Title: # NePSim: A Network Processor Simulator with Power Evaluation Framework

Corresponding Author: Yan Luo
Address: Department of Computer Science & Engineering
University of California
Riverside, CA 92521
Phone: (909)-787-2001
Fax: (909)-787-4643
Email: yluo@cs.ucr.edu

Author: Jun Yang
Address: Department of Computer Science & Engineering
University of California
Riverside, CA 92521
Phone: (909)-787-2558
Fax: (909)-787-4643
Email: junyang@cs.ucr.edu

Author: Laxmi Narayan Bhuyan
Address: Department of Computer Science & Engineering
University of California
Riverside, CA 92521
Phone: (909)-787-2347
Fax: (909)-787-4643
Email: bhuyan@cs.ucr.edu

Author: Li Zhao
Address: Department of Computer Science & Engineering
University of California
Riverside, CA 92521
Phone: (909)-787-2001
Fax: (909)-787-4643
Email: zhao@cs.ucr.edu

Abstract : There is an increasing interest in the NP architecture design as well as its power dissipation characteristics. Up to now, there has not been an open-source simulation infrastructure that makes the performance/power tradeoffs in NPs clearly visible to computer architects.

This paper presents NePSim, an integrated infrastructure for analyzing and optimizing NP design and power dissipation at architecture-level. NePSim contains a cycle-accurate simulator for a typical NP architecture, an automatic verification framework for testing and validation, and a power estimation model for measuring the power consumption of the simulated NP. NePSim achieves satisfactory accuracy in both performance and power modeling, and will soon be released as a publicly available NP research infrastructure. In addition, we present in this paper a low-power technique that exploits unique features of typical NPs. We experimented these techniques in NePSim and obtained up to 17% power savings with little impact on the overall packet throughput.

Keywords : Network Processor, Simulation, Power Modeling, Performance Evaluation, Low-power

NePSim: A Network Processor Simulator with Power Evaluation Framework *

Yan Luo, Jun Yang, Laxmi N. Bhuyan and Li Zhao

June 29, 2004

1 Introduction

Network processors (NPs) have emerged recently as a successful platform to provide both high performance and flexibility in building powerful routers. A typical NP consists of multiple fast execution cores on a single chip that exploits packet level parallelism. Example products include Intel's IXP family [9], IBM's PowerNP, Motorola's C-Port and Agere's APP550. With the exponential increase in clock frequency and the core complexity, power dissipation will become one of the major design considerations in NP development. For example, in a typical router configuration, there may be one or two NPs per line card. A group of line cards, e.g. 16 or 32, are generally placed within a single rack or cabinet [6]. Thus, the aggregated heat dissipation becomes a big concern, given that each NP typically consumes ~20 Watts and the operating temperature can reach as high as 70°C.

Research in the network processors and their power dissipation is in its infancy. The main obstacle lies in the lack of open-source infrastructure that analyzes and quantifies the performance and power ramifications among NP architecture alternatives. Although there have been cycle-accurate architecture simulators for commercial network processors, e.g. Intel's SDK [10] and Motorola's C-Ware, they are not open-source for internal architecture design by the users. Moreover, these simulators do not incorporate power modeling and evaluation tools.

In this paper, we present NePSim, an integrated system that includes a cycle-accurate architecture simulator, an automatic formal verification engine, and a parameterizable power estimator of a network processor that consists of clusters of multi-threaded execution cores, memory controllers, input/output ports, packet buffers and high speed buses. To perform concrete simulation and provide reliable performance-power analysis, we defined our system to comply with the Intel's IXP1200 specification since it is widely adopted in academia as a representative model for network processor research. The verification engine, called IVERI [3], can validate the simulation from the execution log traces using user-defined constraints in a verification language. The power estimator combines a suite of models (XCacti [5], Wattch [1], and Orion [8]) for dynamic power measurements and calculates results based on per-cycle resource usage counts.

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

2 Cycle-Level Simulation

We start with the high level overview of IXP1200 followed by the description of our simulator software structure.

*This research was supported by Intel IXA University Program and NSF grant No. CCR0220096.

2.1 Background — Intel IXP1200 and its microengines

The IXP1200 is an integrated processor that is comprised of a single StrongARM processor, six microengines (MEs), standard memory interfaces, and high-speed bus interfaces. The StrongARM core is used for management functions such as initializing MEs, running routing protocols, exception handling, and other tasks. The MEs are fast RISC engines that can be programmed to perform any high-speed packet inspection, data manipulation, and data transfer. The SDRAM Unit is a shared memory interface that can be accessed by the StrongARM, the MEs, and devices on the PCI bus. SDRAM is mostly used for temporarily storing packet payloads. The SRAM Unit is also a shared memory interface accessed by StrongARM and MEs. SRAM is mostly used for storing control data structures such as forwarding or routing tables. The IX Bus Unit (FBI in short) is controlled by MEs, and transfers data blocks between IXP1200 and networking devices such as MACs. Readers can refer to [9] to obtain the detailed description of IXP1200.

2.2 The simulator

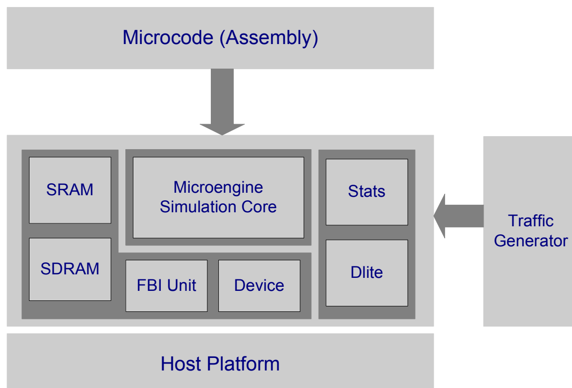

Figure 1: NePSim software structure.

Our simulator implements most of the functionalities of the IXP1200, the external SDRAM, and SRAM. We also implemented full-fledged command bus arbiter and FBI unit, which contains the scratchpad memory and packet input/output buffers. We did not model the StrongARM core since it is mainly used for control plane functions that do not affect the critical path of packet processing.

Figure 1 depicts the NePSim hardware model's software architecture. The applications are first compiled into microcode assembly. The reason we did not choose to use the binary directly is to leave room for instruction extensions in future research. Usually, it is easier to modify the program using assembly than binary. The inputs to NePSim are network packet streams generated from the "traffic generator". The ideal packet streams are those collected from real-world, or downloaded from trace archive sites such as NLANR. If those were not available for an application, a synthetic trace of packets will be generated by this module.

The body of NePSim is the module that simulates the following 5 stages of the ME pipeline: p0(lookup of instruction); p1(initial instruction decode and formation of the source register address); p2(read of operands from source registers); p3(perform ALU, shift or compare operations, and generate the condition codes); p4(write result to destination register). The threads on a ME share the pipeline and functional units such as ALU. The execution of threads is not preemptive, which means that a thread cannot get hold of the pipeline unless the running thread yields control. There are a couple of mechanisms of mutual exclusion: absolute register, atomic scratchpad operations, etc [9].

Modeling the memory module and memory controllers, is a bit involved due to the lack of timing specification from IXP1200. The memory controllers of SRAM and SDRAM 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 command enqueue and service time is not stated explicitly in the documents available to us. Our methodology here is to analyze memory access traces from Intel's SDK simulator which contains detailed cycle information of all possible events such as issues and commits of memory accesses. From the traces, we can extract relatively accurate timing relations among different events . Based on those relations, we implemented the memory units to match NePSim with the IXP1200 as much as possible.

The "Device" module implements the I/O devices such as input, output ports, and MAC. The "DLite" module is similar to the debugger in SimpleScalar. For example, users can set break points, print pipeline status, display register values, and dump memory content etc.

The simulator is highly parameterized so that the user can model components with different configurations. For example, the MEs can take different clock rates and supply voltages. The SRAM and SDRAM can be configured with different latencies and bandwidth; The incoming traffic can be defined with different arrival rates and arrival patterns, e.g. Poisson, Uniform, Random. Moreover, the simulator also provides the following advantages over Intel's SDK.

1. *Enables designing new architecture.* This is probably the most eminent motivation behind developing NePSim. In SDK, if a user wants to change the behavior of certain hardware just to see what difference it makes, (s)he might need to get around by modifying the program that is being run. This is certainly intrusive and non-trustable. With NePSim, any modification in the architecture can be tested at ease, providing the users with maximum freedom.

2. *Models varying number of MEs and varying number of threads per ME.* We can take advantage of the MicroC compiler being able to create multiple ME executables (even > 6) and feed them into NePSim. The Intel's SDK for IXP1200 is, however, unable to run more than 6 MEs. This feature allows us to explore architectures with higher number MEs in future NPs such as IXP2400, or 2800.

3. *Provides instruction set extensibility in microcode assembly code.* Users can tag instructions with certain flags when an ISA optimization is experimented. Also, users can create new instructions and insert them into the assembly code without the care of modifying PC-relative or offset fields in other instructions. This functionality can not be achieved through SDK.

4. *Provides faster execution speed.* Similar to SDK, the NePSim also has a comprehensive set of statistical variables. Unlike SDK, we can collect data selectively increasing the simulation speed. On average, we can simulate 120K instructions per second (on a 900MHz PIII machine), which is about 10 times faster than Intel's SDK.

3 Validation

Validating NePSim simulator against IXP1200 architecture is crucial because the performance and power simulation is useful only if it has satisfactory accuracy. It is relatively simple to verify the functional properties of NePSim by comparing the program results. However, we have to spend significant effort on the performance verification in order to make this process accurate, efficient and formal.

We built a verification backend tool called IVERI, with which the simulator properties are formally specified in a verification language. We first run same benchmarks on both NePSim and Intel SDK and get two simulation traces. The trace from SDK is treated as the standard reference trace. The log traces contain events annotated with information in a pre-defined 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 simulator until a bug is revealed. Thus, through IVERI, we can formally and accurately describe the performance properties or requirements of a new design. Detailed specification of IVERI can be found in [3].

Next, we present the verification results of overall NePSim throughput and average processing time against IXP1200 [1]. We ran four benchmarks that we ported to NePSim and measured the performances in terms of throughput (Mbps, megabits per second) and average packet processing time (cycles). The benchmark descriptions will be given in Section 5.1. The comparison is shown in Figure 2 and 3. We observed an average error of 1% in throughput and 6% in average processing time across the four benchmarks. The error

[1]The verification results of individual major components such as SRAM and SDRAM are presented in an unbridged verion of this paper.

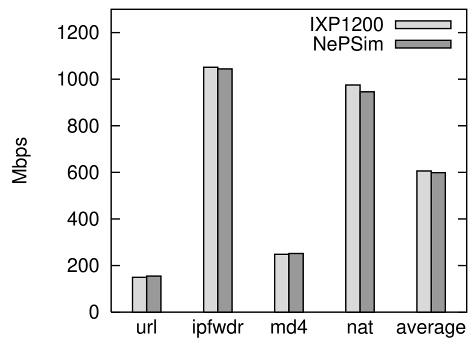

Figure 2: Throughput comparison between NePSim and IXP1200.

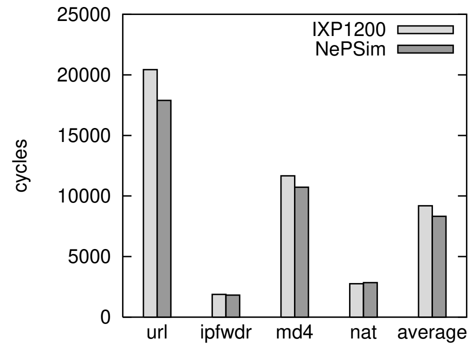

Figure 3: Average cycles comparison between NePSim and IXP1200.

of average processing time is mainly due to the inaccuracy of our SDRAM model. Such timing error affects the packet processing order and time. Overall speaking, we think the simulation can produce relatively dependable results.

4 Power Modeling

We extracted and combined various power models from XCacti [5] , Wattch [1] , and Orion [8] for different hardware structures in NePSim. XCacti is an enhanced version over Cacti2.0 by adding cache writes, misses, and write backs power modeling. We first classify the hardware component structures for IXP1200 into five different categories: ALU and shifter, registers, cache-like structures, queues, and arbiters. The registers include the general purpose register files, the transfer register files, and the control register files per ME. The cache-like structures include the control store per ME and the scratchpad memory in FBI. The queue type structures include the command FIFO per ME, receive and transmit FIFOs in FBI, and groups of queues in SDRAM and SRAM units. The arbiter type includes the context event arbiter per ME, command bus arbiter and reference arbiters in the SDRAM and SRAM units. The IXP1200 uses 0.28μm technology. However, none of the three tools supplies necessary scaling factors[2] for 0.28μm. Thus, we use 0.25μm technology, the closest feature to 0.28μm among the provided ones. As one can expect, the overall power consumption we can obtain should be smaller than the reported power data for IXP1200. Table 1 gives a detailed list of the components we modeled, the tools we adopted, and their corresponding configurations.

Hardware component	Model type	Tool	Configurations
GPRs per ME	array	XCacti	2 64-entry files, 1 read/write port per file
transfer reg. per ME	array	XCacti	4 32-entry files, 1 read/write port per file
control reg. per ME	array	XCacti	1 32-entry file, 1 read/write port
control store per ME and scratchpad memory	cache without tag path	XCacti	4K-byte, 4 bytes per block, direct mapped, 10-bit address
ALU and shifter	ALU and shifter	Wattch (cc1 clock gating)	32 bits
command FIFO per ME RFIFO, TFIFO high priority queues order queues odd/even bank queues readonly queues	array	Wattch	2-entry, 64-bit width 8-entry (SDRAM & SRAM) 24/16-entry (SDRAM/SRAM) 16-entry (SDRAM) 16-entry (SRAM)
command bus arbiter context arbiter per ME reference arbiters	matrix and rr. arbiter rr. arbiter matrix	Orion	3 priority req, 6 rr. req. at max 4 rr. req. up to 4 priority req.

Table 1: Component modeling information. “GPRs” stands for general purpose registers. “reg.” stands for register file. “rr.” stands for round-robin.

[2] the scaling factors are used to scale wire capacitance, resistance, transistor length, area, voltage, threshold voltage, and sense voltage [2, 1, 8]

Validate power results against IXP1200 datasheet. The power data available to us is only the total power consumption of the entire IXP1200 chip. To the best of our knowledge, the power breakdown of each component has never been reported. Thus, we will base on the total power to validate our modeling. According to IXP1200 datasheet, the typical power of the IXP core excluding I/O is 4.5W at 232MHz with 2V voltage supply and 0.28μm technology. This number includes all the MEs, memory units, FBI which contains the control store, and the StrongARM core. The StrongARM core consumes on average 0.5W power. This leaves ~4W for the rest of the chip. From our power models, each ME consumes 0.468W power, the FBI (containing the scratchpad memory) consumes 0.363W, the SRAM unit consumes 0.0639W, the SDRAM unit consumes 0.0643W. These values are average power consumptions. Assuming every component switches on every cycle, the total power is $0.468 \times 6 + 0.363 + 0.0639 + 0.0643 = 3.3W$. The $4.0 - 3.3 = 0.7W$ difference is due to the fact that we are using a smaller technology, 0.25μm instead of 0.28μm, and there are other structures that we did not model, e.g., internal buses, clock. These will be included in our upgraded version of NePSim.

5 Power-Performance Analysis

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

5.1 Benchmark descriptions

We compiled and ported four representative NP applications to conduct the experiments: IP forwarding (*ipfwdr*), URL routing (*url*), 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 two MEs transmit processed packets through output interfaces.

The *ipfwdr* is a sample IP forwarding software provided in Intel's SDK [10]. This application implements an IPv4 router which forwards IP packets between networks. The processing includes Ethernet and IP header validation and trie-based routing table lookup. The routing table is stored in the SRAM and the output port information is in SDRAM. The rest MEs transmit the processed packets to the next-hop router based on the output port information.

The *url* routing program routes packets based on their contained URL request. It is a primary task in content-aware switches that often examine the payload of packets when processing them. The *url* then performs a string matching algorithm which we ported from NetBench. The string patterns are initialized in SRAM, so SRAM accesses need to be generated in later comparisons. Also, large number of requests are generated to SDRAM, where payload data is stored since they need to be scanned for pattern matching.

The *nat* is a network address translation program. It uses the source and destination IP addresses and port numbers to compute an index. This index is used for a hash table lookup to retrieve a replacement address and port. Thus, each packet will access the SRAM for looking up the hash table. No SDRAM accesses are necessary.

The *md4* algorithm works on arbitrary length of messages and provides a 128-bit "fingerprint" or "digital signature". This algorithm is used for SSL or firewall that can be implemented at the edge routers. The algorithm moves data packets from SDRAM to SRAM, and access SRAM multiple times to compute the digital signature. This program is both memory and computation intensive.

The distribution among the number of MEs for receiving and transmitting data has been studied previously. For NPs like IXP1200 where a total of six MEs are present, the usual configuration is to use four MEs (16 threads) for receiving and two MEs (8 threads) for transmitting. This 4:2 ratio was tested to provide maximum throughput in general. We also adopt this configuration throughout our experiments.

5.2 Performance observations

In the first set of experiments, we study the impact of having more number of MEs on the total packet throughput measured in Mbps (megabits per second). Intuitively, the throughput of a benchmark should increase with the number of MEs or threads because higher packet level parallelism is being explored. However, for memory intensive benchmarks (url and md4), increasing the number of threads means increasing the memory contention since the memories are shared among all threads. The throughput tends to level off after certain point as seen in both Figure 4 and 5. Figure 5 also shows that doubling the core frequency does not bring doubled throughput for memory-intensive applications.

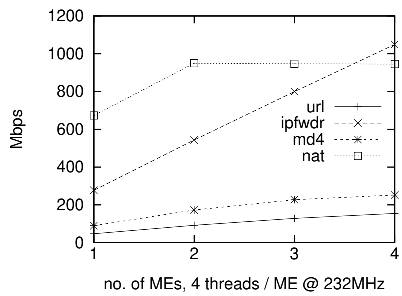

Figure 4: Impact of number of cores on throughput at 232MHz.

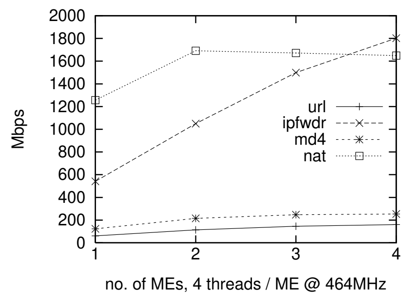

Figure 5: Impact of number of cores on throughput at 464MHz.

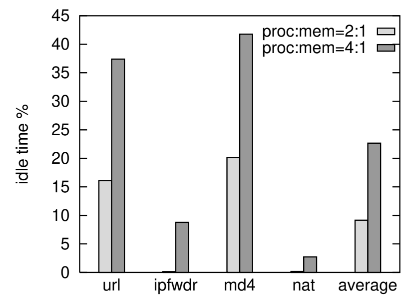

Figure 6: Average core idle time with different speed ratio of the core and the memory.

A strange result from these two graphs is the decreasing Mbps for *nat* with increase in the number of MEs. It turned out that, unlike other programs, this program is not memory bound. To see this, we measured the average ME idle time corresponding to different ME/memory speed ratio, i.e., 2:1 vs. 4:1 as used in Figure 4 and 5 respectively. When a program is memory bound, it often happens that all the threads in a ME are swapped out of pipeline waiting for their memory references, resulting an idle ME. For the same program, the slower the memory, the longer the ME idle time.

As we can see from Figure 6, *nat* does not have much ME idle time even when the ME/memory speed ratio is 4:1. This implies that all the MEs are busy but they are not producing much useful work since the throughput goes down. We traced its execution and found out that the bottleneck lies in the transmitting threads. The receiving threads process the packets fast enough that they can not allocate new memory slots for incoming packets. The memory slots should be released by the transmitting threads once a packet is sent out. At this time, all the receiving threads are busy requesting for new memory slots but few can get them. This phenomenon suggests that a 3:3 receiving/transmitting ME ratio may work better for *nat* than the traditional 4:2 configuration.

5.3 Where does the power go?

In the second set of experiments, we are interested in the power consumption characteristics of the IXP1200 and the ME. In particular, we want to identify those components where the bulk of power is spent. Figure 7 plots the power distribution of IXP1200 among MEs, whereas Figure 8 plots the power consumed by individual components inside an ME.

In Figure 7 ME0 to ME3 are working as receiving MEs, and ME4 and ME5 are the transmitting MEs. Almost all receiving MEs consume the same power. The transmitting MEs consume about 5% more than the receiving MEs. It is interesting to see that the ALU takes up the most power (45% on average), followed by the control store (28%) where the program is stored. This is because the control store is accessed on almost every cycle. The third power hungry component is the general purpose register (GPR) files (13%) where instruction operands and results are stored. Even when data are loaded from the memory the transfer register files, they are moved again to GPRs for ALU operations. The control memory size and the number of GPRs were assumed to be 1024 words and 128 respectively, same on the IXP1200. If these numbers are varied, the power consumption will vary accordingly. The fourth component is the static power (∼7%

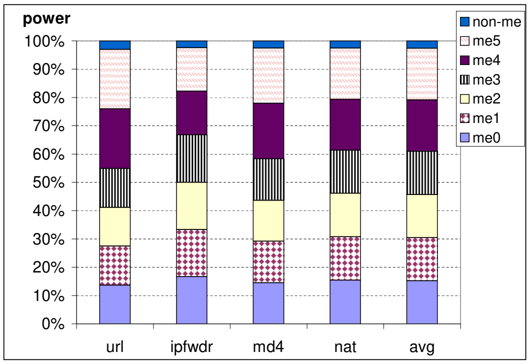

Figure 7: Simulated IXP1200 power consumption breakdown.

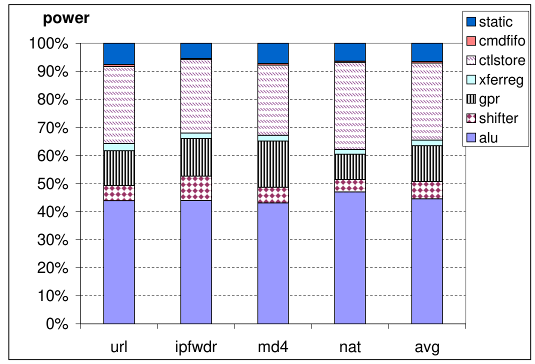

Figure 8: Microengine power consumption breakdown.

on average). We included static power estimation in the figure considering the significant ME idle time. For 0.25μm technology, the percent of the static power with respect to the dynamic power is less than 5% (we chose 2% in this experiment). The result in the figure shows a $\sim$7% of static power budget. This indicates that even though the dynamic power of MEs dominates, future NPs might adopt smaller and smaller technology feature size and higher and higher clock frequency, the static power may become an important factor considering the potential ME idle time.

5.4 Performance-power observations

Next we want to find out if the performance variation is consistent with the power variation. Particularly, with the trend in adding more processing capabilities on chip, i.e., adding more cores, how do the performance and power respond? To see this, we measured the normalized increase in performance and power with growing number of MEs. The core and memory frequency ratio is set to 4:1. The results are shown in Figure 9.

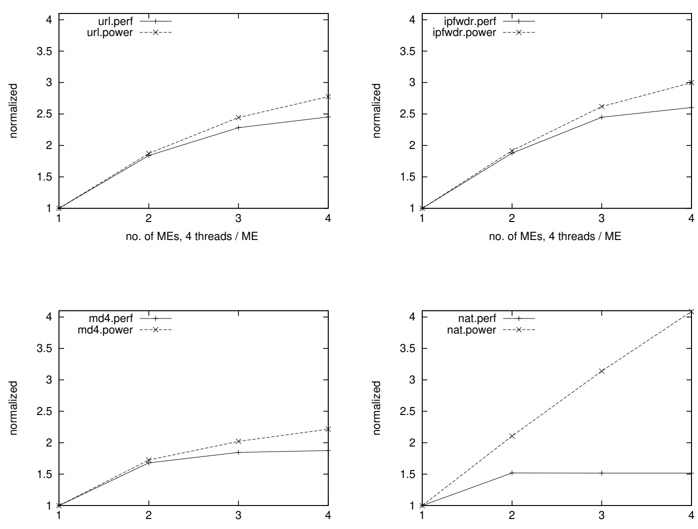

Figure 9: Analysis of power efficiency.

From the graph, it is seen that both performance and power consumption grow with the increasing number of MEs except for the performance of *nat* (explained before). Our first observation is that the power consumption of MEs does not multiply with the increase in number of MEs, e.g. 2 MEs spend less than

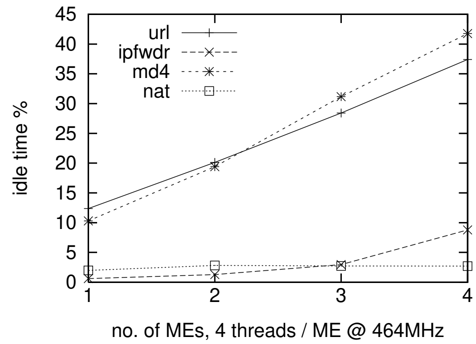

Figure 10: Average core idle time with different number of active threads.

twice the power of 1 ME. This is because the ME idle time increases when more MEs are added since more threads are competing for the same memories. We measured the average ME idle time varying the number of MEs and plot the results in Figure 10.

The most important observation from Figure 9 is the trend in the performance-power curve pair for each application. Every such pair in the graph presents the same property that the gap between the two curves widens as the number of ME increases. Put in another way, the power consumption increases *faster* than performance. Similar conclusions were also made by Franklin and Wolf using an analytical model [6]. Thus, future NPs will face power consumption as a higher constraint than the processing capability. For this reason, we next present a technique that uses the classic dynamic voltage scaling (DVS) to conserve the power of the MEs.

6 Using Dynamic Voltage Scaling to Reduce Processing Core Power

Dynamic voltage scaling has been employed widely for microprocessors, revealing significant power savings. DVS exploits the microprocessor utilization variance, reducing voltage and frequency when the processor has low activity and increasing them when the peak processor performance is required. Through the study in the previous sections, we have identified significant low-activity percentage, i.e., the idle time, of the MEs. Thus, there is plenty of room for deploying DVS to conserve power.

6.1 DVS policy

Figure 10 shows that the average ME idle time is abundant since most of the benchmarks are memory bound. During the ME idle time, all threads are put to “wait” state and the MEs are running with the lowest activity. Applying DVS while MEs are not very active can reduce the total power consumption substantially. Our scheme observes the ME idle time periodically. Once the percentage of the idle time in the past period is over a threshold, we scale down the voltage and frequency (VF in short) by one step unless the minimum allowable VF is hit. If the percentage is below threshold, indicating that the ME in the past period is relatively busy, we scale up the VF by one step unless they are at maximum allowable values.

There are four issues that we need to consider here: (1) transition step meaning whether or not to use continuous or discrete changes in VF. (2)transition status indicating if we allow the ME to continue working during VF regulation, (3)transition time between two different VF states, and (4)transition logic complexity which is the overhead of the control circuit that monitors and determines a transition,

We use discrete VF levels similar to that of the Intel XScale on a frequency range from 600MHz to 400MHz, and voltage range from 1.3V to 1.1V. These ranges comply with Intel's IXP2400 configurations. We set the frequency step as 50MHz and voltage is computed as in XScale. From previous study and the current circuit technology, we set the latency of VF transition between two adjacent levels to 10μs. This delay is converted to stall cycles, and inserted into the simulator.

The hardware required to implement the DVS policy is very trivial. We need a timer that signals over every certain number of cycles—a register suffices. To keep track of ME idle time, we use an accumulator that counts the ME idle cycle. Once the timer signals a DVS period, the accumulator compares its result with a pre-initialized value T to determine whether there are enough idle cycles in the past period. T is simply the threshold that we set, e.g. a 10% of idle time translates to 2000 cycles for T on a period of 20000 cycles. Figure 11 presents the schematic diagram of our DVS policy control mechanism.

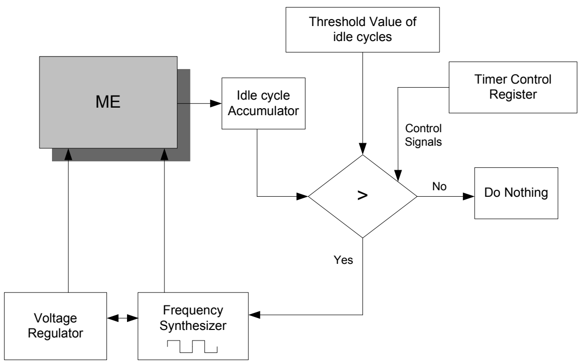

Figure 11: DVS policy control mechanism.

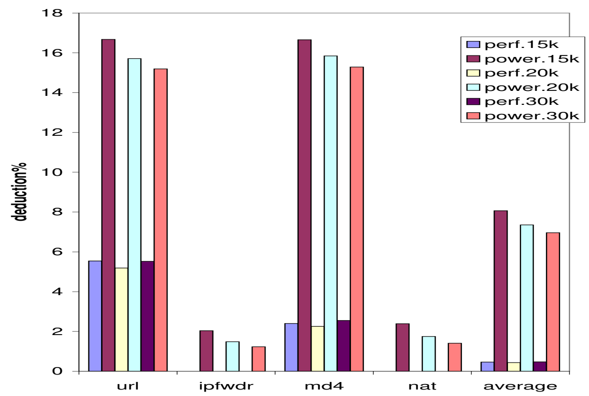

Figure 12: Normalized power and performance results employing DVS for IXP1200.

We start the voltage and frequency at 1.3V and 600MHz initially. The SRAM and SDRAM frequencies are set at 200MHz and 150MHz. Figure 12 plots the results when the DVS period is set at 20000 cycles and 30000 cycles with 10% idle time as the threshold. The interval time is chosen long enough to accommodate the VF transition latency. Figure 12 shows that DVS can save up to 17% of power consumption with less than 6% performance loss. On average, we achieve 8.1%, 7.5% and 7% of power savings with only 0.5%, 0.5% and 0.5% degradation in throughput for intervals of 15000, 20000 and 30000 cycles respectively. The throughput is hardly affected due to DVS because the MEs have enough idle cycles to cover the stall penalties. We also tested using different threshold than 10% and got very similar results.

7 Future Work

Our power model in NePSim currently estimates the performance and power for the NP(IXP1200). Often, it is important to know the power dissipation in the external module such as the memories. We are now adding the power estimation for SRAM and SDRAM modules and hoping to complete this part of the project within short amount of time. We will release the NePSim after that with the memory module power calculation.

We believe there are plenty of other techniques that can be employed in reducing the power. For example, one might shut down MEs during the low activity time suppressing dynamic power aggressively. However, turning off MEs may involve change of program since each ME is currently pre-mapped to a subset of I/O ports by the compiler. If the control store is sufficiently large, different versions of program can be pre-loaded and selected to execute on the fly. Otherwise, the MEs may request for change of program from the StrongARM which could potentially increase delay and power. We plan to study this in the near future.

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 IXP1200. We will extend it to incorporate more advanced products in the IXP family such as IXP2400 and 2800, to facilitate future research in a broader range of NPs.

8 Existing Work on Network Processors

Franklin and Wolf developed an analytic performance model that captures a generic network processor's processing performance and power consumption. The modeled prototypical NP contains a number of identical multithreaded general-purpose processors clustered together to share a memory interface. Power was measured through Wattch [1] and Cacti [2]. Their extended research explored the design space of NPs and showed performance-power impact of different systems [6]. The advantage of our NePSim over the above model is that we target a real NP architecture (IXP1200) and perform accurate cycle-level simulation and power modeling. We also made effort in porting network applications so that they can execute in NePSim as well as in IXP1200. Thus, the statistics we collected are more realistic and dependable.

9 Conclusion

We developed a cycle-accurate simulation framework NePSim for a typical network processor such as IXP1200. NePSim includes a full-fledged simulator, an automatic verification engine, and a parameterizable power estimator. We tested NePSim through four network applications, achieving satisfactory accuracy and execution speed. In addition, we used NePSim for an implementation of DVS on MEs resulting power saving up to 16%. Little performance degradation was observed. We believe that our simulator will be used by many researchers to start a wave in modern NP architecture research. The source code of NePSim can be downloaded from *http://www.cs.ucr.edu/~yluo/nepsim/*.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," *The 27th International Symposium on Computer Architecture (ISCA)*, pp. 83–94, 2000.

[2] CACTI, HP-Compaq Western Research Lab, *http://research.compaq.com/wrl/people/jouppi/CACTI.html*.

[3] X. Chen, Y. Luo, H. Hsieh, L. Bhuyan, and F. Balarin, "Utilizing Formal Assertions for System Design of Network processors," *Design Forum, Design, Automation and Test in Europe (DATE04)*, 2004.

[4] J. Hasan, S. Chandra, and T. N. Vijaykumar, "Efficient Use of Memory Bandwidth to Improve Network Processor Throughput," *The 30th International Symposium on Computer Architecture (ISCA)*, pp. 288–299, 2003.

[5] M. Huang, J. Renau, S. M. Yoo, J. Torrellas, "L1 Data Cache Decomposition for Energy Efficiency," *ISLPED'01*, pp. 10–15, 2001.

[6] M. Franklin and Tilman Wolf, "Power Considerations in Network Processor Design," *Workshop on Network Processors – NP2, in conjunction with HPCA9*, pp. 10–22, 2003.

[7] T. Sherwood, G. Varghese, and B. Calder, "A pipelined memory architecture for high throughput network processors," *The 30th Annual International Symposium on Computer Architecture (ISCA)*, pp. 288–299, 2003.

[8] H.-S. Wang, X.-P. Zhu, L.-S. Peh and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks", *The 35th International Symposium on Microarchitecture*, pp. 294–305, 2002.

[9] Intel Corporation, "IXP1200 Network Processor Family Hardware Reference Manual," http://developer.intel.com/design/netwrok/ixa.html, 2001.

[10] "Developer Workbench/Transactor," http://developer.intel.com/design/network/products/ npfamily/sdk2.htm.