Dynamic Frequency and Voltage Scaling for a Multiple-Clock-Domain Microprocessor

Multiple clock domains is one solution to the increasing problem of propagating the clock signal across increasingly larger and faster chips. The ability to independently scale frequency and voltage in each domain creates a powerful means of reducing power dissipation.

Grigorios Magklis
Intel

Greg Semeraro
Rochester Institute of Technology

David H. Albonesi
Steven G. Dropsho
Sandhya Dwarkadas
Michael L. Scott
University of Rochester

Demand for higher processor performance has led to a dramatic increase in clock frequency as well as an increasing number of transistors in the processor core. As chips become faster and larger, designers face significant challenges, including global clock distribution and power dissipation.

A multiple clock domain (MCD) microarchitecture,[1] which uses a globally asynchronous, locally synchronous (GALS) clocking style,[2,3] permits future aggressive frequency increases, maintains a synchronous design methodology, and exploits the trend of making functional blocks more autonomous. In MCD, each processor domain is internally synchronous, but domains operate asynchronously with respect to one another. Designers still apply existing synchronous design techniques to each domain, but global clock skew is no longer a constraint. Moreover, domains can have independent voltage and frequency control, enabling dynamic voltage scaling at the domain level.

Global dynamic voltage scaling already appears in many systems and can help reduce power dissipation for rate-based and partially idle workloads. An MCD architecture can save power even during intensive computation by slowing domains that are comparatively unimportant to the application's current critical path, even when it is impossible to completely gate off those domains. The disadvantage is the need for interdomain synchronization, which, because of buffering, out-of-order execution, and superscalar data paths, has a relatively minor impact on overall performance, less than 2 percent.[4]

MCD potentially has a significant energy advantage with only modest performance cost, if the frequencies and voltages of the various

Published by the IEEE Computer Society 0272-1732/03/$17.00 © 2003 IEEE

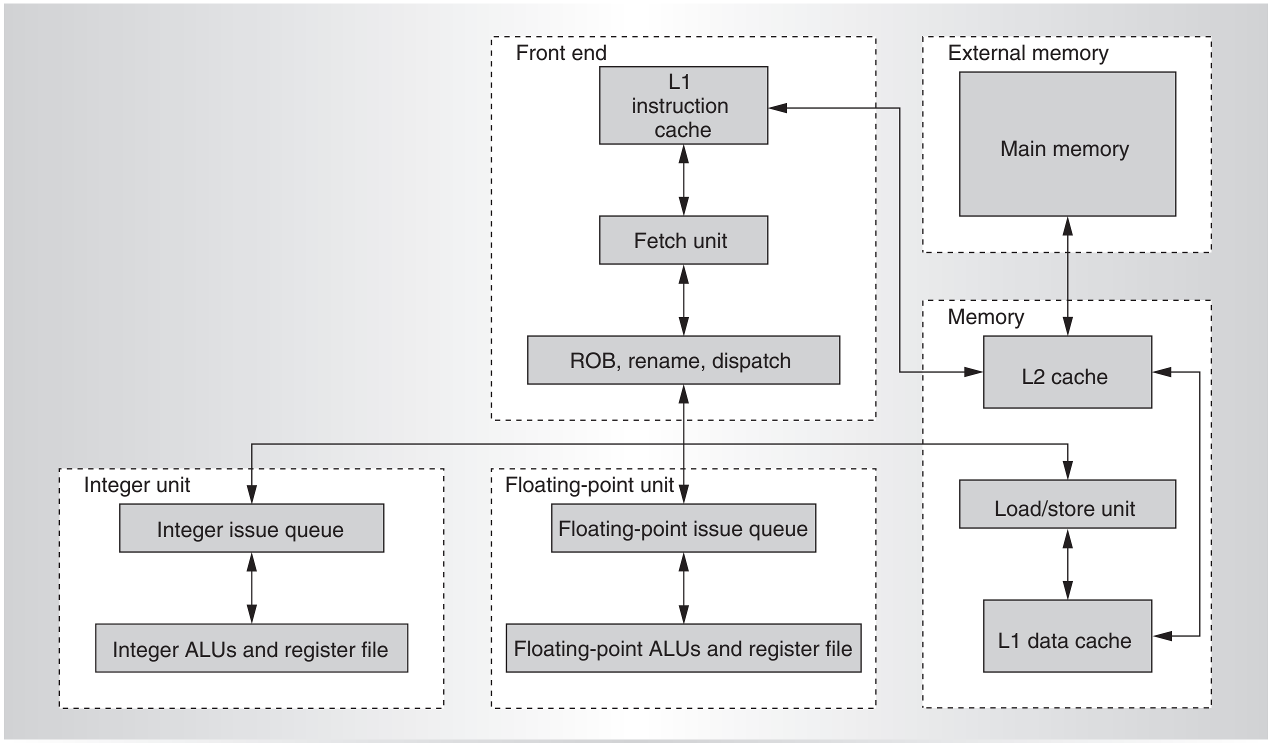

Figure 1. MCD processor block diagram.

domains assume appropriate values at appropriate times.[1] Designers can implement this control function completely online in hardware, making it transparent to the user and system software.[4] Online control is useful in environments where legacy applications must run without modification, or significant user involvement is undesirable. Otherwise, profiling and instrumentation of the application provides a more global view of the program than in a hardware implementation, and has the potential to provide better results, if the behavior observed during the profiling run is consistent with that occurring in production.[5] This article briefly summarizes both of these approaches and compares their performance against a near-optimal offline technique.

MCD microarchitecture

The MCD microarchitecture[1] consists of four different on-chip clock domains, shown in Figure 1, each with independent control of frequency and voltage. In choosing the boundaries among domains, we identified points where

- there already existed a queue structure that decoupled different pipeline functions or
- relatively little interfunction communication occurred.

Main memory is external to the processor, and we can view it as a fifth domain that always runs at full speed.

We based our frequency and voltage-scaling model on the Intel XScale processor (as described by L.T. Clark in the short course "Circuit Design of Xscale Microprocessors," at the 2001 Symp. VLSI Circuits). The XScale continues to execute through the voltage/frequency change. There is, however, a substantial delay before the change becomes fully effective.

Key to MCD's fine-grained adaptation is efficient, on-chip voltage scaling circuitry, a rapidly emerging technology. New microinductor technologies are paving the way for highly-efficient, on-chip, buck converters.[6] This circuit technology should be mature enough for commercialization within the next few years, and the MCD microarchitecture, including the voltage control algorithms we present, will be ready to take advantage of the technology.

Online control algorithm

Analysis of processor resource utilization reveals a correlation, over an interval of instructions, between the valid entries in the input queue (for each of the integer, floating-point, and load/store domains) and the desired frequency for the domain. This correlation follows from considering the instruction processing core as the domain queue's sink and the front end as the source. Queue utilization indicates the rate at which instructions flow through the core; if utilization increases, instructions are not flowing fast enough. Queue utilization is thus an appropriate metric for dynamically determining the desired domain frequency (except in the front-end domain, which the online algorithm does not attempt to control).

This correlation between issue queue utilization and desired frequency is not without challenges. Notable among them is that changes in a domain's frequency might affect the issue queue utilization of that domain and possibly others. This interaction among the domains is a potential source of error that might degrade performance beyond acceptable thresholds or lead to lower-than-expected energy savings. Interactions might lead to instability in domain frequencies, as changes in the other domains influence each particular domain.

The online algorithm consists of two components that act independently but cooperatively. The result is a frequency curve that approximates the envelope of the queue utilization curve, creating a small performance degradation and a significant energy savings. In general, an envelope detection algorithm reacts quickly to sudden changes in the input signal (queue utilization, in this case). In the absence of significant changes, this algorithm slowly decreases the controlling parameter.

Such an approach represents a feedback control system. For a control system, if the plant (the entity under control) and the control point (the parameter being adjusted) are linearly related, then the system will be stable, and the control point will correctly adjust to changes in the plant. Because of the rapid adjustments necessary for significant changes in utilization and the otherwise slow adjustments, we call the approach an attack/decay algorithm.[4] The attack-decay-sustain-release (ADSR) envelope-generating techniques in signal processing and signal synthesis inspired this algorithm.[7]

The MCD architecture employs the attack/decay algorithm independently in each back-end domain. The hardware counts the entries in the domain issue queue over a 10,000-instruction interval. Using that number and the corresponding number from the prior interval, the algorithm determines if there has been a significant change (a threshold of 1.75 percent), in which case the algorithm uses the attack mode: The frequency changes (up or down as appropriate) by a modest amount (6 percent). If no significant change occurs or if there is no activity in the domain, the algorithm uses the decay mode: It decreases the domain frequency slightly (0.175 percent).

In all cases, if the overall instructions per cycle (IPC) changes by more than a certain threshold (2.5 percent), the frequency remains unchanged for that interval. This convention identifies natural decreases in performance that are unrelated to the domain frequency and prevents the algorithm from reacting to them. Thresholding tends to reduce the interaction of a domain with adjustments in other domains. The IPC performance counter is the only global information that is available to all domains.

To protect against settling at a local minimum when a global minimum exists, the algorithm forces an attack whenever a domain frequency has been at one extreme or the other for 10 consecutive intervals. This is a common technique to apply when a control system reaches an end point and the plant/control relationship becomes undefined.

Profile-based control algorithm

The profile-based control algorithm has four phases: It

- uses standard performance profiling techniques to identify subroutines and loop nests that run long enough to justify reconfiguration;
- constructs a directed acyclic graph (DAG) that represents dependences among domain operations in these long-running fragments of code, and distributes the slack in the DAG to minimize energy;

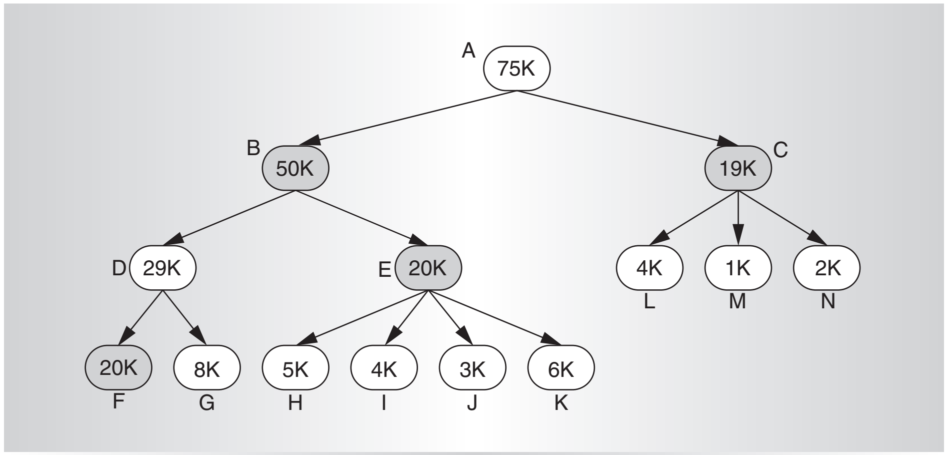

Figure 2. Call tree with associated instruction counts. The shaded nodes are candidates for reconfiguration.

- uses per-domain histograms of operating frequencies to identify, for each long-running code fragment, the minimum frequency for each domain that would permit execution to complete within a fixed slow-down bound; and
- edits the application's binary code to embed path-tracking and reconfiguration instructions that will instruct the hardware to adopt appropriate frequencies at appropriate times during production runs.

Choosing reconfiguration points

Phase one uses a binary editing tool[8] to instrument subroutines and loops. When the instrumented binary executes, it counts when each subroutine or loop executes in a given context. In the most general case we considered, a call tree that captures all call sites between the main and the current point of execution can represent a context. The call tree differs from the static call graph that many compilers construct because it has a separate node for every path over which a given subroutine is reachable (it will also be missing any nodes that the profiling tool did not encounter during its run). The call tree is not a true dynamic call trace, but a compressed one, which superimposes multiple instances of the same path. For example, if the program calls a subroutine from inside a loop, this loop will have the same call history every time, and can be represented by a single node in the tree, even though the program might have actually called it many times.

After running the binary code and collecting its statistics, we annotate each tree node with the dynamic instances and the total instructions executed, from which we can calculate the average instructions per instance (including instructions executed in the node's children). We then identify all nodes that run long enough (10,000 instructions or more) for a frequency change to take effect and to have a potential impact on energy consumption.

Starting from the leaves and working up, we identify all nodes whose average instance (excluding instructions executed in long-running children) exceeds 10,000. Figure 2 shows a call tree; the long-running nodes are shaded. Note that these nodes, taken together, are guaranteed to cover almost all of the application history in the profiled run.

The shaker algorithm

To select frequencies and corresponding voltages for long-running tree nodes, we run the application through a heavily modified version of the SimpleScalar/Wattch tool kit,[9,10] with all clock domains at full frequency. During this run, in phase two, we collect a trace of all primitive events (temporally contiguous work performed within a single hardware unit on behalf of a single instruction), and the functional and data dependences among these events. The trace output is a dependence DAG for each long-running node in the call tree. Working from this DAG, the shaker algorithm attempts to "stretch" (make longer in time) individual events that are not on the application's critical execution path, as if they could run at their own, event-specific, lower frequency.

Whenever an event in a dependence DAG has two or more incoming arcs, it is likely that one arc constitutes the critical path and that the others will have slack. Slack indicates that the previous operation completed earlier than necessary. If all of the outgoing arcs of an event have slack, then we have an opportunity to save energy by performing the event at a lower frequency. With each event in the DAG, we associate a power factor whose initial value is based on the relative power consumption of the corresponding clock domain in our processor model. When we stretch an event, we scale its power factor accordingly.

The shaker tries to distribute slack as uniformly as possible. It begins at the end of the DAG and works backward. When it encounters a stretchable event whose power factor exceeds the current threshold (originally set to be slightly below that of the few most power-intensive events in the graph) the shaker scales the event until it either consumes all the available slack or its power factor drops below the current threshold. If any slack remains, the event moves later, so that as much slack as possible occurs at its incoming edges.

When the shaker reaches the beginning of the DAG, it reverses direction, reduces its power threshold by a small amount, and makes a new pass forward through the DAG, scaling high-power events and moving slack to outgoing edges. It repeats this back-and-forth process until all the available slack is consumed, or until all the events adjacent to slack edges have been scaled to the minimum permissible frequency. When it completes its work, the shaker constructs a per-domain summary histogram that indicates, for each of the frequency steps, the total cycles for events in the domain that have been scaled to run at or near that frequency. A combination of the histograms for multiple dynamic instances of the same tree node then becomes the input to the slow-down thresholding algorithm.

Slow-down thresholding

Phase three recognizes that we cannot in practice scale the frequency of individual events: We must scale each domain as a whole. If we are willing to tolerate a small performance degradation, *d*, we can choose a frequency that causes some events to run slower than ideal. Using the histograms generated by the shaker algorithm, we choose a frequency based on all the events in higher bins of the histogram. For the chosen frequency, the extra time necessary to execute those events must be less than or equal to *d* percent of the total time required to execute all the events in the node, run at their ideal frequencies.

Application editing

In phase four, to effect the reconfigurations chosen by the slow-down thresholding algorithm, we must insert code at the beginning and end of each long-running subroutine or loop. Although the instrumentation overhead necessary to track the full definition of context is low (about 9 extra cycles for each 10,000-instruction interval, plus 8 more cycles if the frequency requires changing), simply tracking the program counter yields results that are almost as accurate as in this full definition.

The results we present associate a single desired frequency with each long-running subroutine or loop, regardless of calling context. At the beginning of each such code fragment, the instrumented binary writes a statically known frequency into an MCD hardware reconfiguration register. More complex definitions of context require additional instrumentation as well as a lookup table containing the frequencies chosen by the slow-down thresholding algorithm of phase three.[5]

Results

We assume a processor microarchitecture similar to that of the Alpha 21264 with a frequency range of 250 MHz to 1 GHz and a corresponding voltage range of 0.65 V to 1.2 V. Traversing the entire voltage range requires 55 s. We select applications from the MediaBench and SPEC CPU2000 suites. For the profile-based approach, we use a smaller training input data set during profiling, but gather final results using the larger reference data set.

The MCD processor has an inherent performance penalty of less than 2 percent compared to its globally clocked counterpart, and an energy penalty of about 1 percent. Figure 3 shows energy × delay improvements for the online and profile-based algorithms relative

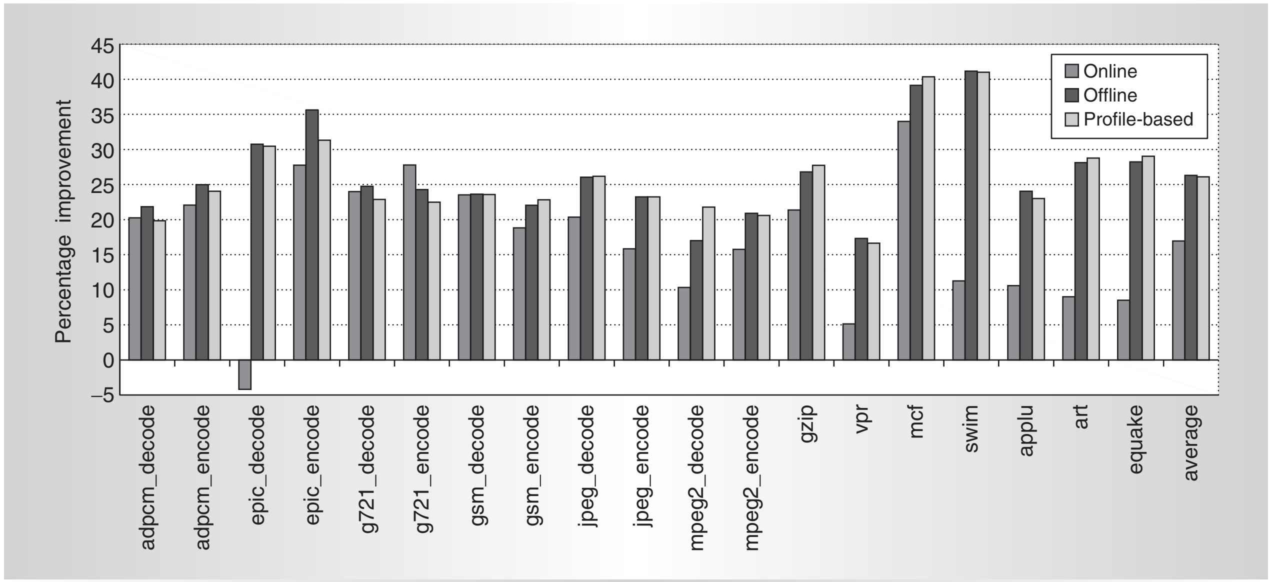

Figure 3. Energy × delay improvement results.

to this baseline MCD processor with no voltage control. We obtain the so-called offline results with perfect future knowledge.[1]

For all three control strategies, the average performance degradation (not shown) is approximately 7 percent. The online algorithm achieves a significant overall energy × delay improvement, about 17 percent, although its reactive nature results in a slight degradation for one benchmark. As expected, profiling yields better and more consistent results, about a 27 percent overall energy × delay improvement, nearly matching that of the omniscient offline algorithm.

The MCD approach alleviates many of the bottlenecks of fully synchronous systems, while exploiting proven synchronous design methodologies. The union of the MCD microarchitecture with emerging on-chip voltage scaling technology permits fine-grained voltage scaling that is broadly applicable. Both the online and profile-based techniques that we have developed exploit this capability to provide significant energy savings. MICRO

References

1. G. Semeraro et al., "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," *Proc. 8th Int'l Symp. High-Performance Computer Architecture* (HPCA 02), IEEE CS Press, 2002, pp. 29-40.
2. D.M. Chapiro, "Globally Asynchronous Locally Synchronous Systems," PhD thesis, Stanford Univ., 1984.
3. A. Iyer and D. Marculescu, "Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 158-170.
4. G. Semeraro et al., "Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture," *Proc. 35th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO-35), IEEE CS Press, 2002, pp. 356-370.
5. G. Magklis et al., "Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), ACM Press, 2003, pp. 14-27.
6. V. Kursun et al., "Analysis of Buck Converters for On-Chip Integration with a Dual Supply Voltage Microprocessor," *IEEE Trans. VLSI Systems*, vol. 11, no. 3, June 2003, pp. 514-522.
7. K. Jensen, "Envelope Model of Isolated Musical Sounds," *Proc. 2nd COST G-6 Workshop on Digital Audio Effects* (DAFx99), Norwegian University of Science and Technology, 1999, pp. W99-1–W99-5.
8. A. Eustace and A. Srivastava, "ATOM: A

Flexible Interface for Building High Performance Program Analysis Tools," *Proc. Usenix 1995 Technical Conf.*, Usenix Assoc., 1995, pp. 303-314.

9. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 83-94.
10. D. Burger and T. Austin, *The SimpleScalar Tool Set, Version 2.0*, technical report CS-TR-97-1342, Computer Science Dept., Univ. of Wisconsin, June 1997.

Grigorios Magklis is a researcher at the Intel-UPC Barcelona Research Center. His research interests include architecture, operating systems, application analysis, and tools. Magklis is a PhD candidate and has an MSc in computer science from the University of Rochester. He is a member of the ACM and IEEE.

Greg Semeraro is an assistant professor in the Department of Computer Engineering at the Rochester Institute of Technology. His research interests include the modeling, analysis, and simulation of microarchitecture; digital and real-time systems; and nonlinear control systems. Semeraro has a PhD in electrical and computer engineering from the University of Rochester. He is a member of the IEEE Computer Society, the IEEE Education Society, and the American Society for Engineering Education.

David H. Albonesi is an associate professor in the Department of Electrical and Computer Engineering at the University of Rochester. His research interests include microarchitecture with an emphasis on adaptive architectures, power-aware computing, and multithreading. Albonesi has a PhD from the University of Massachusetts at Amherst. He is a senior member of the IEEE, and a member of the IEEE Computer Society and the ACM.

Steven G. Dropsho is a postdoctoral researcher in the Department of Computer Science at the University of Rochester. His research interests include architecture, power efficiency, and parallel and distributed systems. Dropsho has a PhD in computer science from the University of Massachusetts at Amherst. He is a member of the IEEE Computer Society and the ACM.

Sandhya Dwarkadas is an associate professor in the Department of Computer Science at the University of Rochester. Her research interests include parallel and distributed computing, computer architecture, and networks, and the interactions among and interfaces between the compiler, runtime system, and underlying architecture. Dwarkadas has a PhD in electrical and computer engineering from Rice University. She is a member of the IEEE, the IEEE Computer Society, and the ACM.

Michael L. Scott is a professor of computer science at the University of Rochester. His research interests include operating systems, languages, architecture, and tools, with a particular emphasis on parallel and distributed systems. He has a PhD in computer sciences from the University of Wisconsin-Madison. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Direct questions and comments about this article to David H. Albonesi, Computer Studies Bldg., University of Rochester, PO Box 270231, Rochester, NY 14627-0231; albonesi@ece.rochester.edu.