threshold.

5 Simulator Details

We conducted our experiments on a modified version of the simGALS simulator presented in [1] and pictured in Figure 1. This simulator has been modified to support the dynamic clock/voltage scaling of the fetch stage in the pipeline in addition to the data dependency checks described earlier in Section 4.2. Below lists the specific configuration of the simulator used.

decode width:	**4**
number int registers:	**72**
number fp registers:	**72**
int queue size:	**20**
int issue width:	**4**
int commit width:	**4**
fp queue size:	**16**
fp issue width:	**4**
fp commit width:	**4**
mem queue size:	**16**
mem issue width:	**4**
mem commit width:	**4**
level 1 data cache	
# sets:	**128**
block size:	**32**
associativity:	**4**
replacement policy:	**LRU**
level 1 instruction cache	
# sets:	**1024**
block size:	**64**
associativity:	**1**
replacement policy:	**LRU**
level 2 unified cache	
# sets:	**512**
block size:	**32**
associativity:	**1**
replacement policy:	**LRU**

Figure 4: Configuration of the simGALS simulator.

6 Experimental Results

6.1 Baseline Model

[1] and [2] describe a GALS architecture with distinct clock domains for each functional unit group. We use their dynamic voltage scaling algorithm as the baseline for comparison of our results. This configuration does not include either front end scaling or dependency checking for speed and voltage

adjustments of the functional units, which makes it an appropriate baseline model for determining if the front-end throttling and the dependency checks reduce the power consumption of the system. It supports up to five different clock domains, but [1] and [2] only use four (the fetch and decode stages use the same clock).

Experiments were done only on two benchmarks from Spec95: compress95, which is a very regular integer benchmark, and fpppp, which is an irregular floating point benchmark. The IPC, power and energy values for the baseline model are tabulated in Table 1 below.

Benchmark	IPC	Power	Energy
Compress95	1.9526	20.1892	10.4847
fpppp	0.7596	23.9086	31.4743

Table 1: Results of the baseline simulator.

6.2 Fetch Stage

The two variables concerned with the fetch stage throttling algorithm that needed to be fine-tuned were the fetch stage upper and lower thresholds, and the fetch stage window size.

The fetch stage upper and lower thresholds determine when the fetch stage should dynamically change its operating clock frequency and operating voltage. If the thresholds are too lax, then the clock frequency will never be scaled thus limiting the extent of voltage scaling. However, if the thresholds are too tight, then the algorithm will be switching the frequency of the fetch clock after every sampling period, resulting in greater power consumption than the baseline due to switching energy.

The upper and lower thresholds were determined by observing the behavior of the fetch stage at different thresholds and making an educated guess of reasonable values to narrow down the number of simulations conducted. The chosen values were:

- Set 1: High threshold: 10
  Low threshold: 100
- Set 2: High threshold: 10
  Low threshold: 250
- Set 3: High threshold: 10
  Low threshold: 500

The threshold values correspond to the difference between the number of instructions fetched and the number of instructions committed for a sampling period. If the difference is less than the High threshold, then the fetch stage will switch into HIGH mode. If the difference between the number of instructions committed and the number of instructions fetched is greater than the Low threshold value, then the fetch stage will clock down into a LOW mode. Thus, looking at the first set of values, if the difference between the fetch and the commit numbers is less than 10, then the fetch stage will switch to HIGH power (high frequency) mode if it is not already running in the HIGH power state. If the difference is greater than 100, then the fetch stage will switch to a LOW power (low frequency) mode.

The graphs below plot the results of varying the threshold while holding the window size constant. The experiments for Figures 5 through 7 were done with the window size fixed at 40,000 instructions.

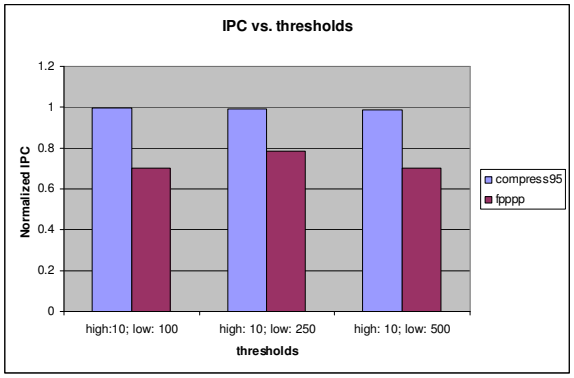


Figure 5: IPC for various thresholds

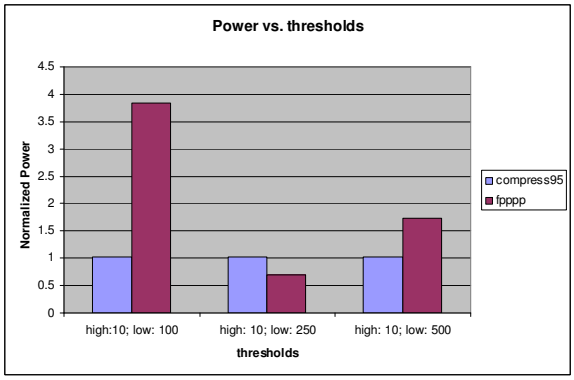


Figure 6: Power for various thresholds

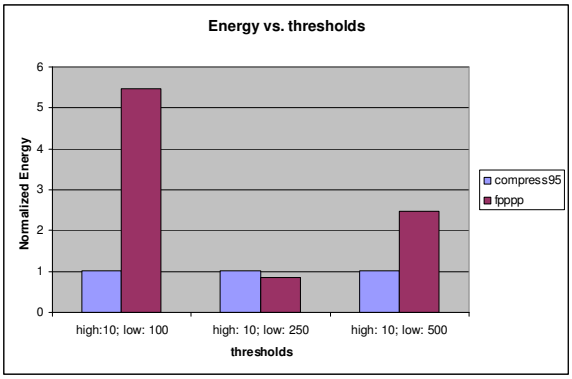


Figure 7: Energy for various thresholds

Figure 5 shows that the IPC from running the modified simulator on compress95, an integer benchmark, and fpppp, a floating point benchmark. The results clearly indicate that running the simulator on compress95 does not really provide any performance benefits. Although the IPC increase for the second data set is higher than the other two, the difference is too negligible to account for by any rational explanation.

Figures 6 and 7 show that the power and the energy consumption of the system do not change with window size. This is due to the fact that compress95 is a very regular benchmark. In fact, there is no significant switching of the fetch stage of the processor, ensuring that the fetch stage clock is in the performance mode over the entire run of the benchmark. Thus, it is reasonable that the results show neither performance benefits nor power savings.

For the floating point benchmark, fpppp, however, there is significant variation in the IPC with the thresholds. With a low threshold of 100, the system really does not have much scope for any variation in the fetch and commit rates. Once the fetch rate and the commit rate are slightly skewed, then the system will switch into the LOW power state. On the other hand, having a low threshold of 500 is not ideal either because the system will continue fluctuating between HIGH and LOW power states.

For the floating point benchmark, fpppp, however, there is more variation in the IPC due to the thresholds. With a low threshold of 100, the system does not have much scope for any variation in the fetch and commit rates. Once the fetch rate and the commit rate are slightly unbalanced, the system will immediately switch into the LOW power state. This will cause the system to quickly decrease its fetch rate, equalizing the fetch and commit rates and sending the system back into a HIGH power state. Thus, the system is constantly fluctuating between the HIGH and LOW power states. On the other hand, when the low threshold is increased to 500, the system will wait for the commit rate and fetch rates to become very unbalanced before switching to a LOW power state. So, the system will take a period of time for the commit rate to catch up to the fetch rate, putting the system back into a balanced state. Of the three chosen sets of high and low thresholds, the middle set, (high threshold: 10, low threshold: 250) seems to be the best compromise between the two.

In addition to determining the high and low switching thresholds for the fetch logic, the optimal window size of the fetch logic also had to be determined. To do this, we simulated running the two spec95 benchmarks on the modified simulator with window sizes of 20,000 instructions through 50,000 instructions, incrementing by 10,000 instructions and keeping all other factors static. The threshold used for the results is high = 10 and low = 250 based on the earlier experiments conducted to calculate optimal thresholds. The results can be seen in Figures 8 through 10. Once again, compress95 doesn't show any improvement because there is practically no fetch stage throttling. For fpppp, it is seen that a window size of 40000 yields best results with respect to power and energy savings. Here, the power consumption is 67% of that of the baseline, with only a 12% drop in IPC. Energy consumption is only 86% of that of the baseline simulator.

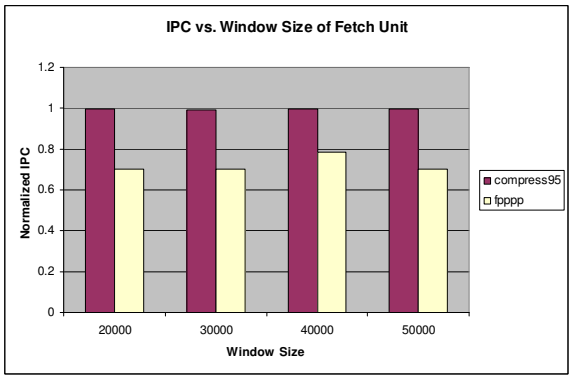


Figure 8: Normalized IPC with respect to the baseline.

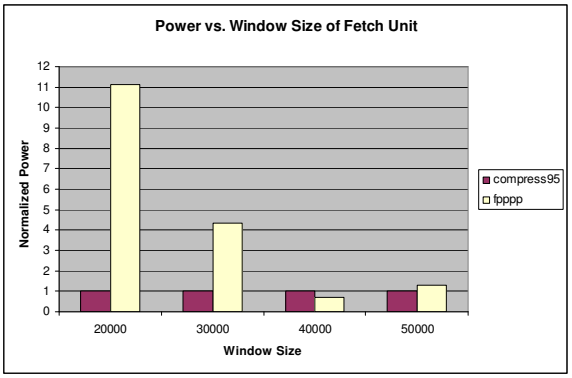


Figure 9: Normalized Power with respect to the baseline.

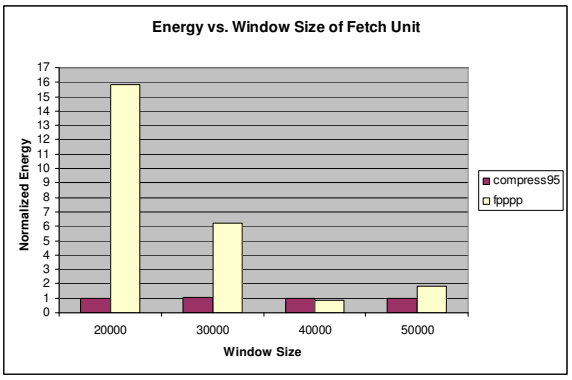


Figure 10: Normalized Energy with respect to the baseline.

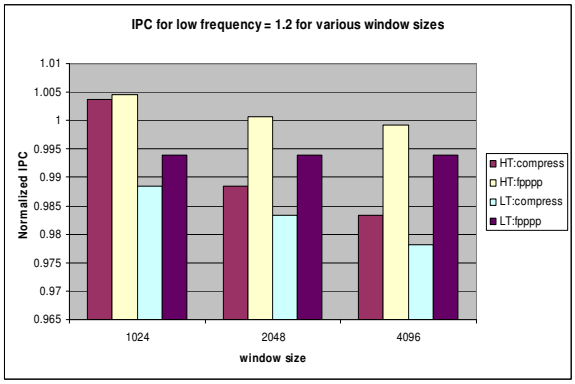
6.3 Data Dependency

The results for the influence of the proposed dynamic voltage scaling algorithm on the various parameters were studied through simulations conducted across two benchmarks: compress95 and fpppp.

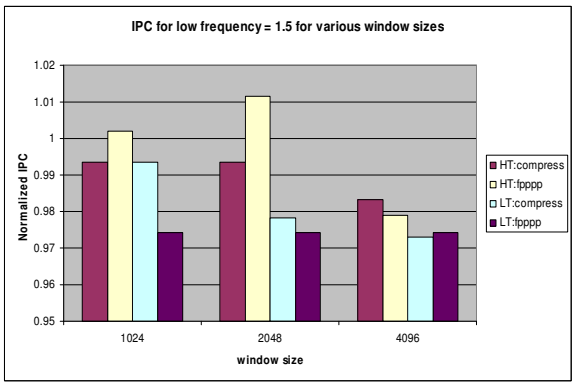
Since the solution space to the problem of finding a set of parameters that scale well across extensively varying program behavior is extremely large, rational decisions were called upon to quickly prune the solution space to converge on a group of parameters that would yield high power savings without significant performance degradation.

The following results were simulated with two sets of thresholds. One set (upper threshold, lower threshold) pair with low thresholds corresponds to a performance mode because only when soon the buffer occupancy drops below a very low value does the transition to the LOW STATE (power state) take place and as soon as buffer occupancy rises above a slightly higher value for a continuous period that is equal to the window size, a transition to the HIGH STATE (performance state) takes place. The other set of thresholds with higher values and a bigger difference between low and high thresholds corresponds to a power mode because the functional unit flips to a low power state soon and only if the independent instructions in the issue queue rises above the large high threshold value for a period that is equal to the window size is there a transition to a HIGH power state. The values chosen for the experiments for the performance mode were (2,4) and for the power mode were (4,12).

The variation of the IPC (Instructions per Cycle) of the test benchmarks with window size is depicted below. As can be seen , as the window size increases there is a slight decrease in the IPC because of the increase in the average instructions that the program stays in the power mode. Overall there is no conspicuous degradation in the IPC.



(a)

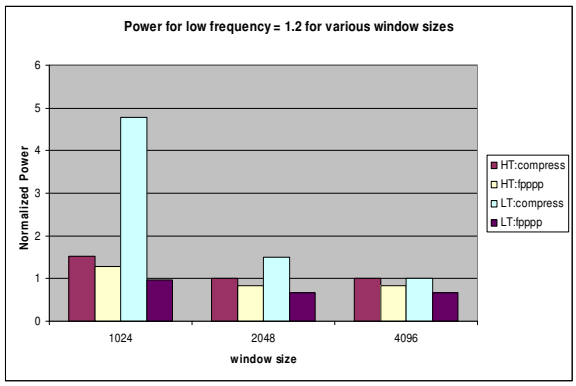


(b)

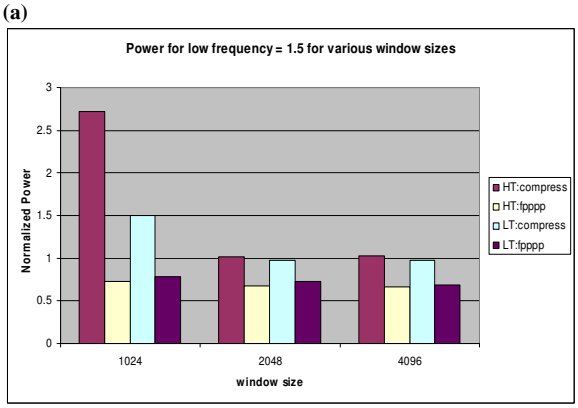
Figure 11: Variation of IPC with window size (a) Frequency: 1.2 (b) Frequency: 1.5.

The variation of the average power per cycle normalized to the baseline implementation of [1] is shown below. The graphs show that there is practically no power savings for window sizes 2048 and 4096 for the compress benchmark. This is because the compress95 is a

fairly regular predictable benchmark without significantly varying performance behavior. In fact the only window size for compress for which there is switching is for window size 1024.However, due to the excessive switching there is high switching power loss and hence normalized power is greater than the baseline. For the fpppp benchmark , there are significant power savings(78% for performance mode and up to 73% for power mode). It was noted that beyond a window size of 4096, there were no significant improvement in the power savings with window sizes to justify the loss in the IPC.



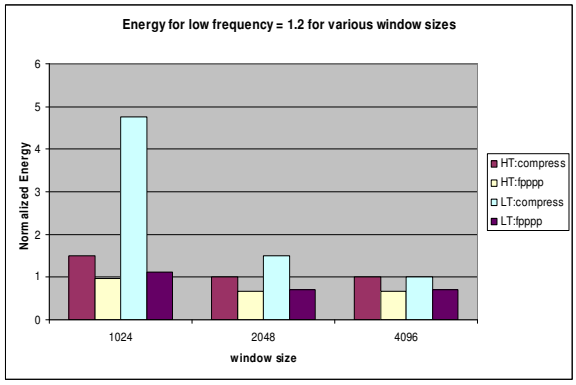
(a)



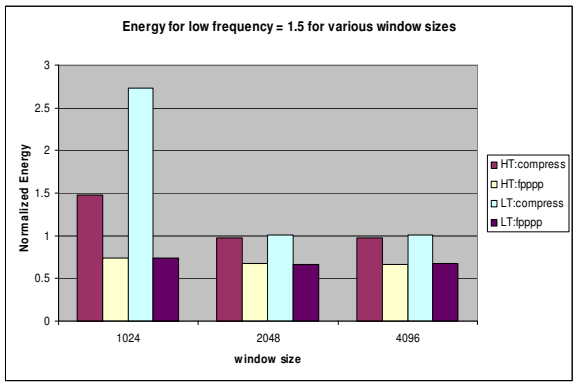
(b)

Figure 12: Variation of power_cc3 with window size (a) Frequency: 1.2 (b) Frequency: 1.5

The trends in the energy savings with window size is consistent with the power savings graph shown in Fig 5. It is seen that for the fpppp there is significant energy savings while for the compress there is no savings because the processor perpetually stays in the high-performance state.



(a)



(b)

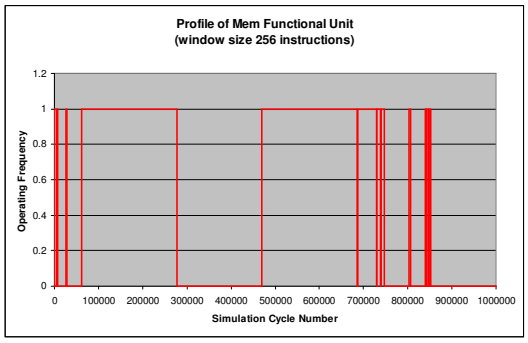
Figure 13: Variation of energy_cc3 with window size (a) Frequency: 1.2 (b) Frequency: 1.5

From the graphs, it is clearly visible that the dominant parameter that affects the size of the dynamic voltage scaling algorithm is the window size. As the window size increases the effect of the thresholds become less prominent. Besides, the period of the low performance mode clock can be relatively large (up to 1.5 times that of the high-performance mode clock) without any significant drop in the IPC.

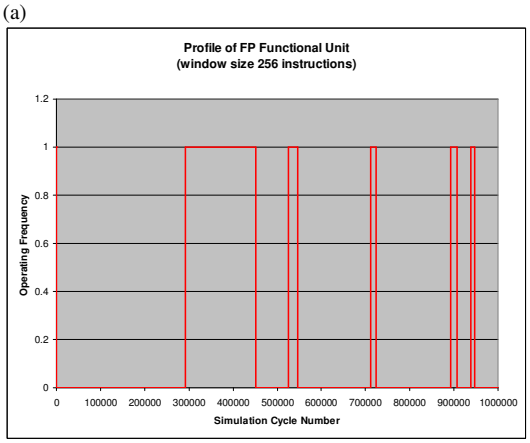
The typical profile of each of the separately clocked functional units in a GALS architecture for the fpppp benchmark is shown in Figure 14. This program trace uses a small window size of 256 over 1 million instructions to depict the activity of the functional unit to a high degree of accuracy. We depict the state-graphs of each functional unit: memory unit, integer unit and floating unit. The state 1 corresponds to the high performance mode while state 0 corresponds to the power mode (low performance mode).

It can be seen that the memory functional unit stays in the high-performance mode more than 50% of the time. This seems intuitively correct because memory operations will be associated both integer and floating

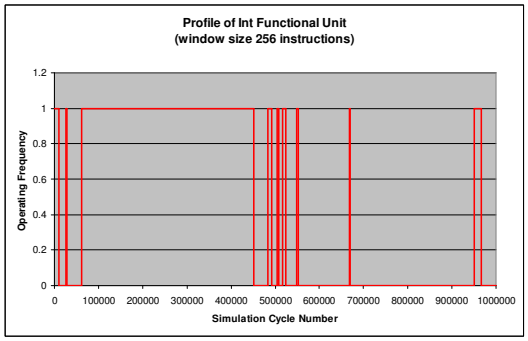
operations when of the respective functional units are active. The floating point unit on the other hand depicts a very low average utilization. It lies in the inactive state most of the time. The utilization of the integer unit is around 50% in the first million cycles due to its high usage early on.



(a)



(b)



(c)

Figure 14: Profile of fpppp benchmark over 1 million instructions of functional unit state vs time (a) Memory unit (b) Floating point unit (c) Integer Unit

Figure 14 depicts the profile of the fpppp benchmark over its entire run-cycle. As can be seen, the integer unit switches states (to a LOW) at cycle 547963 which is consistent with the expected profile behaviour from Figure 14(c). This is because, as shown in Figure 14(c), the integer unit remains fairly active till around cycle 550,000, after which activity drops. The larger window size in Figure 14 prevents the excessive switching that is evident in Figure 15.

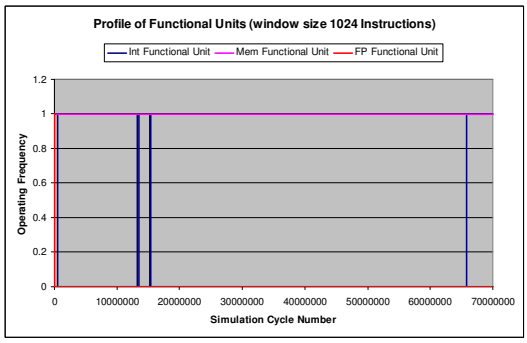


Figure 15: Profile of fpppp benchmark for 1024 window size over entire benchmark for thresholds (2, 4).

7 Conclusions

We modify the original scheme proposed in [1] to incorporate two new features. The first of these features is throttling of the fetch stage of the CPU pipeline while the other modifies the original algorithm proposed in [1] to account for the independent instructions that were waiting in the issue queue and total dependencies associated with that FU as a trigger for the dynamic voltage scaling.

We have implemented our proposed changes on the simGALS simulation platform [1]. Our results show significant power savings with insignificant associated penalty in performance.

8 Future work

Since program behavior is so benchmark and program-phase dependant**,** it would be interesting to evolve analytic probability distribution models that characterize the instruction flow at each phase. This would result in sound mathematical principles in choosing the thresholds and window size values. We do not believe that the scheme of dynamic thresholds is feasible because such an implementation would be necessarily have to be too complex to implement in hardware besides introducing new variables to the DVS algorithm. We also believe that simple mechanisms such as [1] are more feasible to implement in actual

implementations as compared to complicated schemes such as [2].

In addition, the parameters that were chosen as being optimal are very application specific. Only two benchmarks were tested: compress95, an integer benchmark, and fpppp, a floating point benchmark. It would be interesting to test the modified simulator on a wider set of benchmarks to see if there results are similar to the ones we present in this paper

9 Schedule

Below are the steps that we took for completing the research that needed to be done for this project. We followed this schedule fairly tightly and were able to complete the project in the time originally proposed.

Milestone 1 (10/3):

- Identify the dependencies among functional units that will, in turn, be exploited by the Data Dependency Checker.
- Familiarize ourselves with simGALS environment.
- Create module for determining issue and commit rates.

Milestone 2 (10/17):

- Implement throttling of the front end of the pipeline

Milestone 3 (10/31):

- Implement dependency checking per functional unit for speed and voltage adjustments.

Milestone 4 (11/4):

- Integration of the Milestones 2 and 3.

Final Milestone (12/2):

- Simulation/verification of proposed schemes.
- Drawing inferences.

10 Division of Labor

Since there were two seemingly orthogonal methods proposed, it seemed logical to split the work down that line as well. Shelley implemented, tested, and simulated the front end throttling mechanism and Anand implemented, tested, and simulated the Data Dependency Checker module. In the end, when the two modules were integrated, we needed to both simulate the final simulator. Most of the time, we kept to our assigned jobs.

11 References

[1] A. Iyer and D. Marculescu. Power-Performance Evaluation of Globally Asynchronous, Locally Synchronous Processors. *Intl. Symposium on Computer Architecture (ISCA)*. May 2002.

[2] A. Iyer and D. Marculescu. Power Efficiency of Multiple Clock, Multiple Voltage Cores. *IEEE/ACM Intl. Conference on Computer-Aided Design (ICCAD).* Nov. 2002.

[3] G. Semeraro, D.H. Albonesi, S.G. Dropsho, G. Magklis, S. Dwarkadas, and M.L. Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. *35th International Symposium on Microarchitecture*. November 2002.

[4] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. *8th International Symposium on High-Performance Computer Architecture*. pp. 29-40, February 2002.

[5] A. Moshovos, D. N. Pnevmatikatos and A. Baniasadi. Instruction Flow-based Front-end Throttling for Power-Aware Higher-Performance Processors. *Proc. International Conference on Supercomputing (ICS)*, June 2001.

[6] D. Burger, T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342. University of Wisconsin, Madison, Wisconsin, June 1997.