Frontend Frequency-Voltage Adaptation for Optimal Energy-Delay^2

Grigorios Magklis, José González, Antonio González
Intel Barcelona Research Center, Intel Labs—UPC
{grigoriosx.magklis, pepe.gonzalez, antoniox.gonzalez}@intel.com

Abstract

In this paper we present a clustered, multiple-clock domain (CMCD) microarchitecture that combines the benefits of both clustering and Globally Asynchronous Locally Synchronous (GALS) designs. We also present a mechanism for dynamically adapting the frequency and voltage of the frontend of the CMCD with the goal to optimize the energy-delay^2 product (ED2P). Our mechanism has minimal hardware cost, is entirely self-adjustable, does not depend on any thresholds, and achieves results close to optimal. We evaluate it on 16 SPEC 2000 applications and report 17.5% ED2P reduction on average (80% of the upper bound).

1. Introduction

The scalability of current superscalar processors is impeded by the impact of wire delays, by the increasing complexity of processor components and by excessive power dissipation. Microprocessor designers find it increasingly harder to sustain the growth rate of clock frequency, or to construct wider pipelines for high-performance processors.

It has been shown that clustering is an effective approach to overcome some of these issues [1][10]. The impact of wire delays is reduced by keeping intra-cluster communication fast and minimizing inter-cluster communication. Clustering also reduces the effective complexity of large structures, such as caches, allowing for faster clock frequency and reduced power dissipation. Moreover, clustering allows for finer-grained adaptation of resources.

On the other hand, Globally Asynchronous Locally Synchronous designs are a very effective way to address the problem of clock distribution in large microprocessors, including clustered ones. Earlier proposals have shown that a GALS design style can be successfully applied to monolithic, wide-issue, out-of-order processors [7][15]. Moreover, it was also shown that the fine-grained dynamic voltage scaling capabilities of the MCD allow for significant power reduction while at the same time keep performance close to maximum [9][13][15].

In this study we propose (a) a Clustered Multiple Clock Domain microprocessor, and (b) a mechanism for dynamically adapting the frequency and voltage of the frontend of the CMCD to optimize the energy-delay^2 product of applications. In the proposed microarchitecture, the division into clock domains follows naturally the separation into clusters. Our design combines the benefits of the clustered and GALS paradigms. By clustering the various resources we can achieve better power efficiency and higher clock frequency. By dividing the processor into separate clock domains, both the complexity and the power dissipation of the clock network are reduced, running at higher frequency is possible, and a more modular design is allowed.

In our architecture the frontend still follows a monolithic design. This results in disproportionate power dissipation on the frontend compared to the other parts of the processor. In our case it dissipates about 39% of the total power, while each backend only about 11%, and the L2 cache 16%.

Apart from the definition of the microarchitecture, the main contribution of this paper is a novel mechanism for reducing the power dissipation of the frontend of the processor through dynamic voltage and frequency scaling. We have designed a control system based that can accurately predict the runtime of the application based on the frequency of the frontend, the branch misprediction rate and the occupancy of the instruction fetch queue. Our performance predictor is derived from the principles of queuing theory and has several benefits: it is easy to calculate, it has no thresholds, and is self-adjustable. Most importantly it can predict the performance of the application at any frequency, given statistics at any other frequency.

The rest of the paper is organized as follows: Section 2 introduces the CMCD architecture in more detail. Section 3 explains our frontend control system. In Section 4 we present our experimental analysis and compare our system with an optimal one. In Section 5 we discuss related work, and finally in Section 6 we conclude the paper and point out areas for future work.

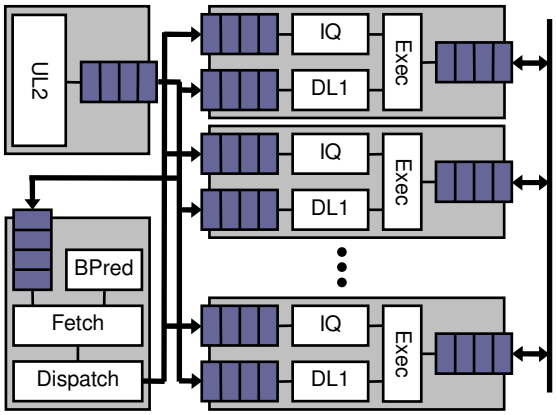

Figure 1. CMCD high-level block diagram

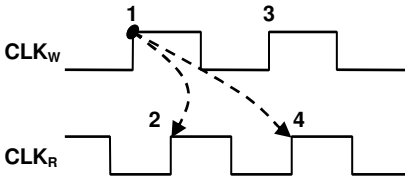

Figure 2. Synchronization timing

2. Architecture

A diagram of our proposed architecture can bee seen in Figure 1. The frontend is responsible for fetching and dispatching instructions. The fetch mechanism is similar to that of the Intel® Pentium® 4, utilizing a branch predictor, a trace cache, and an IA32 decoder. The instruction dispatch consists of the register renaming and the steering of micro-ops to the various backends. The frontend also includes the re-order buffer and the commit logic of the processor.

The backends follow the structure of typical clustered microarchitectures. Each one includes the issue window, register file and execution units for the integer and floating-point micro-ops, as well as the memory order buffer, disambiguation logic and a first-level data cache. Register values are communicated among the backends using special copy micro-ops via a crossbar network [1][4].

Communication between domains happens through special synchronizing FIFO queues. Figure 1 shows all the communication paths between domains and the associated queues (in dark color). In our design, these queues already existed in the architecture. The queues' read and write interfaces are in different domains and there is special circuitry to make sure that the data is always stable when it is read, and that both domains have a coherent view of the full/empty status.

In this study we assume the circuits and timing analysis of Semeraro *et al.* [14]. Figure 2 shows the timing of the synchronization circuit. Assume a queue between two domains with clocks CLK_W and CLK_R, and data written into the queue at the rising edge 1. If the time difference between edge 1 and edge 2 is smaller than some fixed threshold then the data will be visible (and stable) at the read interface of the queue at time 2, otherwise at time 4. In our simulation this threshold is set to 30% of CLK_R.

We also propose that each domain has its own independent voltage regulator and frequency controller We assume a discrete range of frequency-voltage levels. Changes in voltage and frequency can be done without stopping the domain provided that they are between adjacent levels. The available frequency-voltage levels that we assume for our architecture are shown in Table 1. The time to go from level 0 to level 20 is 1μsec.

Table 1. Frequency-voltage levels

#	MHz	Volts	#	MHz	Volts	#	MHz	Volts
0	10000	1.100	7	8348	0.832	14	6696	0.632
1	9764	1.057	8	8112	0.800	15	6460	0.608
2	9528	1.016	9	7876	0.769	16	6224	0.584
3	9292	0.976	10	7640	0.739	17	5988	0.562
4	9056	0.938	11	7404	0.711	18	5752	0.541
5	8820	0.901	12	7168	0.683	19	5516	0.521
6	8584	0.866	13	6932	0.657	20	5280	0.501

3. Frontend Adaptation

The goal of our frontend adaptation mechanism is to reduce the ED2P for any given application. To achieve this we split the execution into fixed intervals (measured in micro-ops) and we try to minimize the ED2P of each interval. In order to minimize ED2P, we first calculate the execution time and energy consumption of an interval. The assumption we make is that the behavior of the program for some interval can be predicted by observing the behavior of the previous interval.

3.1. Dynamic Energy Estimation

Assuming that the amount of work will be the same, the energy consumption of interval n+1 in which the frontend runs at frequency-voltage (f_n+1, V_n+1) can be estimated from the energy of interval n with the frontend at frequency-voltage (f_n, V_n) using the following equation:

$$E_{n+1} = E_n + E_{FE,n} \times \left(\frac{V_{n+1}^2}{V_n^2} - 1 \right)$$

In this equation E_{FE} is the energy of the frontend domain, which is the only part subject to frequency-voltage scaling. To measure the energy of the different processor components at runtime, we use the technique

proposed in [6]. Performance counters are used to measure the activity of the various blocks during an interval. Each block has also associated with it a register, the *Energy per Access Register* (EAR) that holds the average energy per access. The value of the EAR is fixed, and set by the designer. At the end of the interval each performance counter is multiplied with its respective EAR to calculate the energy.

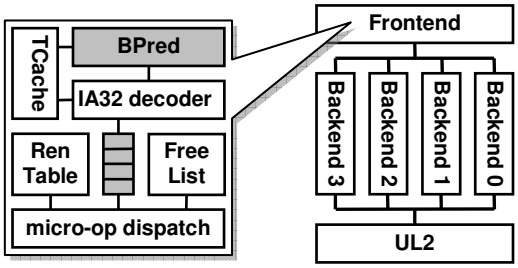

Figure 3. Frontend details

3.2. Dynamic Execution Time Estimation

Our frontend is somewhat similar to that of the Intel® Pentium® 4 processor (see Figure 3). In essence, what we want is to find a relationship between the frequency of the frontend and the speed at which we fetch instructions and the speed at which we feed micro-ops to the rest of the pipeline. It is very convenient for us that we can measure both the fetch and the dispatch bandwidth at the two ends of the fetch queue (dark shaded queue in Figure 3).

For our execution time estimator we used the principles of queuing theory to find the relationship between performance (dispatch bandwidth), frontend frequency, fetch queue utilization and fetch bandwidth:

$$\frac{T_{n+1}-T_n}{T_n}=\left(\frac{f_n}{f_{n+1}}-1\right)\times\frac{1-p_n}{1+b}$$

In this equation p_n is the average number of entries in the frontend queue for the n-th interval and b is the branch misprediction rate since the beginning of the execution. The first term of the product involves the expected speed-up or slowdown due to the change in frequency. This just says that any change in frequency should yield a similar change in execution time.

The more interesting part is the second term. According to Little's Law the average delay T of a client in a queuing system is equal to $T=N/\lambda$, where N is the average number of entries in the queue and λ is the rate of arrivals in the queue. In our case, λ signifies how fast we can feed the fetch queue and depends on the branch misprediction rate, N is the number of unused entries in the queue, and T is an indication of the dispatch speed.

3.3. Dynamic Frequency Adaptation

Our control system at the end of each interval uses the energy and execution time estimators to calculate the ED2P for the upcoming interval for each frequency-voltage level. We only consider the cases where the frequency and voltage are at the same level; we could for example run at high voltage and low frequency but this is not interesting for energy reduction.

After the ED2P for each level is estimated, we pick the level that has the minimum value. More formally, we solve the system:

$$\min_{(f_{n+1},V_{n+1})\in L}\left(T_{n+1}\times T_{n+1}\times E_{n+1}\right)$$

Where T_n+1 and E_n+1 are calculated as before, and L is the set of all 21 possible frequency-voltage pairs.

Table 2. Architectural parameter summary

Frontend	
Trace Cache	32K mops, 4-way, 4 cycles to dispatch
Dispatch	8 mops/cycle, 8 cycles latency
2nd Level Unified Cache	
UL2	2 MB, 8-way, 12 / 500+ cycles
Each of 4 Backends	
Integer	20-entry FIFO, 40-entry IQ, 1mop/cycle
Floating-point	20-entry FIFO, 40-entry IQ, 1mop/cycle
Memory	20-entry FIFO, 96-entry MOB, 1mop/cycle
Copy	20-entry FIFO, 20-entry IQ, 1mop/cycle

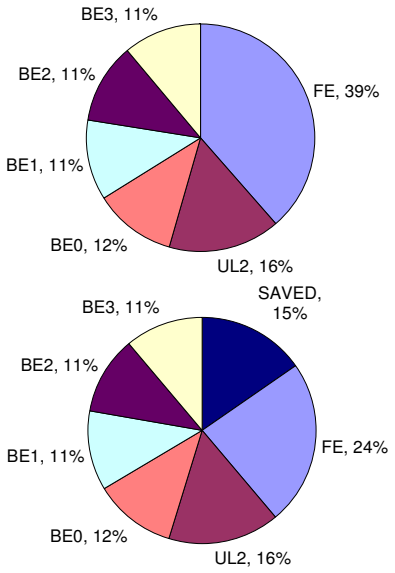

Figure 4. Power distribution before (top) and after (bottom) adaptation

4. Results

For the evaluation of our system we use a trace-driven simulator that executes traces of IA32 binaries, including OS code. The simulator also includes a power estimation module, based on an enhanced version of Cacti [16], utilizing activity counters in a fashion similar to Wattch [3]. The assumed technology is 65nm. Table 2 shows the main architectural parameters of the microprocessor. For our experiments we use sixteen applications of the SPEC CPU2000 benchmark suite—nine integer and seven floating-point. For all cases we simulate one hundred million IA32 instructions from the middle of the application.

Figure 5 shows the slowdown and power reduction of the CMCD microprocessor running at maximum frequency, compared to an identical, fully synchronous one. All of the performance loss shown in this figure is because of inter-domain synchronization penalties. The worst case slowdown is 6.7% for *eon* with an average of 2.8% over all benchmarks. This is because every time register values must be communicated between clusters the data transfer between the source and destination domains needs to be synchronized which increases communication latency by 7%-9% for all programs. As shown elsewhere [11], the performance in clustered microprocessors is quite sensitive to inter-cluster communication latency.

Our CMCD processor dissipates less power than a fully synchronous design—between 10%15% for all benchmarks, with an average of 12.5%. This reduction is due to the simplification of the clock distribution network: for the fully synchronous microprocessor a 4-level H-tree over a regular grid is assumed, following the method presented by Restle *et al.* [12]. For the CMCD, each domain has its own separate clock network. Using the same method for distributing the clock, and using the area calculated by Cacti, we have four 2-level H-trees for the backends, a 3-level H-tree for the frontend, and 3-level H-tree for the L2 cache. The total average dynamic power of the six smaller clock trees is about 30% less than the 4-level one.

Figure 6 shows the same results when dynamically adapting the frontend. The first thing to note here is that application performance is virtually unchanged, even though we slow down the frontend (average 3.7% slowdown vs. 2.8% for CMCD at maximum frequency). This is because our method tries to minimize ED2P, which gives higher priority to performance over power. An interesting phenomenon is the speed-up we observe for *swim*. Intuitively we would expect performance to get worse when slowing down parts of the processor. What happens in this case is that the scheduling of loads changes significantly (not the number of loads though), which results in a notable reduction in the DL1 miss rate. By slowing down the frontend, micro-ops in the backends are executed more in-order than before. In *swim* the highly out-of-order scheduling of loads results in conflicts in the DL1 (i.e. a future load displaces the data of a previous one). This effect is significantly reduced when slowing down the frontend: the average latency of load instructions goes down from 37 to 21 cycles, which of course results in higher performance.

It is also evident that the power dissipation is reduced significantly. This is, as expected, entirely due to the dynamic adaptation of the frontend. Figure 4 shows the average power distribution for the CMCD at full frequency and for the dynamically adaptive frontend. We can see that we reduce power dissipation by about 15% on average over the whole processor.

Figure 7 shows the ED2P improvement of the baseline CMCD ("CMCD"), of the CMCD with dynamic frontend adaptation ("FEDVS"), and of the CMCD with optimal frontend adaptation ("OPT"). The optimal scenario represents an upper bound calculated off-line, by collecting energy and time statistics at intervals identical to FEDVS (100K micro-ops), for all available frequency levels. Then we chose for each interval the frequency with minimum ED2P. This method—apart from using exact runtime information instead of having to predict it—also assumes that the frequency-voltage change is instantaneous, which of course is not true in our FEDVS scheme.

As expected, the ED2P of the baseline CMCD is improved compared to the fully synchronous design, but not significantly (less than 5% on average). Our frontend adaptation on the other hand achieves 17.5% improvement of the energy-delay2 product, which is very close to the upper bound (22% on average). The difference between FEDVS and OPT can be attributed to three factors: (a) the fact that frequency-voltage changes in OPT are instantaneous, (b) rapid changes in the behavior of the application that produce errors in our performance predictor, and (c) side-effects that are not modeled by our predictor (such as the cache behavior discussed above for swim).

5. Related Work

The benefits of clustering have been known for a long time [5][8], and there have been many studies on how to best distribute instructions among clusters [2][4], and how to efficiently communicate values between backends [11]. Our CMCD is based on a state-of-the-art clustered design that borrows main concepts from previous studies in this area.

GALS microprocessors have been proposed before [7][15]. In previous studies the division of the pipeline into domains was vertical: one domain for

integer, one for floating-point, and one for memory operations. We propose a horizontal division (fetch, execute, L2) combined with a vertical one (one domain for each cluster) that more closely follows the physical layout of the processor.

Dynamic voltage and frequency scaling for MCD processors has also been studied before [7][9][13][15], but not for clustered microarchitectures. Moreover, there is not—as far as we know—any previous study that dynamically adapts the frontend frequency of a MCD microarchitecture. Finally, it is worth noting that our control system, unlike previous ones [13], has no thresholds, is self-tuning, and can directly jump to the desired frequency-voltage level instead of approaching it through many small trial-and-error steps.

6. Conclusions

We have presented and evaluated a Clustered Multiple Clock Domain microprocessor, as well as a mechanism for dynamically adapting the frequency and voltage of the frontend domain of the CMCD. The design is based on a conventional clustered architecture where the frontend, the L2, and each backend operate at different clock domains. Our simulations show that the baseline CMCD architecture has acceptable IPC degradation (less than 3% on average) and an average power reduction of about 12.5% just due to the simplification of the clock distribution network.

Our dynamic frontend adaptation mechanism, based on queuing theory, has minimal hardware overhead and no dependence on thresholds, so it needs no fine-tuning. It automatically adapts to the characteristics of each application. Our simulations show that it can achieve energy-delay^2 improvement of about 17.5% on average, which is at least 80% of the improvement that could be achieved with optimal adaptation.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Architectures. In *27th Annual Intl. Symp. on Computer Architecture*, June 2000.

[2] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *33rd Annual Intl. Symp. On Microarchitecture*, Dec. 2000.

[3] D. Brooks, V. Tiwari and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimization. In *27th Annual Intl. Symp. on Computer Architecture*, June 2000.

[4] R. Canal, J.M. Parcerisa and A. González. Dynamic Cluster Assignment Mechanisms. In *6th Intl. Symp. on High-Performance Computer Architecture*, Jan. 2000.

[5] K.I. Farkas, P. Chow, N.P. Jouppi and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *30th Annual Intl. Symp. on Microarchitecture*, Dec. 1997.

[6] J. González and A. González. Dynamic Cluster Resizing. In *21st Intl. Conf. on Computer Design*, Oct. 2003.

[7] A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *29th Annual Intl. Symp. on Computer Architecture*, May 2002.

[8] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), March/April 1999.

[9] G. Magklis, M.L. Scott, G. Semeraro, D.H. Albonesi and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Processor. In *30th Annual Intl. Symp. on Computer Architecture*, June 2003.

[10] S. Palacharla. Complexity-Effective Superscalar Processors. Ph.D. Thesis, University of Madison, 1998.

[11] J-M. Parcerisa, J. Sahuquillo, A. González and J. Duato. Efficient Interconnects for Clustered Microarchitectures. In *11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2002.

[12] P.J. Restle et al. A Clock Distribution Network for Microprocessors. *IEEE Journal of Solid-State Circuits*, 36(5), May 2001.

[13] 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 In *35th Annual Intl. Symp. on Microarchitecture*, Nov. 2002.

[14] G. Semeraro, D.H. Albonesi, G. Magklis, M.L. Scott S. Dropsho and S. Dwarkadas. Hiding Synchronization Delays in a GALS Processor Microarchitecture. In *10th Intl. Symp. on Asynchronous Circuits and Systems*, April 2004.

[15] 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. In *8th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2002.

[16] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Research Report 2001/2, Aug. 2001.

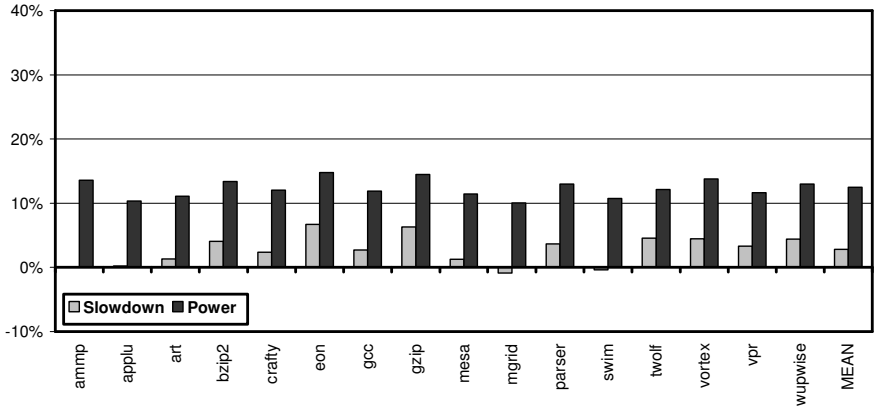

Figure 5. CMCD compared to a fully-synchronous design

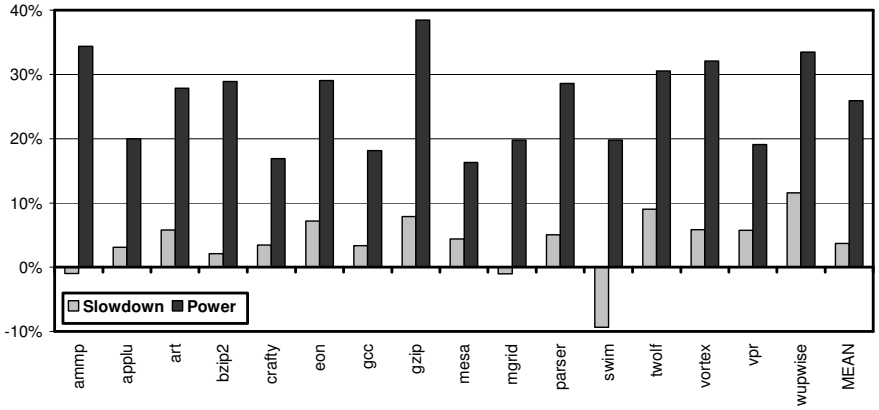

Figure 6. Adaptive CMCD compared to a fully-synchronous, non-adaptive design

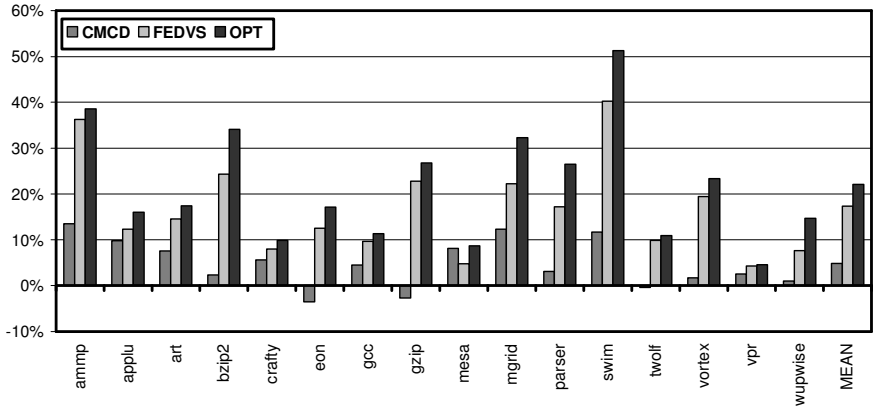

Figure 7. CMCD, adaptive CMCD, and optimal adaptation