Throughput Regulation in Shared Memory Multicore Processors

X. Chen, H. Xiao, Y. Wardi, and S. Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
Email: xchen318@gatech.edu,
hxiao30@gatech.edu, sudha@ece.gatech.edu, ywardi@ece.gatech.edu

***Abstract*—Performance scaling is now synonymous with scaling the number of cores. One of the consequences of this shift is the increasing difficulty of designing processors with predictable and controllable performance. To address this challenge this paper proposes a chip-scale throughput regulation technique that is based on dynamic tracking of instruction execution dynamics in each core. A new variable gain controller design is developed for regulating the throughput of modern out-of-order cores. The gain is adjusted based on an on-line sensitivity analysis of the core's throughput to the control parameter. We explore throughput regulation using two control paramaters - core frequency and instruction issue width and demonstrate via cycle-level, full system simulation the utility of the proposed regulator on both compute and memory intensive workloads. Performance results are presented for the application to a 16 core, cache coherent 3D multicore processor.**

***Keywords*-multicore processor, throughput regulation, 3D multicore, variable gain controller**

I. Introduction

While performance scaling is now synonymous with scaling the number of cores, power and thermal constraints have precipitated the shift to heterogeneous and asymmetric multicore designs. One of the effects of this shift is the increasing difficulty of designing processors with predictable and controllable performance. For example, the need arises in multimedia applications where a fixed frame rate must be maintained to avoid choppy video or audio. Another application is in hard or soft real-time systems where constant throughput processors enable task and thread schedulers to effectively reason about the consequences of scheduling decisions and thereby provide tight performance bounds. This paper addresses one important aspect of design-for-predictability by addressing the problem of throughput regulation where the instruction throughput of a multicore processor is maintained (regulated) at a set target by varying a microarchitectural parameter such as instruction issue width or core frequency.

Throughput regulation in microprocessors presents several challenges. The first is the time-varying instruction level parallelism (ILP) exhibited by applications. Instruction and resource dependencies affect instruction flows in out-of-order cores and consequently instructions' execution times can vary significantly within an application let alone across different applications. Such variability is amplified in asymmetric multicore architectures comprised of cores that support varying degrees of issue width and complexity, e.g., out-of-order vs. in-order cores. Furthermore, communication delays between cores and other components, such as caches, DRAM, and SSDs, can rarely be predicted reliably. Threads executing on distinct cores interfere with each other in shared caches and on-chip networks introducing dynamically determined delays in instruction execution. It is therefore difficult to develop general analytic models that can relate core and chip instruction throughput to microarchitectural parameters such as frequency or core issue width. All of this suggests the merit of dynamic on-line throughput regulation techniques that are not reliant on a priori, accurate analytical models but rather continually adapt to the processor's dynamics to regulate core and chip instruction throughput at set levels.

This paper proposes such a chip-scale throughput regulation technique that is based on dynamic tracking of instruction execution dynamics in each core. In control-theoretic terms, the key ingredient in feedback-based tracking is an integrator, however, it is well known that an integrator alone often results in oscillations and poor stability margins of the closed-loop system. Therefore, it is common to use it in conjunction with proportional and (sometime) derivative control, resulting in a PID controller [6]. However, for the throughput-regulation considered in this paper we seek a control law that is as simple as possible while rapidly responding to changing program-loads in very short time frames. In other words, our fundamental approach is to tilt the balance between precision and low computational complexity in favor of the latter at the expense of the former. For this to work, the control-system's performance must be robust with respect to modeling uncertainties and real-time load variations during an application program. We realize this design philosophy by using an integrator with *variable gain*, in contrast with typical implementations of PID or PI controllers whose gains are fixed or tuned off line. The gain is adjusted based on an on-line sensitivity analysis of the core's throughput to the control parameter, e.g., core frequency. By setting throughput targets for each core on the chip, we can regulate the throughput of the multicore processor, maintaining it at a level equal to the sum of the core-level instruction throughput targets.

The idea of adjustable-gain integrator was first explored in [1] for regulating dynamic power in 2D multicore processors by DVFS, and extended to instruction-throughput in [2]. In [1] the frequency-to-power model is simple and predictable, thereby avoiding a main difficulty associated with controlling the instruction throughput. In [2] the instruction-flow queueing model is imprecise and does not include the details of the memory system. In contrast, the system-model in this paper describes more-accurately modern 3D processors, includes detailed queueing dynamics of the memory system, and its sensitivity-analysis is more precise; consequently, its simulation results indicate faster convergence (by an order of magnitude) of the control algorithm. Moreover, we test the proposed regulation technique on control not only by frequency but also by issue width which is not considered in [2].

This paper seeks to make the following contributions.

1) A new variable gain controller design for regulating the throughput of modern out-of-order cores.
2) The use of detailed, on-line sensitivity analysis to dynamically estimate sensitivity of instruction throughput to microarchitectural parameters such as instruction issue width and frequency
3) The application of this regulator design to 2D and 3D multicore processors.
4) An evaluation of the regulator design with full system, cycle-level multicore simulator executing industry standard benchmark applications under a Linux OS on homogeneous (out-of-order) multicore processors.

The rest of the paper is organized as follows. Section II describes the model and defines the problem. Section III presents the closed loop control in a general setting. Section IV applies the model to throughput regulation while conclusions are summarized in Section V.

II. Multicore Processor Model

There are two emergent trends that are shaping the future landscape of high performance embedded systems. The first is 3D packaging. Multiple dies are stacked vertically with die-to-die interconnects realized with through silicon vias (TSVs). The reduced physical geometries enable higher performance in a smaller footprint, but present thermal and power management challenges [11]. The second is near data processing where cores are integrated within the memory system to reduce the cost of data movement while exploiting the much higher intra-die memory bandwidth. Taken together, both trends can boost the computing power in embedded multimedia applications such as smart cameras, unmanned arial vehicles, and smart phones. Consequently, we evaluate our regulators in such an emergent processor.

The modeled system is a 16-core homogenous x86 processor die coupled with a shared last level cache (LLC) cache die as illustrated in Figure (1). The cores reside on the bottom die and are interconnected by a 2D torus - one core and its L1 cache are connected to a single router. Each of the x86 out-of-order cores includes the front-end, L1 instruction cache, the out-of-order scheduler, the integer ALU, the floating-point unit and a private L1 data cache of 32KB. The structure of a typical out-of-order core is shown in Figure 3. The die floor plans shown in the figure assumes implementation at the 16 nm technology node. The shared L2 LLC SRAM cache resides on the next tier, with 16 banks each of 2MB while each bank is associated with the corresponding core on the lower die. The memory hierarchy is coherent implementing the MOESI protocol. The remaining component is the DRAM stack comprised of the next 8 dies stacked on top of the first two tiers [22]. The DRAM is modeled after the Hybrid Memory Cube (HMC) with 16 channels, where each channel has 8 ranks (on per die) and 2 banks per rank. Each channel has a memory controller attached to the router associated with a core. Finally, the system simulation model computes the power dissipation and resulting thermal fields produced by the package. The throughput targets are typically set to ensure that the total power dissipation does not exceed the thermal design power (TDP) of the package - an important role for throughput regulators.

We investigate the design of a throughput regulator for each core using different microarchitectural parameters to demonstrate the generality of our approach. The first parameter we study is core frequency for which the model and approach are described in Section IV-B. A key set of microarchitecture parameters control the bandwidth of certain pipeline stages, e.g., fetch width, dispatch width, issue width, and retire width. We assume that the front-end decode and rename pipeline stages match the dispatch width. It is important that the instruction delivery subsystem provides a sustained flow of instructions that matches the execution rate requirements of the processor to issue, execute, and commit instructions. However, the specific number of instructions that can be issued in a cycle are a function of the dynamically occurring dependencies between instructions. Effective throughput regulation must track these dependencies. Therefore the second control parameter we study is the issue width. Section IV-C describes this approach in detail.

III. Feedback Control Law

The purpose of the controller described in this section is to regulate the instruction-rate of each core to a given setpoint reference, and we assume that for this purpose each core has its own controller and target reference. Furthermore, the vector of setpoints is computable by a high-level performance-management module according to chip-level considerations such as power capping, load balancing, or temperature management. These computations are performed off line at far-longer time frames than the throughput controller

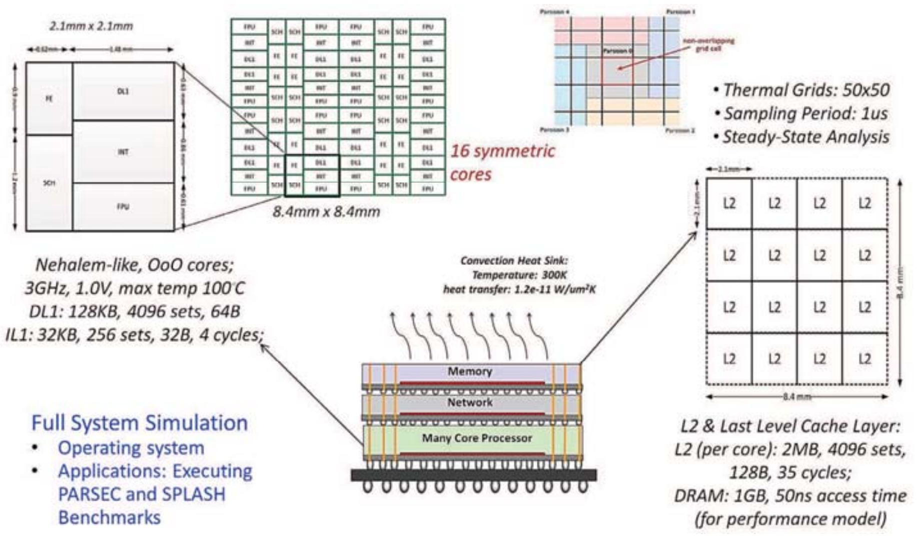

2.1mm x 2.1mm
16 symmetric cores
8.4mm x 8.4mm
• Thermal Grids: 50x50
• Sampling Period: 1us
• Steady-State Analysis
Nehalem-like, OoO cores;
3GHz, 1.0V, max temp 100°C
DL1: 128KB, 4096 sets, 64B
IL1: 32KB, 256 sets, 32B, 4 cycles;
Convection Heat Sink:
Temperature: 300K
heat transfer: 1.2e-11 W/um²K
Memory
Network
Many Core Processor
Full System Simulation
• Operating system
• Applications: Executing PARSEC and SPLASH Benchmarks
L2 & Last Level Cache Layer:
L2 (per core): 2MB, 4096 sets, 128B, 35 cycles;
DRAM: 1GB, 50ns access time (for performance model)

Figure 1. System Architecture Model Including Power and Thermal Models

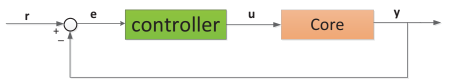


Figure 2. Close-loop control

described below, and we are not concerned with their specific details. This section describes the control law in general terms, and the next section specifies it for the special case of throughput regulation.

Consider the configuration shown in Figure (2), associated with a particular core. Suppose that the instruction throughput is measured over contiguous observation periods, called *control cycles*, denoted by C_n, $n = 1,2,\ldots$; the throughput measured during C_n is denoted by y_n. Let u_n denote the control variable during C_n, where we consider u_n to be either the core's clock frequency or the issue width. u_n is assumed to be assigned by the controller at the start of C_n and maintain its value throughout that control cycle, while y_n is assumed to be computed from measurements during C_n and be obtained at the end of it. Let r be the setpoint target for the output y_n, and the objective of the controller is to ensure that y_n approaches r.

The action of the controller is defined by the equation

$$u_n = u_{n-1} + A_n e_{n-1}, \qquad n = 1,2,\ldots, \tag{1}$$

where A_n is its gain during C_n. We observe that for the case where $A_n = 1$ for all $n = 1,\ldots$, the controller acts as an adder, or integrator in the context of discrete time, in that its output u_n is the sum of its past inputs, e_k, $k = 1,\ldots,n-1$. In the general case defined in Equation (1) the gain A_n is a function of n, and hence we say that the controller is an integrator with adjustable gain. As for the error signal, it is evident from Figure (2) that

$$e_n = r - y_n. \tag{2}$$

For reasons explained in [1], [20] and summarized below, we define the gain A_n as

$$A_n = \xi \left(\frac{dy_{n-1}}{du_{n-1}} \right)^{-1}, \tag{3}$$

where $\xi \in (0,1)$ is a given constant determined experimentally in a way that maximizes the controller's tracking performance. The term $\frac{dy_{n-1}}{du_{n-1}}$ in Equation (3) is the sample derivative of the core's input-output relation during C_{n-1}. We point out that the relationship between u_{n-1} and the throughput y_{n-1} during C_{n-1} cannot be described by a simple function, but rather by a complicated queueing model (described below) that defies analysis. However, it is fairly simple to compute y_{n-1} from observation of the system (core) by simply counting the number of instructions completed during C_{n-1} and dividing it by the duration of C_{n-1}. This procedure yields the *sample throughput* and not the *mean throughput*, where different control cycles are associated with different sample paths. The term $\frac{dy_{n-1}}{du_{n-1}}$ is the derivative of that sample relation, hence called the *sample derivative*. As we shall see it can be computed quite easily from the sample path by observing the instructions' schedule in the core throughout C_{n-1}. Furthermore, the result is available at the end of C_{n-1}, hence can be used to compute u_n via Equation (1) at the start of C_n.

The rationale behind the definition of the gain A_n via Equation (3) can be seen by considering the case where the $u_n - y_n$ relation is given by a deterministic function $L(u)$, so that $y_n = L(u_n)$. For $\xi = 1$, it can be seen that the control law implements the Newton-Raphson method for solving the equation $L(u) = r$, which is the objective of tracking. Convergence of the Newton-Raphson method is known to be robust to variations in that equation as well as to computational errors [13], and therefore we expect the control law to yield tracking regulation in the stochastic, time-varying setting under consideration. The results, presented below, satisfy our expectation. The factor $\xi \in (0,1)$ in Equation (3) is used to reduce oscillations that are due to the randomness.

IV. Throughput Regulation

To quantify the throughput, recall that the instruction flow through the core involves four steps, or stages (see Figure (3)): Issue, Execute, Memory, and Commit. We next derive the equations that describe these four steps. To start with the Issue step, consider a sequence of instructions, denoted by $I_1, I_2, \ldots$, according to their issue order. Let ξ_i denotes the arrival time of instruction I_i to the Reorder Buffer (ROB) in terms of clock cycles. If the instruction has a data dependency, we use $k(i)$ to denote the index of the instruction that computes the last operand required for instruction I_i . Let τ denote the core's cycle time, and denote by α_i the enqueue time of I_i, namely the time that all the operands of I_i are available and the instruction is ready to be executed. Then

$$\alpha_i \;=\; max\{\; \xi_i\tau \;,\; \beta_{k(i)} \;\} + \tau. \tag{4}$$

Secondly, in Execute stage, assume that the execution time of a non-memory instruction I_i is approximated by $\mu_i\tau$, where μ_i is total number of clock cycles it takes the execution unit to process instruction I_i . Denote by β_i the completion time of executing I_i. Then,

$$\beta_i \;=\; \alpha_i + \mu_i\tau, \tag{5}$$

and we note that β_i is also the time that the result of instruction I_i becomes available as an operand for other instructions.

Thirdly, if instruction I_i is a memory instruction, the memory hierarchy is involved in the process. Let us denote the sequence of instructions I_i that are in the memory path by $I_{i(j)},\ j = 1,2,\ldots$. The processing time of instruction $I_{i(j)}$ in the cache is $\nu_{i(j)}\tau$, where $\nu_{i(j)}$ is the number of clock cycles it takes to proceed the instruction in cache. The completion time of executing a cache-hit instruction is the dequeuing time from cache, that is,

$$\gamma_{i(j)} \;=\; max\{\; \alpha_{i(j)} + \nu_{j(i)}\tau,\; \delta_{i(j)-\lambda} \;\}, \tag{6}$$

where λ is the total number of MSHR (Miss Status Holding Register) entries. We assume if the number of instructions in MSHR reaches λ, the whole memory system stops processing.

If instruction $I_{i(j)}$ is a cache miss then it needs to access other storage devices such as DRAM. The major part of its latency can be approximated by a term denoted by $MEM_{i(j)}$, which typically is hundreds of clock cycles and hence one-to-two orders of magnitude longer than compute instructions. Note that $MEM_{i(j)}$ is independent of τ since the clock of such memory systems is different from the clock of cores and caches. The completion time of a cache-miss instruction $I_{i(j)}$ in Memory stage is its departure time from the MSHR back to execution, and denoting it by $\delta_{i(j)}$, it can be seen from the above discussion that

$$\delta_{i(j)} \;=\; max\{\; \gamma_{i(j)} + M_{i(j)}\tau + MEM_{i(j)} \;,\; \delta_{i(j)-1}\}, \tag{7}$$

where $M_{i(j)}\tau$ is the proceeding time in MSHR.

Thus, the completion time of executing instruction I_i is computed as follows:

$$\beta_i = \begin{cases} \alpha_i + \mu_{(i)}\tau, & \text{if instruction } I_i \text{ is a} \\ & \text{non-memory instruction} \\ \gamma_{i(j)}, & \text{if instruction } I_i \text{ is a cache} \\ & \text{hit memory instruction} \\ \delta_{i(j)}, & \text{if instruction } I_i \text{ is a cache} \\ & \text{miss memory instruction.} \end{cases} \tag{8}$$

Finally, let us consider the final stage, Commit. The order of departure of instructions should be the same as their arrival order. Let d_i denote the time that instruction I_i in the ROB is committed (dequeuing time), then we have

$$d_i \;=\; max\{\; \beta_i + \tau \;,\; d_{i-1} + \tau \;\}. \tag{9}$$

Considering all of this during a control cycle C_n comprised of M instructions, the throughput y_n is given by

$$y_n = \frac{M}{d_M}. \tag{10}$$

We will use these equations to compute the sample derivatives, $\frac{dy_n}{du_n}$, for the two aforementioned control parameters: the core's clock rate and instruction issue width. Recall that these derivatives define the gain A_n by Equation (3). They will be computed by a technique called Infinitesimal Perturbation Analysis (IPA), a general method for estimating performance sensitivities in discrete event dynamic systems [3]. While the details of the derivations are relegated to the appendix, the simulation results of experiments with the control algorithm are presented in this sections.

A. *Simulation Model*

The control techniques proposed in this paper have been simulated and tested by Manifold, a discrete event simulation framework for modern multicore computer architectures [18]. Manifold enables cycle-level full system processor simulation, i.e. application and operating system binaries driving cycle-level models of cores, coherent caches, on-chip networks, and the DRAM system. Manifold also

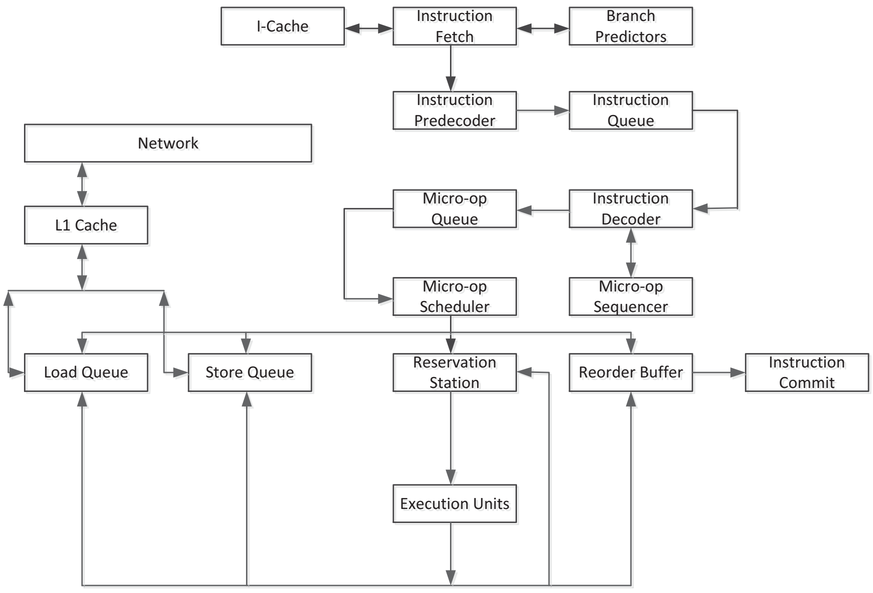


Figure 3. Data Flow in an Out-of-order Execution Core

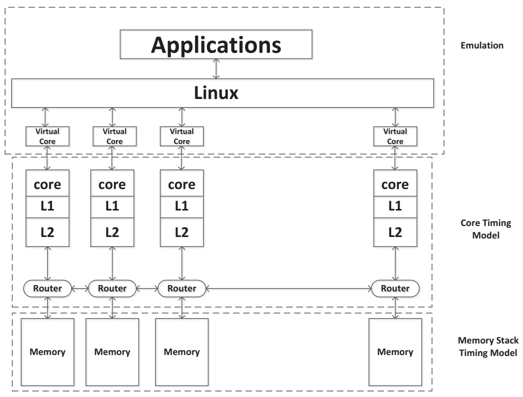


Figure 4. Manifold Execution Model

supports dynamic voltage frequency scaling and is coupled to energy and thermal models via the Energy Introspector multi-physics modeling library [15]. Figure (4) depicts a Manifold cycle-level model of the 3D near-data processor core described in Section II, whose specific parameters are shown in Table I. The simulation results are describes in the next two subsections.

B. Throughput Regulation Using Clock Frequency

In this experiment we execute two SPLASH-2 benchmarks, Cholesky and Ocean-nc [21]. Cholesky is a computation intensive application while Ocean-nc is a memory intensive application. Eight cores execute the Cholesky benchmark, and eight cores execute the Ocean-nc benchmark. Each control cycle consists of 50,000 instructions, chosen to balance the settling (convergence) time with local high-frequency oscillations. The frequency-range of the cores is 0.5 GHz to 5 GHz. These simulations assume that a continuous range of frequencies are feasible. We set the target throughput of each core at 4000 MIPS (Million Instruction Per Second) for Cholesky, and 1000 MIPS for Ocean-nc.

A typical simulation run for the Cholesky benchmark (chosen at random from the eight cores executing this benchmark) is shown in Figure (5), where the horizontal axis indicates time in ms and the vertical axis indicates instruction throughput for a single core. The value of ξ in Equation (3) is $\xi = 1$. The total run time of 333 ms is

Parameters	Out-of-order Core
Architectural Configuration	
ISA	x86 IA32
Pipeline Depth	10 stages
Fetch/Decode	4 instructions
Execution	6 Issue ports
L1 Cache	8-way 16KB/core
L2 Cache	64-way 2MB/bank, 16 banks
Physical Configuration	
Clock Frequency	0.5-5.0GHz
Supply Voltage	0.5-1.2V
Feature Size	16nm

Table I
SIMULATED PROCESSOR CORE CONFIGURATION

the duration of the Cholesky program, and it corresponds to about 10^6 control cycles at $2GHz$ clock frequency. The target throughput $4000MIPS$ is the highest tracking execution rate for Cholesky benchmark. We discern from the graph a fast rise in throughput from an initial value of 400 MIPS to about 4300 MIPS in 0.8 ms. Thereafter the throughput stabilizes at about the target value of 4,000 MIPS except for sporadic variations which are due to variable program workload and other random aspects of the system. However, the controller seems to compensate for them in short time-frames. Furthermore, the average throughput computed over the time interval $[0.8ms,333ms]$ (soon after the throughput has reached the target value) is 3964.4 MIPS, which is quite close to the target throughput of 4,000 MIPS. Similar results for the Ocean-nc benchmark are shown in Figure (6). Part (a) of the figure depicts the graph of the instruction throughput for the first 35 ms of the program, while part (b) shows the throughput for the entire run of 333 ms. The reason for restricting the results to a subset of the program's duration is that the graph shows the rapid convergence of the throughput from its initial value of 600 MIPS to about the target value at time 0.5 ms, which is not visible in part (b) of the figure. In both parts of the figure we discern fluctuations of the throughput from its target value, but the control algorithm stabilizes the throughput rapidly. These fluctuations (oscillations) are more pronounced than in the results concerning the Cholesky benchmark; the reason is that Ocean-nc is more memory intensive than Cholesky, hence it experiences wider load variations. The frequency keeps changing in every control cycle, which is $100\mu s$. As mentioned earlier, the parameter $\xi \in (0,1)$ in Equation (3) can be used to reduce oscillations in the throughput profile. To test this point we simulated the control algorithm with $\xi = 0.2$ for both the Cholesky and Ocean-nc benchmarks. The results, shown in Figure (7) and Figure (8), respectively, exhibit fewer and smaller oscillations but larger settling times as compared to the respective results in Figure (5) and Figure (6), resp., where $\xi = 1.0$. This is not surprising in light of the fact that the controller's gain is smaller. In fact, the measured average throughput for Choleaky is 3989.8 MIPS for $\xi = 0.2$ and 3964.4 MIPS for $\xi = 1.0$, whereas for Ocean-nc, it is 1004.6 MIPS for $\xi = 0.2$ and 1008.9 MIPS for $\xi = 1.0$.

C. Throughput Regulation Using Issue Width

The experiment uses the same two SPLASH-2 benchmarks as in the last experiment, Cholesky and Ocean-nc. The issue width of each core ranges from 1 to 4, and hence the control parameter u is constrained to the set $U := \{k : k = 1,2,3,4\}$. The target throughput of each core for the Cholesky benchmark is 1.2 GIPS, and for the Ocean-nc benchmark is, 1.0 GIPS. The control cycle is 0.1 ms, and we note that this is in units of time rather than instructions per second, as it was in the frequency-control experiment.

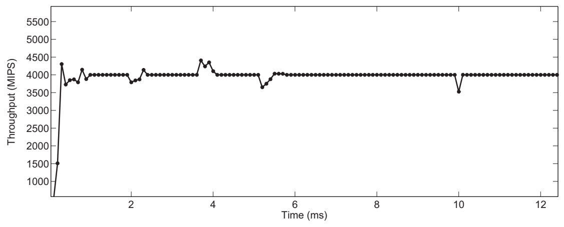
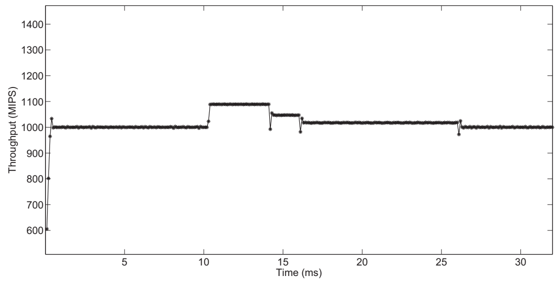
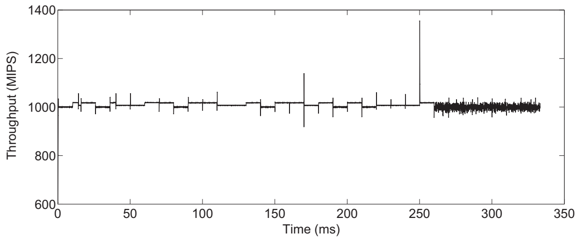


Figure 5. Frequency regulation: Cholesky



(a)



(b)

Figure 6. Frequency regulation: Ocean-nc

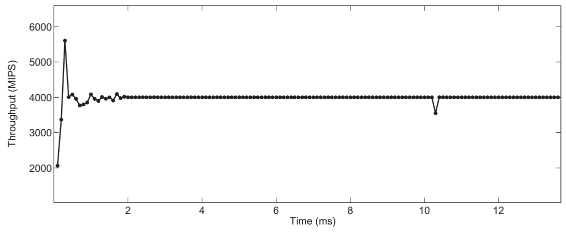


Figure 7. Frequency regulation (modified algorithm): Cholesky

The IPA derivative of the u-to-y relation, presented in the appendix, assumes that u has a continuous range of the control. However, the implementation of the controller allows only the aforementioned, four-point set of values. To address this issue we modify the control algorithm in the following way. Suppose that $u_{n-1} \in U$ is the control variable at the start of the control cycle C_{n-1}. We replace Equation (1) by the following calculation of the auxiliary variable v_n,

$$v_n = u_{n-1} + A_n e_{n-1}, \tag{11}$$

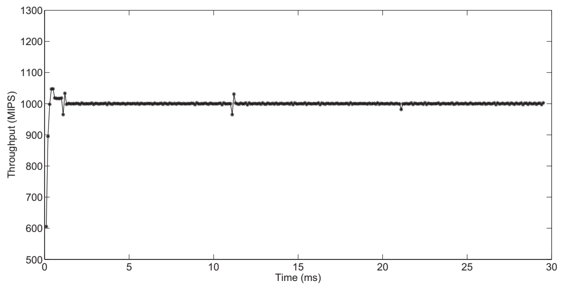


Figure 8. Frequency regulation (modified algorithm): Ocean-nc

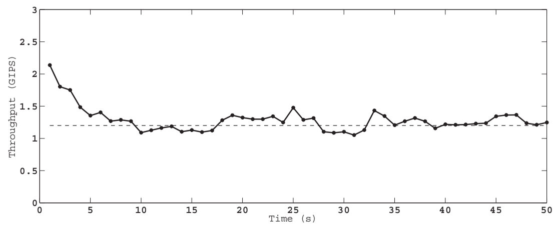


Figure 9. Issue width regulation: Cholesky

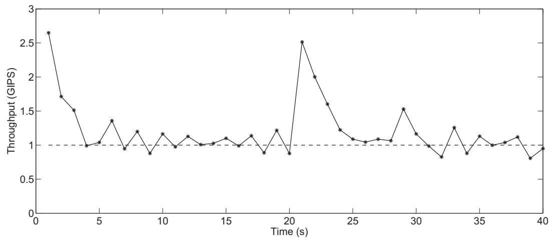


Figure 10. Issue width regulation: Ocean-nc

and then take u_n to be the point in U closest to v_n. In all runs we took $\xi = 1$ in Equation (3). The results are shown in Figure (9) for Cholesky and Figure (10) for Ocean-nc, respectively. Still, the target throughput of each core for the Cholesky benchmark is 1.2 GIPS, and for the Ocean-nc benchmark it is 1.0 GIPS. Although the figures indicate convergence towards the target values (notwithstanding the oscillations that are due to the system's variability), a bias is discerned from the graphs. As a matter of fact, the average instruction throughput was computed at 1.2868 GIPS for the Cholesky experiment, and 1.2025 GIPS for the Ocean-nc benchmark. The bias is due in part to the quantization inherent in the computation of u_n, which permits the control to get trapped in a set of values close to the target.

To reduce the bias we modify Equation (11) as follows. Let α_{n-1} be a running variable computed at the start of C_{n-1}. At the start of C_n, compute v_n via

$$v_n = u_{n-1} + A_n e_{n-1} + \alpha_{n-1}, \tag{12}$$

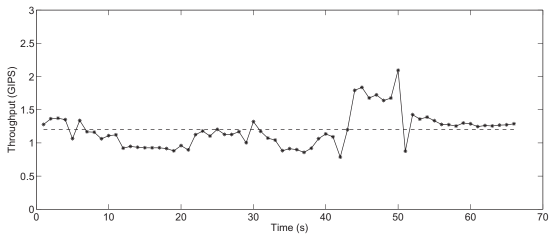


Figure 11. Issue width regulation: Cholesky (with accumulator)

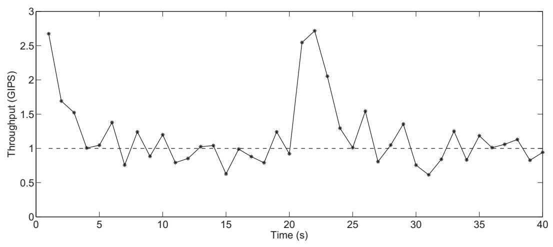


Figure 12. Issue width regulation: Ocean-nc (with accumulator)

then set u_n to the value in U closest to u_n, and set

$$\alpha_n = \alpha_{n-1} + u_n - v_n. \tag{13}$$

In other words, α_n tracks to cumulative quantization error in successive iterations of the control algorithm. Running the experiments with this modification we obtained the results shown in Figures (11) (Cholesky) and Figures (12) (Ocean-nc). The average throughput is 1.1936 GIPS for Cholesky and 1.1839 GIPS for Ocean-nc, which is closer to the target values of the benchmarks than the results of the controller without quantization-error accumulation. The pipeline width varies from 1 to 4, and it changes in every couple control cycles depending on the application phase.

V. Concluding Remarks

This paper addressed the challenge of realizing predictable performance for the execution of multithreaded applications on cache coherent shared memory multicore processors - specifically realizing predictable throughput. This was achieved via the design of a per core variable gain throughput regulator that adjusted a control parameter (core frequency or instruction issue width) to maintain fixed instruction throughput/core in the presence of dynamically varying parallelism and inter-instruction dependencies in the instruction stream. The regulator illustrates a novel application of on-line sensitivity analysis to periodically change the controller gain thereby avoiding the need for any a priori profiling or characterization of the applications. The performance results on both compute and memory intensive applications demonstrate robust and agile tracking performance. Our future research involves using this capability for designing soft real time systems with controllable and predictable performance.

VI. Appendix

In this section we derive the IPA derivatives $\frac{dy_n}{du_n}$ discussed in Section IV, where u_n is the control input to a core (clock rate or issue width) and y_n is the resulting throughput during the *nth* control cycle. In the forthcoming discussion we omit the explicit notational dependence on n, and use prime notation to indicate derivative with respect to the indicated variable.

To begin with, consider the case where the control parameter is the clock frequency, and hence $u = \tau^{-1}$ in Equations (4) - (10). Therefore we have that

$$y'(\tau) = y'(u)\frac{du}{d\tau} = -\frac{y'(u)}{\tau^2}. \tag{14}$$

The term $y'(\tau)$ can be computed by taking derivatives in Equations (4) - (10, as follows.

By Equation (4),

$$\alpha_i'(\tau) = \begin{cases} \xi_i + 1, & \text{if all the operands of instruction } I_i \text{ are ready before } I_i \text{ arrives at ROB.} \\ \beta_{k(i)}'(\tau) + 1, & \text{otherwise.} \end{cases} \tag{15}$$

while by Equation (6),

$$\gamma_{i(j)}'(\tau) = \begin{cases} \alpha_{i(j)}'(\tau) + \nu_{i(j)}, & \text{if Load/Store Unit does not stop after processing instruction } I_{i(j)-1}. \\ \delta_{i(j)-\lambda}'(\tau), & \text{otherwise.} \end{cases} \tag{16}$$

Next, by (7),

$$\delta_{i(j)}'(\tau) = \begin{cases} \gamma_{i(j)}'(\tau) + M_{i(j)}, & \text{if instruction } I_{i(j)-1} \text{ leaves MSHR before instruction } I_{i(j)} \text{ is completed.} \\ \delta_{i(j)-1}'(\tau), & \text{if instruction } I_{i(j)-1} \text{ stays in MSHR when instruction } I_{i(j)} \text{ is completed.} \end{cases} \tag{17}$$

and hence, and by Equations (8) and (15) - (17), we obtain,

$$\beta_i'(\tau) = \begin{cases} \alpha_i'(\tau) + \mu_{(i)}, & \text{if instruction } I_i \text{ is not a memory instruction.} \\ \gamma_{i(j)}'(\tau), & \text{if instruction } I_i \text{ is a cache hit memory instruction.} \\ \delta_{i(j)}'(\tau), & \text{if instruction } I_i \text{ is a cache miss memory instruction.} \end{cases} \tag{18}$$

By Equation (9) we have that

$$d_i'(\tau) = \begin{cases} \beta_i'(\tau) + 1, & \text{if the entry of instruction } I_i \text{ is head of the ROB} \\ d_{i-1}' + 1, & \text{if the entry of instruction } I_{i-1} \text{ still remains in the ROB} \end{cases} \tag{19}$$

and we note that this is a recursive equation which, with the aid of (18), gives out $d_i'(\tau)$ for all $i = 1, \ldots, M$, and in particular, we can obtain $d_M'(\tau)$.

Recall (Equation (10)) that $y = M/d_M$, and hence,

$$y'(\tau) = -M\frac{d_M'(\tau)}{d_M(\tau)^2}. \tag{20}$$

This, in conjunction with (14), gives

$$y'(\tau) = \frac{1}{M}\Big(\frac{y}{u}\Big)^2 d_M'(\tau). \tag{21}$$

Consider next the case where the control parameter, u, is the issue width, hence an integer which we assume to be in the set $\{1,2,3,4\}$. In order to apply the IPA derivative we consider an abstraction where $u \in [1,4]$ is a continuous variable. As before we let y be the core's throughput during a typical control cycle, and we estimate the IPA derivative $\frac{dy}{du}$. In contrast to the frequency-regulation problem where we define the control cycle in terms of the number of instructions, here we define it in terms of a given time. Therefore, in Equation (10), the term d_M is a constant independent of u, while M is a function of u. Furthermore, assuming that all of the four pipelines are homogeneous, M is related to u via the equation

$$M = \Phi\Psi u, \tag{22}$$

where Φ is the instruction flow rate on a single pipeline and $\Psi \in (0.5, 1.0]$ is a number that reflects the relationship between the total throughput and the execution rate of each single pipeline. Φ and Ψ vary from one control cycle to the next depending on the application patterns. However, their variations between consecutive cycles often are small enough to justify the following procedure: Measure the product $\Phi_{n-1}\Psi_{n-1}$ during a control cycle, and use the result in Equation (22) for the next control cycle. Thus, Equation (22) becomes $M_n = \Phi_{n-1}\Psi_{n-1}u_n$, where the subscript n indicates quantities associated with the *nth* control cycle, C_n. Then, an adequate approximation to the sample derivative is

$$y_n'(u_n) = \Phi_{n-1}\Psi_{n-1}. \tag{23}$$

Of course the effects of drastic changes in the product $\Phi_{n-1}\Psi_{n-1}$ would be delayed by one control cycle.

Acknowledgment

The research described in this paper has been supported in part by the NSF under Grant CNS-1239225.

References

[1] N. Almoosa, W. Song, S. Yalamanchili, and Y. Wardi, “A Power Capping Controller for Multicore Processors,” in *Proc. American Control Conference*, Montreal, Canada, June 27-29, 2012.

[2] N. Almoosa, W. Song, S. Yalamanchili, and Y. Wardi, “Throughput Regulation in Multicore Processors via IPA,” in *Proc. 51st IEEE Conference on Decision and Control*, Maui, Hawaii, December 10-13, 2012.

[3] C.G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, Springer Science+Business Media, New York, New York, 2008.

[4] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, “An Integrated Quadcore Opteron Processor” *International Solid State Circuits Conference*, pp. 102-103, Feb. 2007.

[5] M.S. Floyd, S. Ghiasi, T.W. Keller, K. Rajamani, F.L. Rawson, J.C. Rubio, and M.S. Ware, “System power management support in the ibm power6 microprocessor,” *IBM Journal of Research and Development*, Vol.51, No. 6, 2007.

[6] G. Franklin, J.D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamical Systems*, Pearson, Upper Saddle River, New Jersey, Sixth Edition, 2009.

[7] M. Gries, U. Hoffmann, M.Konow, and M.Riepen, “SCC: A Flexible Architecture for Many-Core Platform Research,” *Computing in Science & Engineering*, Vol. 13, Issue No. 06, pp. 79-83, 2011.

[8] W. Kim, M. Gupta, G. Wei, and D. Brooks, “System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators,” *High Performance Computer Architectures*, pp 123-134, 2008.

[9] J. Kuang and L. Bhuyan, “Optimizing Throughput and Latency under Given Power Budget for Network Packet Processing,” *Proc. INFOCOM*, March 14-19, 2010.

[10] K. Li, “Optimal configuration of a multicore server processor for managing the power and performance tradeoff,” *The Journal of Supercomputing*, Vol. 61, Issue 1, pp 189-214, July 2012.

[11] G.H. Loh, Y. Xie, and B. Black, “Processor Design in 3D Die-Stacking Technologies”, *IEEE Micro*, Vol. 27, Issue 3, pp. 31-48, 2007.

[12] D. Lohn, M. Pacher, and U. Brinkschulte, “A Generalized Model to Control the Throughput in a Processor for Real-Time Applications,” *14th IEEE International Symposium on Object Component Service-Oriented Real-Time Distributed Computing*, pp. 83-88, 2011.

[13] P. Lancaster “Error analysis for the Newton-Raphson method,” in *Numerische Mathematik, 1966*, Vol. 9, pp. 55–68, 1966.

[14] P. Macken, M. Degrauwe, M.V. Paemel, and H. Oguey, “A voltage reduction technique for digital systems,” in *Proc. IEEE International Solid- State Circuits Conference*, pp. 238-239, 1990.

[15] W. Song, S. Mukhopadhyay, and S. Yalamanchili, “Energy Introspector: a parallel, composable framework for integrated power-reliability-thermal modeling for multicore architectures,” *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2014.

[16] J. Suh and M. Dubois, “Dynamic MIPS Rate Stabilization in Out-of-Order Processors,” in *Proc. 36th annual international symposium on Computer architectures* (ISCA ’09), Pages 46-56, 2009.

[17] G. Sun, C.G. Cassandras, Y. Wardi, C.G. Panayiotou, and G.F. Riley, “Perturbation analysis and optimization of stochastic flow networks,” *IEEE Trans. Automatic Control*, Vol. 49, No. 12, pp.2143-2159, 2004.

[18] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili, ‘‘Manifold: A Parallel Simulation Framework for Multicore Systems,” in *Proc. Performance Analysis of Systems and Software*, (ISPASS), pp. 106 - 115, 2014 .

[19] J. Wang, J. Beu, S. Yalamanchili, and T. Conte, “Designing Configurable, Modifiable And Reusable Components For Simulation of Multicore Systems,” in *Proc. High Performance Computing, Networking, Storage and Analysis*, (SCC), pp. 472 - 476, 2012 .

[20] Y. Wardi, C. Seatzu, X. Chen, and S. Yalamanchili, “Performance Regulation of Stochastic Discrete Event Dynamic Systems Using Infinitesimal Perturbation Analysis”, *Nonlinear Analysis: Hybrid Systems*, submitted, 2014.

[21] S.C. Woo , M. Oharat, E. Torriet, J. Singhi, and A. Guptat , “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proc. 22nd Annual International symposium on Computer architectures* (ISCA’95), pp. 24-36, 2005.

[22] H. Xiao, W. Yueh, S. Mukhopadhyay, and S. Yalamanchili, “Multi-Physics Driven Co-design of 3D Multicore Architectures,” in *Proc. International Technical Conference and Exhibition on Packaging and Integration of Electronic and Photonic Microsystems (InterPACK) and the International Conference on Nanochannels, Microchannels and Minichannels (ICNMM)*, July. 2015.

[23] S. Yalamanchili, G.F. Riley, and T.M. Conte, “http://manifold.gatech.edu/”.

[24] Y. Yao and Z. Lu, “Fuzzy Flow Regulation for Network-on-Chip based Chip Multiprocessors Systems,” in *Proc. 19th Asia and South Pacific Design Automation Conference*, pp 343-348, 2014.