International Journal of Software & Hardware Research in Engineering
ISSN No: 2347-4890 Volume 2 Issue 4, April 2014

# Intelligent DVFS architecture for a Multi-Core Systems

[1]Mr. K.Kohila Padmanaban, [2]Mrs. J.Mangaiyarkarasi
[1]P.G.Scholar, Anna University ,Regional centre, Madurai.
[2]Faculty, Dept of ECE, Anna University, Regional centre Madurai.

***Abstract*: Dynamic voltage and frequency scaling (DVFS), a widely adopted technique to ensure safe thermal characteristics while delivering superior energy efficiency, is rapidly becoming inefficient with technology scaling due to two critical factors: 1) inability to scale the supply voltage due to reliability concerns and 2) dynamic adaptations through DVFS cannot alter underlying power hungry circuit characteristics, designed for the nominal frequency. In this paper, we proposed an architecturally homogeneous DVFS technique which performed DVS and DFS identically, a fundamentally alternate means to design energy efficient multicore systems. Multicore chip designs throughout the last decade have witnessed the trend of increasing number of on-chip cores. For example, IBM POWER7 and Sun ROCK processors have eight and 16 on-chip cores, respectively. Utilizing several on-chip cores, these systems can run many applications simultaneously, creating a wide diversity in power-performance requirements. In such an environment, meeting the steep energy efficiency demands requires a flexible multicore platform with the ability to exercise fine grain control. On the other hand, interaction of multiple on-chip cores plays havoc in maintaining safe thermal conditions. Dynamic Thermal Management (DTM), a broad class of techniques that aim to deliver energy efficiency within safe thermal limits, has now become ubiquitous. DVFS can be split into dynamic voltage scaling and dynamic frequency scaling.**

## I. INTRODUCTION

Across the entire spectrum of microprocessor design, dynamic voltage and frequency scaling (DVFS) has been the basic foundation for DTM. Driven by an increasingly diverse application pool, the runtime utilization rate of on-chip cores often falls substantially below the nominal frequency. DVFS is beneficial during these phases, as it saves energy consumption with minimal loss in performance. The effectiveness of DVFS, however, is rapidly degrading with technology scaling. This growing DVFS inefficiency stems from two critical factors: 1) dynamic adaptations fail to alter the intrinsic circuit characteristics (such as gate sizes and threshold voltages), which are designed for the nominal frequency, and tend to be power hungry and 2) forthcoming technology generations are restricting the supply voltage scaling margins which is the key component behind power savings through DVFS. Hence, it is now critical to find alternative scalable means of achieving energy efficiency.

We demonstrate that dynamic adaptations using DVFS are markedly energy inefficient than techniques that design circuits ground up for lower performance. Fundamentally, lower performance requirements allow certain circuit components to choose appropriate gate sizes and device attributes, for lower power. This design style can yield significantly better energy efficiency than achieved by applying DVFS on circuits designed for higher performance. The key question is how to use this design style at the system level, and analyze its impact on real workload execution.

## II. DVFS Based multi-core systems

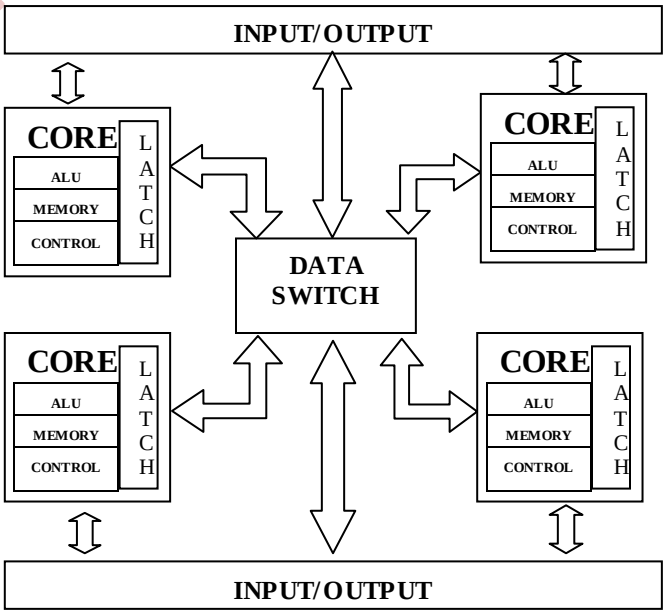


Figure.1 Block diagram for multicore system

### A. ALU

An arithmetic-logic unit (ALU) is the part of a computer processor (CPU) that carries out arithmetic and logic operations on the operands in computer instruction words. In some processors, the ALU is divided into two units, an arithmetic unit (AU) and a

logic unit (LU). Some processors contain more than one AU - for example, one for fixed-point operations and another for floating-point operations. (In personal computers floating point operations are sometimes done by a floating point unit on a separate chip called a numeric coprocessor).

Typically, the ALU has direct input and output access to the processor controller, main memory (random access memory or RAM in a personal computer), and input/output devices. Inputs and outputs flow along an electronic path that is called a bus. The input consists of an instruction word (sometimes called a machine instruction word) that contains an operation code (sometimes called an "op code"), one or more operands, and sometimes a format code.

The operation code tells the ALU what operation to perform and the operands are used in the operation. (For example, two operands might be added together or compared logically.) The format may be combined with the op code and tells, for example, whether this is a fixed-point or a floating-point instruction. The output consists of a result that is placed in a storage register and settings that indicate whether the operation was performed successfully. (If it isn't, some sort of status will be stored in a permanent place that is sometimes called the machine status word.)

In general, the ALU includes storage places for input operands, operands that are being added, the accumulated result (stored in an accumulator), and shifted results. The flow of bits and the operations performed on them in the subunits of the ALU is controlled by gated circuits. The gates in these circuits are controlled by a sequence logic unit that uses a particular algorithm or sequence for each operation code.

In the arithmetic unit, multiplication and division are done by a series of adding or subtracting and shifting operations. There are several ways to represent negative numbers. In the logic unit, one of 16 possible logic operations can be performed - such as comparing two operands and identifying where bits don't match. The design of the ALU is obviously a critical part of the processor and new approaches to speeding up instruction handling are continually being developed.

*B. Memory*

The term "memory", meaning primary memory is often associated with addressable semiconductor memory, i.e. integrated circuits consisting of silicon-based transistors, used for example as primary memory but also other purposes in computers and other digital electronic devices. There are two main types of semiconductor memory: volatile and non-volatile. Examples of non-volatile memory are flash memory (sometimes used as secondary, sometimes primary computer memory) and ROM/ PROM/ EPROM/ EEPROM memory (used for firmware such as boot programs). Examples of volatile memory are primary memory (typically dynamic RAM, DRAM), and fast CPU cache memory (typically static RAM, SRAM, which is fast but energy-consuming and offer lower memory capacity per area unit than DRAM) .

Most semiconductor memory is organized into memory cells or bistable flip-flops, each storing one bit (0 or 1). Flash memory organization includes both one bit per memory cell and multiple bits per cell (called MLC, Multiple Level Cell). The memory cells are grouped into words of fixed word length, for example 1, 2, 4, 8, 16, 32, 64 or 128 bit. Each word can be accessed by a binary address of *N* bit, making it possible to store 2 raised by *N* words in the memory. This implies that processor registers normally are not considered as memory, since they only store one word and do not include an addressing mechanism.

*C. Random Access Memory (RAM)*

RAM (random access memory) stores the operating system, application programs, and data in current use so that they can be quickly reached by the computer's processor.RAM is much faster to read from and write to than the other kinds of storage devices like hard disk, floppy disk, and CD-ROM available in a computer. However, the data in RAM stays there only as long as the computer is running. RAM is a volatile memory and when the computer is turned off, RAM loses its data. When the computer is turned on again, operating system and other files are once again loaded into RAM usually from the hard disk. If RAM becomes filled up, the processor needs to continually go to the hard disk to replace old data in RAM with new data which slows down the computer.

*E. Read Only Memory (ROM)*

ROM is an acronym for Read-Only Memory. It refers to computer memory chips containing permanent or semi-permanent data. Unlike RAM, ROM is non-volatile.

The contents of ROM remain even after the computer is switched off. A computer comes with a small amount of ROM containing the boot firmware. This consists of a few kilobytes of code that tell the computer what to do when it starts up, e.g., running hardware diagnostics and loading the operating system into RAM. On a PC, the boot firmware is called the BIOS. Originally, the ROM used to be read-only and physical replacing of ROM chips was required to update programs written on the ROM. The new versions of ROM allow limited rewriting making it possible to upgrade firmware such as the BIOS by using installation software.

*F. Control unit*

The control unit is a finite state machine that takes as its inputs the IR, the status register (which is partly filled by the status output from the ALU), and the current major state of the cycle. Its rules are encoded either in random logic, a Programmable Logic Array (PLA), or Read-Only Memory (ROM), and its outputs are sent across the processor to each point requiring coordination or direction for the control unit. For example the outputs needed for the portion of the instruction/data path shown in Discussion 13 are Jump/Branch/Next PC, IR Latch, Read Control, Load Control, ALU Function Select, Load/Reg-Reg, and Reg R/W.

The ALU function select takes the instruction op code and translates it into a given function of the ALU (either one line per ALU function or a compact binary code for the function). The Jump/Branch/PC depends on the instruction type and in a RISC architecture these may be directly coded in the op code. Read control occurs at the start of an instruction cycle. IR latch and occurs at the end of the fetch state. Load control happens at the end of the data fetch state of a load instruction. Load Reg/Reg again depends on the op-code. Register R/W is in the start of the data fetch stage and at the write back stage of an operation. It thus depends on the major state and the instruction.

CISC architecture typically uses a more complex control unit. As we've noted before, the IR is often multiple words, and the control unit has to look at different parts of the IR at different stages of execution. In fact, the entire IR may not be available at once; requiring interlocks with fetch logic to ensure the contents of the IR are valid. There are many more control signals coming out of a CISC control unit, partly to control the more complex addressing logic, but also to directly connect to the many special purpose registers. In RISC architecture, the registers are accessed uniformly in a block so a simple decoder in the register file can select the particular register. In CISC architecture, there are restrictions on the particular registers that can be used by a given instruction and these are enforced by the control unit.

To begin the design of a control unit, we start by listing every control signal in the instruction/data path of the processor. This becomes a list of the control unit's outputs. As input, it has the instruction register, any status information (such as branch flags, interrupts, etc.) from the processor, and a "major state" which simply keeps track of where we are in the execution of an instruction. We always begin an instruction with state 0, which corresponds to fetch. During that state, the control unit outputs the necessary signals to route the contents of the PC to the memory address port, to select and clock the memory until it responds with data from that location, and then cause this data to be latched into the IR.

In CISC architecture, the Fetch may only retrieve the first part of an instruction, and (depending on bits in the IR that are then decoded by the control unit) more words may need to be fetched. In a RISC architecture, a single Fetch retrieves a complete instruction, so we may proceed to the next major state, which is usually to begin fetching data from the registers, while we decode the instruction.

In RISC architecture, "decoding an instruction" mainly means that the instruction type field determines what the control unit will do for the remainder of the instructions. If you think of the CU as a finite state machine, the bits in the type field select the next state following the decode. In terms of a program's logic, this is like selecting a branch in a Switch statement – each branch of the Switch contains the series of steps to be performed for one type of instruction. For example, after decoding a Jump instruction, the control unit outputs the signals required to combine the address portion of the instruction with the upper bits of the PC and load the result back into the PC.

The CU then returns to the Fetch step. Thus, a Jump has three major states (Fetch, Decode, and Complete). For a memory Load instruction, the CU first sends one of the selected register values (the address) to the address port of the memory (via a multiplexer) and signal the

memory to fetch this location. When the memory returns the value, then the CU sends signals to the necessary multiplexer(s) and the register file so that the memory data goes over the Dest bus and is stored in the designated register.

Thus, a Load has four major states (Fetch, Decode, Memory, and Write Back).So, for each type of instruction, and for each major state in each type of instruction, we look at the list of control signals and decide what value each signal must have. In some cases, the value doesn't matter (e.g., if memory isn't selected, it doesn't matter whether it is set to read or write, because it simply won't do anything in either case). You can think of this as a large 2-dimensional table indexed by instruction type and major state. Within each cell of the table is a list of the control signal and their values. One last bit of control output that we've neglected is the control of the major state itself. This is usually a register, as shown above, that is input to the CU. But it also receives its next value on each clock from the CU.

In the above example, the Jump proceeds from State 0 (Fetch) to State 1 (Decode) to State 3 (Complete) and then goes back to State 0. While a Load adds a State 4. In some designs, the state register also encodes the instruction type. Thus, it is really referring to the different states of the finite state machine (FSM) rather than the major steps of the instructions. So, for example, the FSM states for a Load might be the sequence 0, 1, 12, 13. The latter two distinguish Memory and Write Back from the complete stage of the Jump. In other designs, we might see Jump going through states 0, 1, 2, and Load going through 0, 1, 2, 3, with the type field used to distinguish the different behavior of the latter states. This is all just a matter of using somewhat different ways of naming the same things. The important point is just that the CU has the inputs it needs to know what it is supposed to be doing on the present clock and what it will do next. In the CU design process, this translates to ensuring that one of the control signals on the list is the "next state" signal, and that we always specify this in every cell of our table.

## III.DYNAMIC VOLTAGE AND FREQUENCY SCALLING

### *A. Dynamic voltage scalling*

Dynamic Voltage Scaling (DVS) has been a key technique in exploiting the hardware characteristics of processors to reduce energy dissipation by lowering the supply voltage and operating frequency. The DVS algorithms are shown to be able to make dramatic energy savings while providing the necessary peak computation power in general-purpose systems. However, for a large class of applications in embedded real-time systems like cellular phones and camcorders, the variable operating frequency interferes with their deadline guarantee mechanisms, and DVS in this context, despite its growing importance, is largely overlooked/under-developed. To provide real-time guarantees, DVS must consider deadlines and periodicity of real-time tasks, requiring integration with the real-time scheduler. In this paper, I can present a class of novel algorithms called real-time DVS (RT-DVS) that modify the OS's real-time scheduler and task management service to provide significant energy savings while maintaining real-time deadline guarantees. In this paper show through simulations and a working prototype implementation that these RT-DVS algorithms closely approach the theoretical lower bound on energy consumption, and can easily reduce energy consumption 20% to 40% in an embedded real-time system.

Computation and communication have been steadily moving toward mobile and portable platforms/devices. This is very evident in the growth of laptop computers and PDAs, but is also occurring in the embedded world. With continued miniaturization and increasing computation power, I will see ever growing use of power-full microprocessors running sophisticated, intelligent control software in a vast array of devices including digital camcorders, cellular phones, and portable medical devices. Recently, significant research and development efforts have been made on Dynamic Voltage Scaling (DVS). DVS tries to address the tradeoff between performance and battery life by taking into account two important characteristics of most current computer systems: the peak computing rate needed is much higher than the average throughput that must be sustained; and the processors are based on CMOS logic. The first characteristic effectively means that high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance, low-power processor would suffice. I can achieve the low performance by simply lowering the operating frequency of the processor when the full speed is not needed. DVS goes beyond this and scales the operating voltage of the processor along with the frequency. This is

possible because static CMOS logic, used in the vast majority of microprocessors today, has a voltage-dependent maximum operating frequency, so when used at a reduced frequency, the processor can operate at a lower supply voltage.

Since the energy dissipated per cycle with CMOS circuitry scales quadratic ally to the supply voltage (E/V2), DVS can potentially provide a very large net energy savings through frequency and voltage scaling. In time-constrained applications, often found in embedded systems like cellular phones and digital video cameras, DVS presents a serious problem. In these real-time embedded systems, one cannot directly apply most DVS algorithms known to date, since changing the operating frequency of the processor will affect the execution time of the tasks and may violate some of the timeliness guarantees. In this paper, I will present several novel algorithms that incorporate DVS into the OS scheduler and task management services of a real-time embedded system, providing the energy savings of DVS while preserving deadline guarantees. This is in sharp contrast with the average throughput-based mechanisms typical of many current DVS algorithms.

*B. Dynamic frequency scaling*

Dynamic frequency scaling (also known as CPU throttling) is a technique in computer architecture whereby the frequency of a microprocessor can be automatically adjusted "on the fly," either to conserve power or to reduce the amount of heat generated by the chip. Dynamic frequency scaling is commonly used in laptops and other mobile devices, where energy comes from a battery and thus is limited. It is also used in quiet computing settings and to decrease energy and cooling costs for lightly loaded machines. Less heat output, in turn, allows the system cooling fans to be throttled down or turned off, reducing noise levels and further decreasing power consumption. It is also used for reducing heat in insufficiently cooled systems when the temperature reaches a certain threshold, such as in poorly cooled over clocked systems.

The dynamic power (*switching power*) dissipated per unit of time by a chip is $C \cdot V^2 \cdot A \cdot f$, where C is the capacitance being switched per clock cycle, V is voltage, A is the Activity Factor indicating the average number of switching events underwent by the transistors in the chip and f is the switching frequency (as a unit less quantity). The voltage required for stable operation is determined by the frequency at which the circuit is clocked, and can be reduced if the frequency is also reduced. Dynamic power does not account for the total power of the chip, however, as there is also static power, which is primarily because of various leakage currents. Due to static power consumption and asymptotic execution time it has been shown that the energy consumption of a piece of software shows convex energy behavior, i.e., there exists an optimal CPU frequency at which energy consumption is minimal. Leakage current has become more and more important as transistor sizes have become smaller and threshold voltage levels lower. A decade ago, dynamic power accounted for approximately two-thirds of the total chip power. The power loss due to leakage currents in contemporary CPUs and SoCs tend to dominate the total power consumption. In the attempt to control the leakage power high-k metal-gates and power gating have been common methods.

## IV.RESULTS AND DISCUSSIONS

The DVFS is performed with DVS and DFS identically using Xilinx software. Here we consider the system consists of four chips. Figure.2 shows the dynamic voltage scaling and dynamic frequency scaling performance, that the multicore system dynamically selects the voltage and frequency for the processor 1.

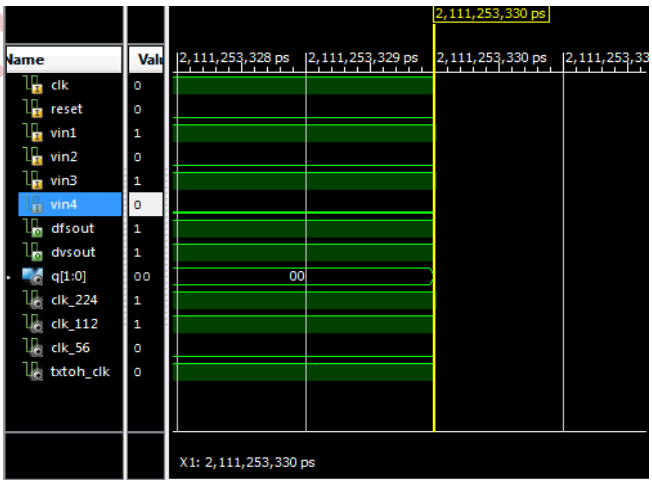


**Figure.2 DVFS output for core 1 execution**

Figure.3 shows the dynamic voltage scaling and dynamic frequency scaling performance, that the multicore system dynamically selects the voltage and frequency for the processor 2. Figure.4 shows the dynamic voltage scaling and dynamic frequency scaling performance, that the multicore system dynamically selects the voltage and frequency for the processor 3. Figure.5 shows the dynamic voltage scaling and dynamic frequency scaling performance, that the multicore system dynamically

selects the voltage and frequency for the processor 4.

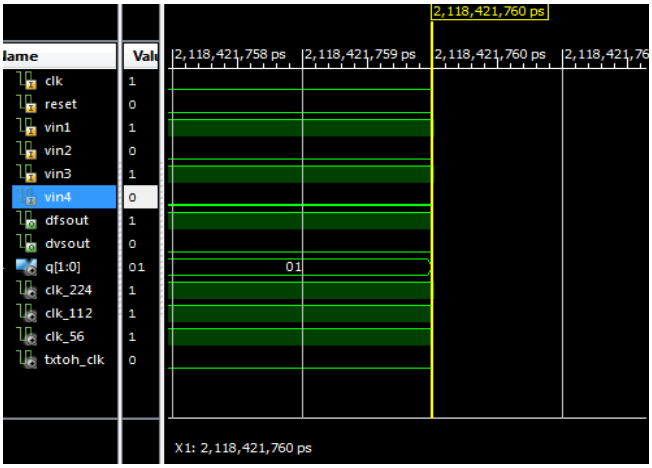


**Figure.3 DVFS output for core 2 execution**

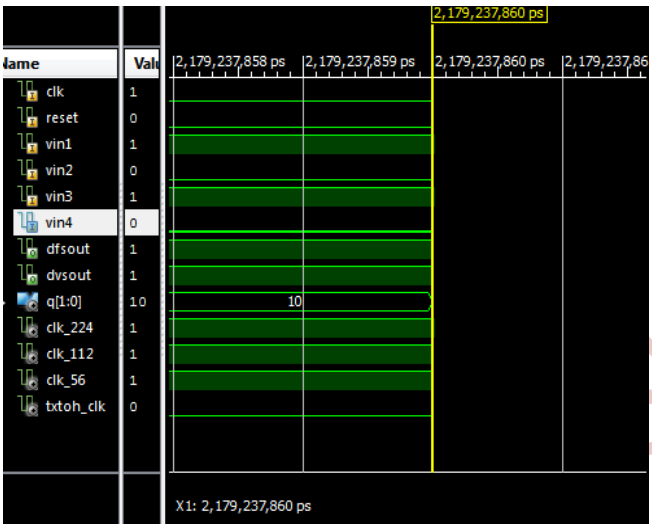


**Figure.4 DVFS output for core 3 execution**

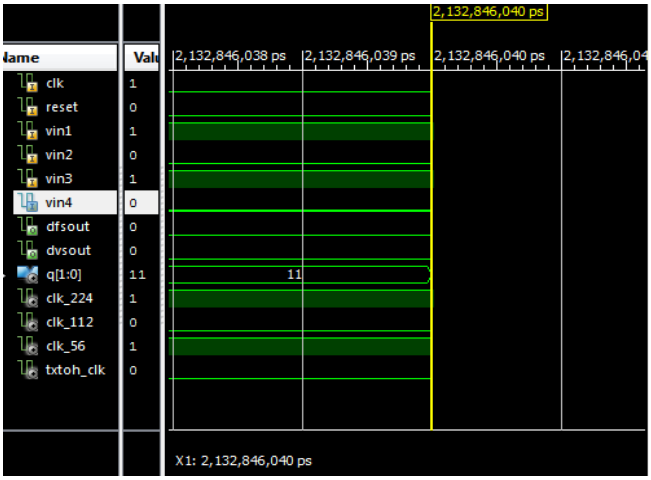


**Figure.5 DVFS output for core 4 execution**

## V.PERFORMANCE COMPARISION

The proposed model performance is compared with an existing AHPH model. From fi rure.6 we fo und that the proposed technique power utilization is 3.3% which is very small compare to the 58.8% of AHPH. Similarly the junction temperature of the multicore system is reduced to 48% from 52.2%. Figure.7 shows the various delays present in the two techniques, from the plot we observed the total delay is reduced in our proposed technique.

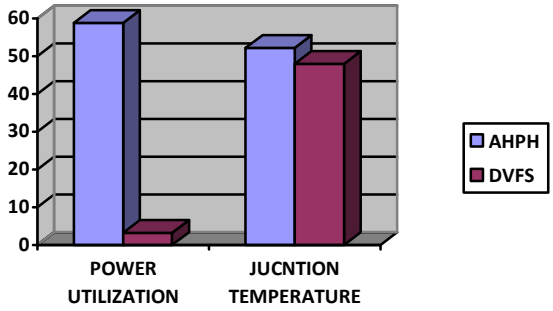


**Figure.6 Power and Temperature comparision**

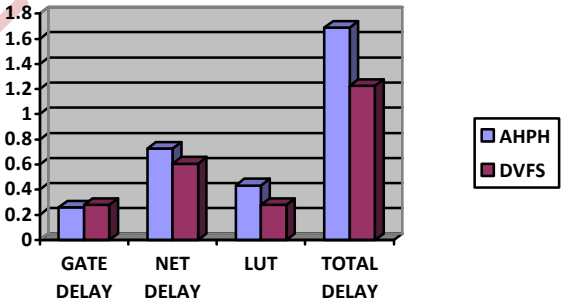


**Figure.7 Delay comparision**

## VI. CONCLUSION

In this proposed work to implement DVFS NoC scheme, a novel distributed management scheme for large-scale chip multiprocessor study reveals that: Global on-chip L2 cache can effectively alleviate the memory pressure caused by the data-thirsty on-chip computing engines. However, its potential is still limited by both the off-chip and the in-chip bandwidth, especially when increasing the number of active threads. On-chip traffic congestion is largely caused by the intensive memory access requests issued from the on-chip cores. These conclusions are generally applicable to a wide variety of many-core processors with similar

design.

## VII. REFERENCES

[1] R. R. Dobkin, R. Ginosar, and C. P. Sotiriou, "*High rate data synchronization in GALS SoCs*," IEEE Trans. Very Large Scale Integr. (VLSI)Syst., vol. 14, no. 10, pp. 1063–1074, Oct. 2006.

[2] J. Donald and M. Martonosi, "*Techniques for multicore thermal management: Classification and new exploration*," in Proc. Int. Symp. Comput. Arch., 2006, pp. 78–88.

[3] V. Hanumaiah, S. Vrudhula, and K. S. Chatha, "*Maximizing performance of thermally constrained multi-core processors by dynamic voltage and frequency control*," in Proc. Int. Conf. Comput.-Aided Design, 2009, pp. 310–313.

[4] S. Herbert and D. Marculescu, "*Analysis of dynamic voltage/frequency scaling in chip-multiprocessors*," in Proc. Int. Symp. Low Power Electron. Design, 2007, pp. 38–43.

[5] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "*An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget*," in Proc. 39th Annu. IEEE/ACM Int. Symp. Microarch., Dec. 2006, pp. 347–358.

[6] R. N. Kalla, B. Sinharoy,W. J. Starke, and M. S. Floyd, "*Power7: IBM's next-generation server processor*," IEEE Micro, vol. 30, no. 2, pp. 7–15, Mar.–Apr. 2010.

[7] C. R. Lefurgy, A. J. Darke, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter, "*Active management of timing guard band to save energy in POWER7*," in Proc. 44th Annu. IEEE/ACM Int. Symp. Microarch., Dec. 2011, pp. 1–11.

[8] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "*Architecting efficient interconnects for large caches with CACTI 6.0*," IEEE Micro, vol. 28, no. 1, pp. 69–79, Jan. 2008.

[9] G. Semeraro, D. H. Albonesi, S. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott, "*Dynamic frequency and voltage control for a multiple clock domain microarchitecture*," in Proc. 35th Annu. ACM/IEEE Int.Symp. Microarch. Nov. 2002, pp. 356–367.