NePSim: A Network Processor Simulator with a Power Evaluation Framework

THIS OPEN-SOURCE INTEGRATED SIMULATION INFRASTRUCTURE CONTAINS A CYCLE-ACCURATE SIMULATOR FOR A TYPICAL NETWORK PROCESSOR ARCHITECTURE, AN AUTOMATIC VERIFICATION FRAMEWORK FOR TESTING AND VALIDATION, AND A POWER ESTIMATION MODEL FOR MEASURING THE SIMULATED PROCESSOR'S POWER CONSUMPTION.

Yan Luo, Jun Yang, Laxmi N. Bhuyan, and Li Zhao
University of California, Riverside

Network processors recently emerged as a successful platform providing both high performance and flexibility in building powerful routers. A typical network processor (NP) includes multiple fast-execution cores on a single chip that exploits packet-level parallelism. Example products include Intel's IXP family,[1] IBM's PowerNP, Motorola's C-Port, and Agere's APP550. With the exponential increase in clock frequency and core complexity, power dissipation will become a major design consideration in NP development. For example, a typical router configuration might include one or two NPs per line card. A group of line cards, say 16 or 32, generally resides within a single rack or cabinet.[2] Because each NP typically consumes approximately 20 W and the operating temperature can reach 70 degrees centigrade, aggregated heat dissipation becomes a major concern.

Research into NPs and their power dissipation is in its infancy. The main obstacle is the lack of open-source infrastructure for analyzing and quantifying the performance and power ramifications among NP architecture alternatives. Although there have been cycle-accurate architecture simulators for commercial NPs—for example, Intel's Software Development Kit (SDK)[3] and Motorola's C-Ware—their internal architecture design is not open source for users. Moreover, these simulators do not incorporate power modeling and evaluation tools.

This article presents NePSim, an integrated system that includes a cycle-accurate architecture simulator, an automatic formal verification engine, and a parameterizable power estimator for NPs consisting of clusters of multithreaded execution cores, memory controllers, I/O ports, packet buffers, and high-speed buses. To perform concrete simulation and provide reliable performance and power analysis, we defined our system to comply with Intel's IXP1200 processor specification because academia has widely adopted it as a representative model for NP research. The verification engine, called Iveri,[4] can validate the simulation from the execution log traces by employing user-defined constraints in a verification language. The power estimator combines a suite of models (Xcacti,[5] Wattch,[6] and Orion[7]) for dynamic power measure-

Published by the IEEE Computer Society

ments and calculates results on the basis of per-cycle resource usage counts.

In addition, we propose low-power techniques tailored to NPs and using our NePSim system. We adopted the classic dynamic voltage scaling (DVS) technique to each execution core, observing that there is abundant idle time (on average, 10 percent to 23 percent) resulting from contention in the shared memory. Overall, we achieve a maximum of 17 percent power savings for the NP over four network application benchmarks, with less than a 6 percent performance loss.

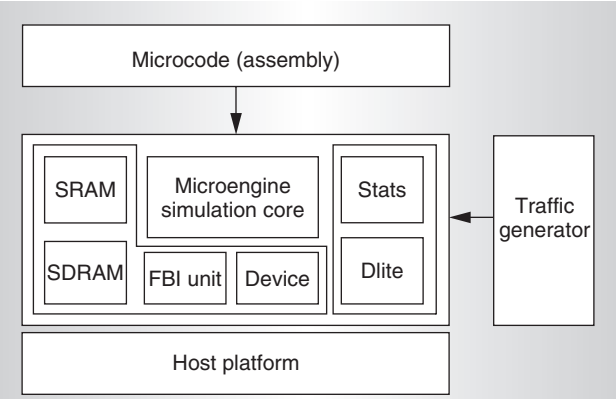

Figure 1. NePSim software structure.

Cycle-level simulation

We start with a high-level overview of the IXP1200, then we describe our simulator software structure.

Background: Intel IXP1200 and its microengines

The IXP1200 is an integrated processor comprising a single StrongARM processor, six microengines (MEs), standard memory interfaces, and high-speed bus interfaces. The StrongARM core manages functions such as initializing MEs, running routing protocols, and handling exceptions. The MEs are fast RISC engines that can be programmed to perform high-speed packet inspection, data manipulation, and data transfer. The SDRAM unit is a shared-memory interface accessible by the StrongARM, the MEs, and devices on the PCI bus. SDRAM serves mostly for temporarily storing packet payloads. The SRAM unit is also a shared-memory interface accessible by the StrongARM and the MEs. SRAM serves mostly for storing control data structures such as forwarding or routing tables. The IX bus unit, controlled by MEs, transfers data blocks between the IXP1200 and networking devices such as media access controllers (MACs). Intel's reference manual describes the IXP1200 in detail.[1]

The simulator

Our simulator implements most functionalities of the IXP1200, the external SDRAM, and the SRAM. We also implemented a full-fledged command bus arbiter and an IX bus unit, which contains the scratchpad memory and packet I/O buffers. We did not model the StrongARM core because its main task is control plane functions that don't affect the critical path of packet processing.

Figure 1 shows the NePSim hardware model's software architecture. We first compile the applications into microcode assembly. We didn't use the binary directly because we wanted to leave room for instruction extensions in future research. It's usually easier to modify the program using assembly rather than binary. NePSim's inputs are network packet streams generated from the traffic generator. Ideal packet streams are those collected from real-world networks or downloaded from trace archive sites such as the National Laboratory for Applied Network Research. If these aren't available for an application, the traffic generator module will generate a synthetic trace of packets.

The NePSim body (the microengine simulation core) is the module that simulates the following five stages of the ME pipeline:

1. instruction lookup;
2. initial instruction decoding and formation of the source register address;
3. reading of operands from the source registers;
4. ALU operations, shift or compare operations, and generation of condition codes; and
5. writing of result to destination register.

An ME's threads share the pipeline and functional units such as the ALU. Thread execution is not preemptive, which means a thread cannot gain control of the pipeline unless the running thread yields control. Mutual

exclusion mechanisms include absolute register and atomic scratchpad operations.[1]

The lack of an IXP1200 timing specification complicates modeling the memory module and memory controllers. The SRAM and SDRAM memory controllers consist of multiple reference-command queues with different priorities. The arbiter in the controller schedules the memory reference commands to achieve high throughput. However, the documents available to us do not explicitly state the command enqueue and service time. Our methodology here is to analyze memory access traces from Intel's SDK, which contains detailed cycle information for all possible events, such as memory access issues and commits. From the traces, we can extract relatively accurate timing relations among different events. On the basis of these relations, we implemented the memory units to match NePSim with the IXP1200 as closely as possible.

The device module in Figure 1 implements I/O devices such as I/O ports and the MACs. The Dlite (lightweight debugger) module resembles the debugger in SimpleScalar. For example, it lets users set breakpoints, print pipeline status, display register values, and dump memory content.

The simulator is highly parameterized, enabling users to model components with different configurations. For example, the MEs can be set to run at different clock rates and supply voltages. Users can configure the SRAM and SDRAM with different latencies and bandwidths, and they can define incoming traffic with different arrival rates and arrival patterns—for example, Poisson, uniform, or random.

Our simulator provides several other advantages over Intel's SDK.

Enables new architecture design. This is probably the strongest motivation for developing NePSim. If an SDK user wants to change the behavior of certain hardware to observe any differences, it might be necessary to modify the program being run. This is certainly intrusive and not trustworthy. NePSim gives users maximum freedom by letting them test any modification in the architecture with ease.

Permits number of MEs and threads to vary. In addition to modeling a varying number of MEs, NePSim models a varying number of threads per ME. The MicroC compiler can create six or more ME executables, which can be fed into NePSim. Intel's SDK for the IXP1200, however, can't run more than six MEs. This feature lets us explore future NP architectures with even more MEs, such as the IXP2400 or the IXP2800.

Provides instruction set extensibility in microcode assembly code. Users can tag instructions with flags when experimenting with an instruction set architecture optimization. Also, users can create new instructions and insert them into the assembly code without fear of modifying PC-relative or offset fields in other instructions. SDK cannot achieve this functionality.

Provides faster execution speed. Like SDK, NePSim has a comprehensive set of statistical variables. Unlike SDK, however, the simulator lets us collect data selectively, thereby increasing the simulation speed. On average, we can simulate 120 K instructions per second (on a 900-MHz Pentium III machine), which is about 10 times faster than the SDK.

Validation

Validating NePSim against the IXP1200 architecture is crucial because the performance and power simulation is useful only if it is satisfactorily accurate. Verifying NePSim's functional properties by comparing program results is relatively simple. However, performance verification requires significant effort to make this process accurate, efficient, and formal.

We built a verification backend tool called Iveri that formally specifies the simulator properties in a verification language. First, we run the same benchmarks on NePSim and SDK to get two simulation traces. We treat the SDK trace as the standard reference trace. The log traces contain events annotated with information in a predefined format. Iveri processes the log traces and prints a detailed report stating how many times a property is violated and where it is violated in the log trace. Using this information, we go back to the simulator and trace the simulation until we discover a bug. Thus, through Iveri, we can formally and accurately describe the performance properties or requirements of a new

design. Chen et al. provide a detailed specification for Iveri.[4]

Next, we compare the verification results of overall NePSim throughput and average processing time against those of the IXP1200. (We provide verification results of individual major components such as SRAM and SDRAM in a longer version of this article published as a technical report.[8]) We ran four benchmarks ported to NePSim and measured the performance in terms of throughput (Mbps) and average packet processing time (cycles). The benchmark descriptions appear later in this article. Figures 2 and 3 show the comparisons. We observed an average error of 1 percent in throughput and 6 percent in average processing time across the four benchmarks. The error in average processing time is due mainly to the inaccuracy of our SDRAM model. Such a timing error affects the packet processing order and time. Overall, though, we think the simulation can produce relatively dependable results.

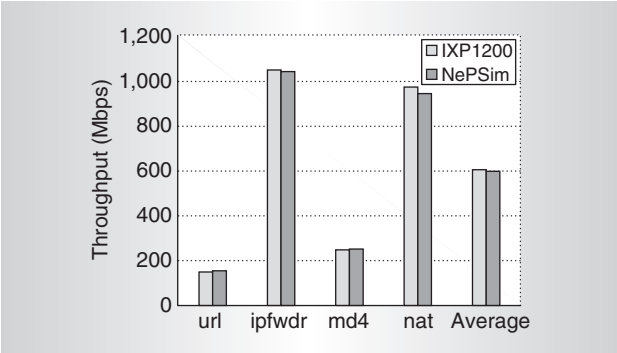

Figure 2. NePSim and IXP1200 throughput comparison.

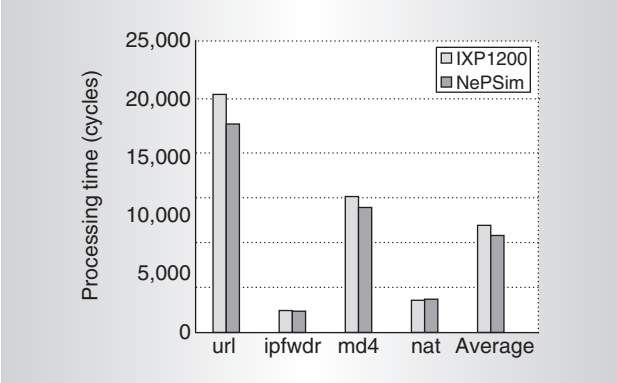

Figure 3. NePSim and IXP1200 processing time comparison.

Power modeling

We extracted and combined various power models from Xcacti,[5] Wattch,[6] and Orion[7] for different hardware structures in NePSim. XCacti is an enhanced version of Cacti 2.0 that includes power modeling for cache writes, misses, and writebacks. We first classify the IXP1200 hardware component structures into five categories: ALU and shifter, registers, cachelike structures, queues, and arbiters. Registers include general-purpose register files, transfer register files, and control register files for each ME. Cachelike structures include the control store for each ME and the scratchpad memory in the IX bus unit. Queue-type structures include each ME's command FIFO, receive and transmit FIFOs in the IX bus unit, and groups of queues in the SDRAM and SRAM units. The arbiter type includes each ME's context event arbiter, the command bus arbiter, and reference arbiters in the SDRAM and SRAM units. The IXP1200 uses 0.28-μm technology. However, Xcacti, Wattch, and Orion do not supply the necessary scaling factors for 0.28 μm. (The scaling factors serve to scale wire capacitance, resistance, transistor length, area, voltage, threshold voltage, and sense voltage.[6,7,9]) Therefore, we use 0.25-μm technology because it is the closest available feature size to 0.28 μm, and we should expect to obtain lower overall power consumption than that reported for the IXP1200. Table 1 lists the components we modeled, the tools we adopted, and the corresponding configurations.

Only the IXP1200 chip's total power consumption was available. To the best of our knowledge, each component's power breakdown has never been reported. Therefore, we validate our modeling on the basis of the total power. According to the IXP1200 datasheet, the IXP core's power, excluding I/O, is typically 4.5 W at 232 MHz, with a 2-V supply voltage and 0.28-μm technology. This wattage includes all the MEs; the memory units; the IX bus unit, which contains the control store; and the StrongARM core. The latter consumes an average of 0.5 W, leaving approximately 4 W for the rest of the chip. From our power models, each ME consumes 0.468 W, the IX bus unit (containing the scratchpad memory) consumes 0.363 W, the SRAM unit 0.0639 W, and the SDRAM unit

Table 1. Component modeling information.

Hardware component	Model type	Tool	Configuration
Two general-purpose register files per ME	Array	XCacti	Two 64-entry files, one read/write port per file
Four transfer register files per ME	Array	XCacti	Four 32-entry files, one read/write port per file
One control register file per ME	Array	XCacti	One 32-entry file, one read/write port
One control store per ME and scratchpad memory	Cache without tag path	XCacti	4 Kbytes, 4 bytes per block, direct mapped, 10-bit address
ALU and shifter	ALU and shifter	Wattch (cc1 clock-gating technique)	32 bits
One command FIFO per ME	Array	Wattch	Two-entry, 64-bit width
Receive FIFO, transmit FIFO			
High-priority queues			Eight-entry (SDRAM and SRAM)
Order queues			24-/16-entry (SDRAM/SRAM)
Odd-/even-bank queues			16-entry (SDRAM)
Read-only queues			16-entry (SRAM)
Command bus arbiter	Matrix and round-robin arbiter	Orion	Three priority requests, six round-robin requests (maximum)
One context arbiter per ME	Round-robin arbiter		Four round-robin requests
Reference arbiters	Matrix		Up to four priority requests

0.0643 W. These are average values. Assuming every component switches on in every cycle, the total power consumption is 0.468 × 6 + 0.363 + 0.0639 + 0.0643 = 3.3 W. The difference, 4.0 – 3.3 = 0.7 W, results from our use of a smaller technology, 0.25 μm instead of 0.28 μm. Also, there are other structures that we did not model—for example, internal buses and the clock. We will include these in our NePSim upgrade.

Power and performance analysis

Because NePSim can provide both power and performance statistics, we take this opportunity to study the NP power/performance tradeoffs that cannot be determined using Intel's SDK. We assume the maximum packet arrival rate because we are interested in an NP's processing capability. In addition, we assume 16 Ethernet interfaces for receiving packets and 16 for transmitting packets. The SRAM and SDRAM frequency is 116 MHz, according to the IXP1200 datasheet. The unloaded latencies for SRAM, SDRAM, and scratchpad memory are 16, 33, and 12 cycles, respectively, for a 232-MHz ME.

Benchmark descriptions

We compiled and ported four representative NP applications for our experiments: Internet Protocol (IP) forwarding (ipfwdr), URL routing (url), a message-digest algorithm (md4), and network address translation (nat). Each benchmark application uses four receiving MEs and two transmitting MEs. The threads on receiving MEs receive and process packets independently (unordered). The other two MEs transmit processed packets through output interfaces.

The ipfwdr is a sample of IP forwarding software provided in Intel's SDK.[3] This application implements an IPv4 router that forwards IP packets between networks. The processing includes Ethernet and IP header validation and trie-based routing-table lookup. The routing table resides in SRAM, and the output port information is in SDRAM. The rest of the MEs transmit the processed packets to the next-hop router on the basis of output port information.

The url routing program routes packets on the basis of their contained URL request. This is a primary task of content-aware switches

that often examine the payload of packets when processing them. The url then performs a string-matching algorithm that we ported from NetBench. Because the string patterns are initialized in SRAM, url's code must generate SRAM accesses in later comparisons. Also, because they must be scanned for pattern matching, many requests are generated to SDRAM, which stores payload data.

The network address translation program, nat, uses the source and destination IP addresses and port numbers to compute an index. This index serves as a hash-table lookup to retrieve a replacement address and port. Thus, each packet accesses the SRAM to look up the hash table. SDRAM accesses aren't necessary.

The md4 algorithm works on arbitrary-length messages and provides a 128-bit fingerprint, or digital signature. Designers use this algorithm to implement a Secure Sockets Layer or firewall at the edge routers. The algorithm moves data packets from SDRAM to SRAM and accesses SRAM multiple times to compute the digital signature. This program is both memory and computation intensive.

Researchers have studied ME distribution for receiving and transmitting data. For NPs such as the IXP1200, with six MEs, the usual configuration is four MEs (16 threads) for receiving and two MEs (eight threads) for transmitting. Researchers have tested this 4:2 ratio to provide maximum throughput, and we adopted this configuration throughout our experiments.

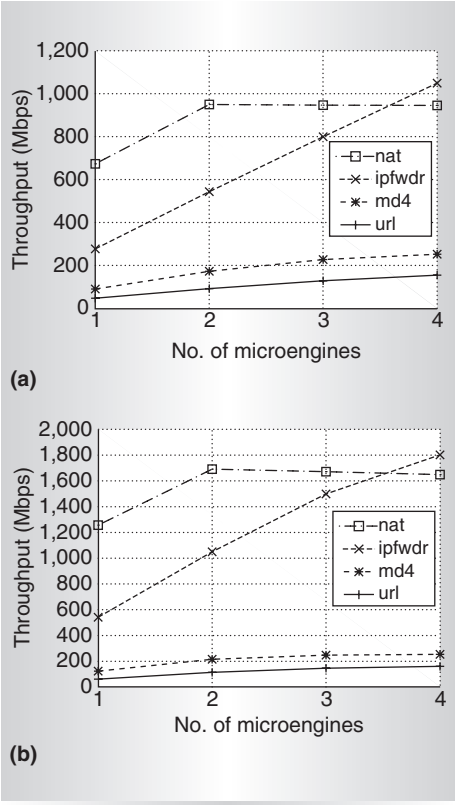

Figure 4. Impact of number of microengines on throughput: four threads per ME at 232 MHz (a), four threads per ME at 464 MHz (b).

Performance observations

In the first set of experiments, we studied the impact of having more MEs on the total packet throughput, as measured in Mbps. Intuitively, a benchmark's throughput should increase with the number of MEs or threads, because this configuration exploits higher packet-level parallelism. However, for memory-intensive benchmarks (url and md4), increasing the number of threads means increasing memory contention, because the memories are shared among all threads. Throughput tends to level off after a certain point, as shown in Figures 4a and 4b. Figure 4b also shows that for memory-intensive applications, doubling the core frequency doesn't double the throughput.

A strange result from these two graphs is the decrease in Mbps for nat as the number of MEs increases. It turned out that, unlike other programs, nat is not memory bound. To show this, we measured the average ME idle time corresponding to different ME-to-memory speed ratios—that is, 2:1 versus 4:1, as used in Figures 4a and 4b, respectively. When a program is memory bound, all the threads in an ME are often swapped out of pipeline waiting for their memory references, resulting in an idle ME. For such a program, the slower the memory, the longer the ME idle time.

As Figure 5 shows, nat doesn't have much ME idle time, even when the ME-to-memory speed ratio is 4:1. This implies that all MEs are busy, but, because the throughput declines, they are not producing much useful work. We traced nat's execution and found that the bottleneck lies in the transmitting threads. The receiving threads process packets fast enough, but they cannot allocate new memory slots for incoming packets because the transmitting threads

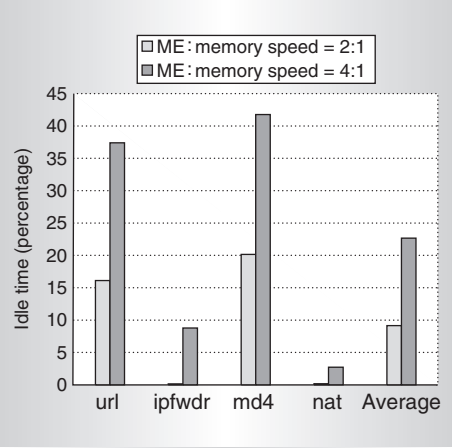

Figure 5. Average microengine idle time with different speed ratios for the core and the memory.

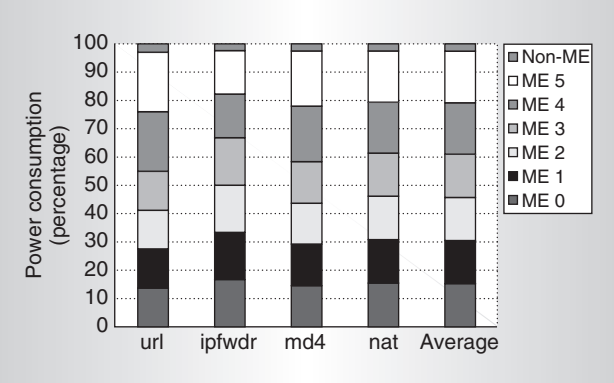

Figure 6. Simulated IXP1200 power consumption breakdown among microengines.

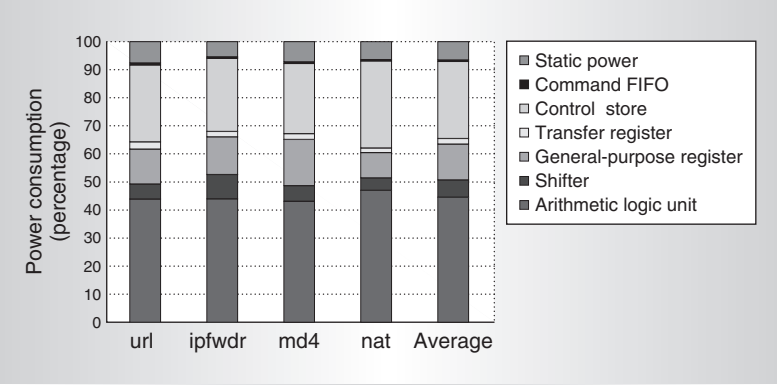

Figure 7. Microengine power consumption breakdown, by component.

don't release memory slots fast enough. The transmitting threads should release the memory slots once a packet has been sent. At this time, all the receiving threads are busy requesting new memory slots, but few can get them. This phenomenon suggests that a 3:3 receiving/transmitting ME ratio might work better for nat than the traditional 4:2 configuration.

Where does the power go?

The second set of experiments addresses IXP1200 and ME power consumption characteristics. In particular, we want to identify the components that consume the bulk of the power. Figure 6 plots the IXP1200's power distribution among the MEs, and Figure 7 plots the power consumed by an ME's individual components.

In Figure 6, MEs 0 through 3 are receiving MEs and MEs 4 and 5 are transmitting MEs. Almost all receiving MEs consume the same amount of power; the transmitting MEs consume about 5 percent more. The ALU consumes the most power (45 percent on average), followed by the control store (28 percent), where the program is stored. This is because the control store is accessed on almost every cycle. The general-purpose register (GPR) files, where instruction operands and results reside, constitute the third power-hungry component (13 percent). Even when data is loaded from the memory to the transfer register files, it is moved again to GPRs for ALU operations. We assumed that the control memory size (1,024 words) and the number of GPRs (128) that we set on NePSim were the same on the IXP1200. If these numbers vary, power consumption varies accordingly.

The fourth component is the static power. We included static-power estimation in Figure 7 because of the significant ME idle time. For 0.25-μm technology, the static power is less than 5 percent (we chose 2 percent in this experiment) of the dynamic power. The result in Figure 7 shows the static-power budget averaging about 7 percent. This indicates that even though the dynamic power of MEs dominates, future NPs might adopt smaller technology feature sizes and higher clock frequencies, making the static power more important when considering potential ME idle time.

Performance and power observations

Next, we want to determine whether the performance variation is consistent with the power variation. Particularly, with the trend toward

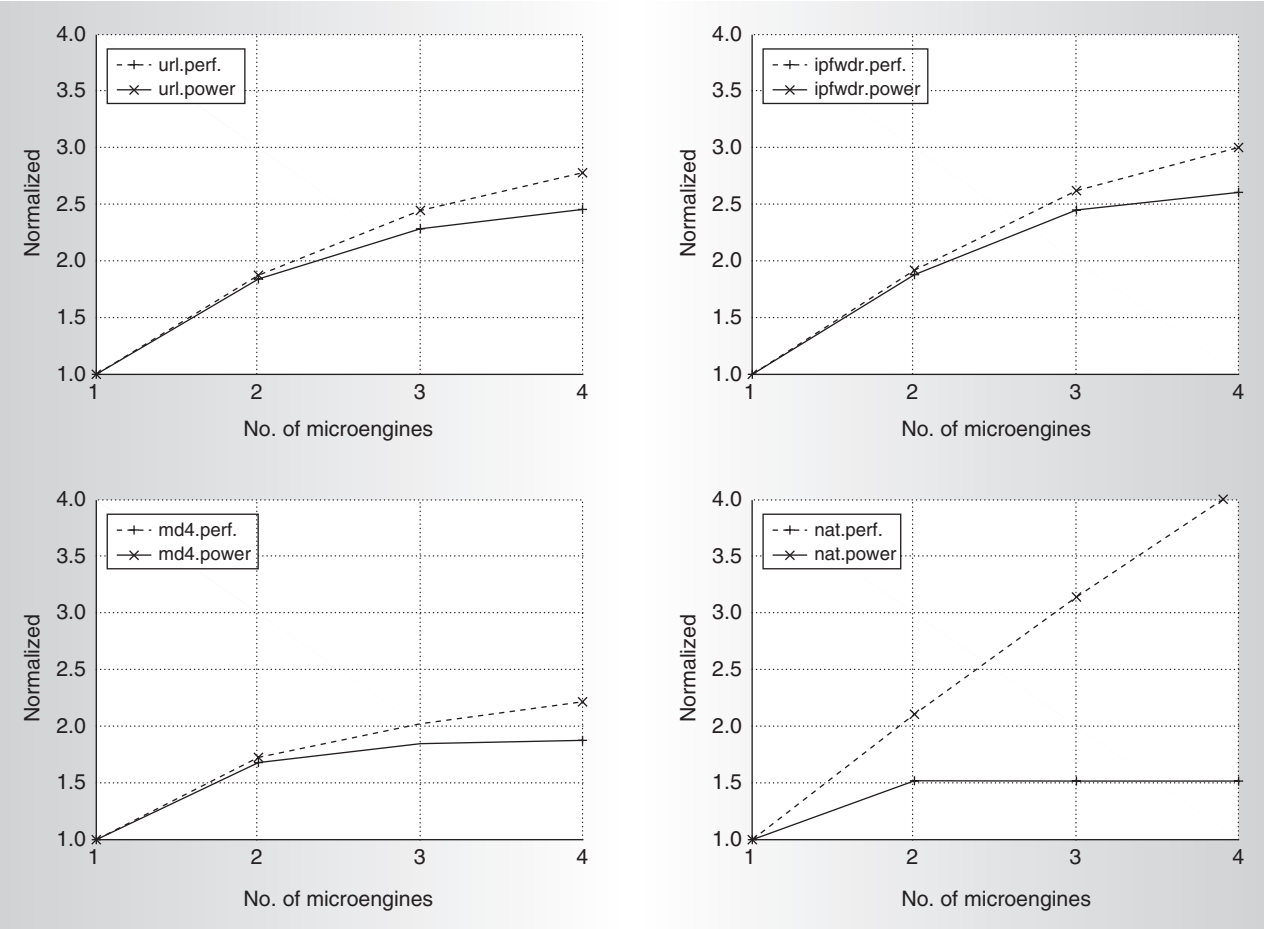

Figure 8. Analysis of power efficiency with a varying number of microengines, each with four threads, by application.

adding more processing capabilities on chip—for example, adding more cores—how do performance and power consumption respond? To discover this, we measured the normalized increase in performance and power for the four applications with a growing number of MEs. The core and memory frequency ratio is set at 4:1. The results appear in Figure 8.

These graphs show that both performance and power consumption grow with the addition of MEs, except for nat's performance (explained earlier). Our first observation is that ME power consumption doesn't multiply with an increase in the number of MEs. In other words, two MEs consume less than twice the power of one ME because ME idle time increases as MEs are added, since more threads are competing for the same amount of memory. We measured average ME idle time, varying the number of MEs. Figure 9 shows the results.

The most important observation from Figure 8 is the trend in the performance/power curve pair for each application. In each case, the gap between the two curves widens as the number of MEs increases. In other words, power consumption increases faster than performance. Franklin and Wolf reached similar conclusions using an analytical model.[2] (See the "Existing work on network processors" sidebar.) Therefore, designers of future NPs will find power consumption a greater constraint than processing capability. For this reason, we next present a technique that uses classic dynamic voltage scaling (DVS) to conserve power in MEs.

Reducing processing core power

Wide use of DVS for microprocessors has produced significant power savings. DVS exploits microprocessor utilization variance, reducing voltage and frequency when the processor has low activity and increasing them

Existing work on network processors

Franklin and Wolf developed an analytic performance model that captures a generic network processor's (NP's) processing performance and power consumption. The modeled prototypical NP contains several identical multithreaded general-purpose processors clustered together to share a memory interface. They used Wattch[1] and Cacti[2] to measure power consumption. Their extended research explored the NP design space and showed the performance/power impact of different systems.[3] NePSim's advantage over their model is that we target a real NP architecture (the IXP1200) and perform accurate cycle-level simulation and power modeling. We also tried to port network applications so that they could execute in NePSim as well as in the IXP1200. Therefore, our statistics are more realistic and dependable.

References

1. 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), IEEE CS Press, 2000, pp. 83-94.
2. CACTI, HP-Compaq Western Research Lab, http://research.compaq.com/wrl/people/jouppi/CACTI.html.
3. M. Franklin and T. Wolf, "Power Considerations in Network Processor Design," http://www.ecs.umass.edu/ece/wolf/papers/npw2003.pdf.

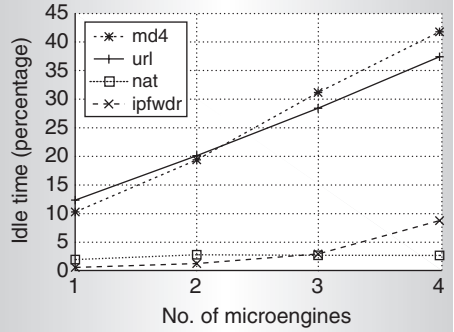

Figure 9. Average core idle time with different numbers of microengines and active threads (four threads per ME at 464 MHz).

when there's a demand for peak processor performance. In the work we've described so far, we identified a significant percentage of low-activity or idle time for MEs. Therefore, there is plenty of room for deploying DVS to conserve power.

DVS policy

Figure 9 shows that average ME idle time is abundant because most of the benchmarks are memory bound. During idle time, all threads are in a wait state, and the MEs run with their lowest level of activity. Applying DVS while MEs are not very active can reduce total power consumption substantially. Our scheme starts by using hardware to observe ME idle time periodically. Once the percentage of idle time in a past period exceeds a threshold, we scale down the voltage and frequency (VF) by one step, unless the ME hits the minimum allowable VF. If the percentage is below the threshold, indicating that in the past period the ME was relatively busy, we scale up the VF by one step, unless they are already at the maximum allowable values.

There are four issues to consider:

- transition step, meaning whether to use continuous or discrete changes in VF;
- transition status, indicating whether we let the ME continue working during VF regulation;
- transition time between two different VF states; and
- transition logic complexity, which is the overhead of the control circuit that monitors and determines a transition.

Deploying DVS

We use discrete VF levels, similar to those of Intel's XScale technology, on a frequency range from 600 MHz to 400 MHz and a voltage range from 1.3 V to 1.1 V. These ranges comply with Intel's IXP2400 configurations. We set the frequency step at 50 MHz and compute the voltage as in XScale. On the basis of previous study and current circuit technology, we set the VF transition latency between two adjacent levels to 10 μs. We convert this delay to stall cycles and insert them into the simulator.

The hardware required to implement the DVS policy is trivial. First we need a timer that signals after a certain number of cycles. A register suffices. To track ME idle time, we use an accumulator that counts the number of ME idle cycles. When the timer signals a DVS period, the accumulator compares its result with a preinitialized value, T, to determine whether there are enough idle cycles in the past period. T is simply the threshold that we set; for example, an idle time of 10 percent translates to 2,000 cycles for T in a period of 20,000 cycles. Figure 10 shows a schematic diagram of our DVS policy control mechanism.

We start the voltage and frequency at 1.3 V and 600 MHz. We set the SRAM and SDRAM frequencies at 200 MHz and 150 MHz, respectively. Figure 11 shows the results with the DVS period set at 15,000, 20,000, and 30,000 cycles and with 10 percent idle time as the threshold. We chose an interval that is long enough to accommodate the VF transition latency. Figure 11 shows that DVS can save up to 17 percent of power consumption with a performance loss of less than 6 percent. On average, we achieve 8.1 percent, 7.5 percent, and 7.0 percent power savings with only 0.5 percent degradation in throughput for intervals of 15,000, 20,000, and 30,000 cycles, respectively. DVS hardly affects throughput because the MEs have enough idle cycles to cover the stall penalties. We also tested using thresholds other than 10 percent and achieved similar results.

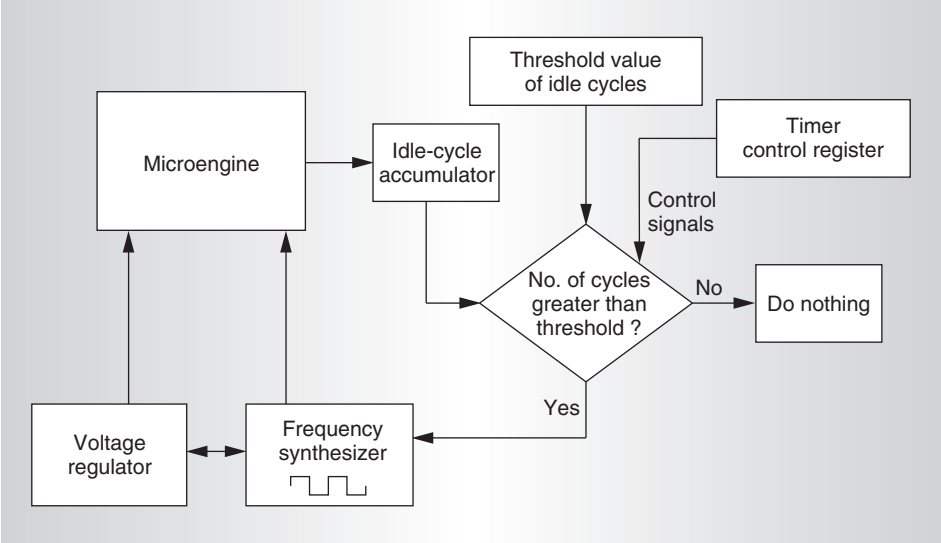

Figure 10. Dynamic voltage scaling (DVS) policy control mechanism.

Future work

In addition to NP performance and power consumption, it is often important to know the power dissipation in external modules, such as memories. We are adding power estimation for SRAM and SDRAM modules and hope to complete this part of the project soon. Then we will release NePSim with the memory module power calculation. Current NePSim source code is available for download at http://www.cs.ucr.edu/~yluo/nepsim/.

We believe there are many other techniques for reducing power consumption. For example, shutting down MEs during their low-activity periods could suppress dynamic power aggressively. However, turning off MEs might involve a program change, because the compiler currently premaps each ME to a subset of I/O ports. If the control store is sufficiently large, different versions of a program can be preloaded and selected to execute on the fly. Otherwise, the MEs might request a program change from the StrongARM core, which could increase delay and power consumption. We are planning to study this.

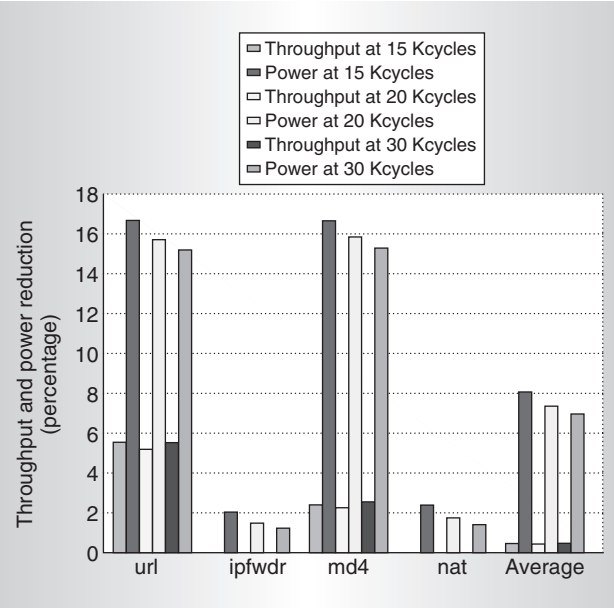

Figure 11. Normalized power and performance results when employing dynamic voltage scaling for the IXP1200 processor. Power savings is substantial, while performance degradation is insignificant.

It would also be interesting to model the StrongARM core and let it work together with the six MEs. In addition, the current version of NePSim is a model for the IXP1200. We will extend it to incorporate more-advanced products in the IXP family, such as the IXP2400 and 2800, thereby facilitating research on a broader range of NPs. MICRO

Acknowledgments

This research was supported by Intel's IXA University Program and National Science Foundation Grant No. CCR0220096.

References

1. Intel, "IXP1200 Network Processor Family Hardware Reference Manual," http://www.intel.com/design/network/manuals/27830309.pdf.
2. M. Franklin and T. Wolf, "Power Considerations in Network Processor Design," http://www.ecs.umass.edu/ece/wolf/papers/npw2003.pdf.
3. "Intel IXP1200 Network Processor Family: Development Tools User's Guide," http://www.intel.com/design/network/manuals/IXP1200_devtool.htm.
4. X. Chen, "Utilizing Formal Assertions for System Design of Network Processors," *Proc. Design, Automation and Test in Europe Conf. and Exhibition Designers' Forum* (DATE 04), IEEE CS Press, 2004.
5. M. Huang et al., "L1 Data Cache Decomposition for Energy Efficiency," *Proc. Int'l Symp. Low Power Electronics and Design* (ISLPED 01), ACM Press, 2001, pp. 10-15.
6. 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), IEEE CS Press, 2000, pp. 83-94.
7. H.-S. Wang et al., "Orion: A Power-Performance Simulator for Interconnection Networks," *Proc. 35th Int'l Symp. Microarchitecture* (Micro 35), IEEE CS Press, 2002, pp. 294-305.
8. Y. Luo et al., *NePSim: A Network Processor Simulator with Power Evaluation Framework,* tech. report, Computer Science and Eng. Dept., University of California, Riverside, 2003, http://www.cs.ucr.edu/index.php/research/publications.
9. CACTI, HP-Compaq Western Research Lab, http://research.compaq.com/wrl/people/jouppi/CACTI.html.

Yan Luo is a PhD candidate in the Computer Science and Engineering Department at the University of California, Riverside. His research interests include network processors, low-power microprocessor architecture, Internet routers, and parallel and distributed processing. He has an ME from Huazhong University of Science and Technology, China, and is a student member of the IEEE.

Jun Yang is an assistant professor in the Computer Science and Engineering Department at the University of California, Riverside. Her research interests include network processors, low-power microprocessor architecture, and secure microprocessor designs. Yang has a PhD in computer science from the University of Arizona. She is a member of the IEEE and the ACM.

Laxmi N. Bhuyan is a professor of computer science and engineering at the University of California, Riverside. His research interests include network processor architecture, Internet routers, and parallel and distributed processing. Bhuyan has a PhD from Wayne State University. He is a Fellow of the IEEE, the ACM, and the American Association for the Advancement of Science.

Li Zhao is a PhD candidate in the Computer Science and Engineering Department at the University of California, Riverside. Her research interests include computer architecture, networking software, and performance evaluation. She has a BS in scientific information from Xidian University, China, and an MS in physics from the Institute of High Energy Physics, Chinese Academy of Science. Zhao is a student member of IEEE.

Direct questions and comments about this article to Yan Luo, Dept. of Computer Science and Engineering, University of California, Riverside, CA 92521; yluo@cs.ucr.edu.