Dynamic MIPS Rate Stabilization for Complex Processors

JINHO SUH, CHIEH-TING HUANG, and MICHEL DUBOIS, University of Southern California

Modern microprocessor cores reach their high performance levels with the help of high clock rates, parallel and speculative execution of a large number of instructions, and vast cache hierarchies. Modern cores also have adaptive features to regulate power and temperature and avoid thermal emergencies. All of these features contribute to highly unpredictable execution times. In this article, we demonstrate that the execution time of in-order (IO), out-of-order (OoO), and OoO simultaneous multithreaded processors can be stable and predictable by stabilizing their mega instructions executed per second (MIPS) rate via a proportional, integral, and differential (PID) gain feedback controller and dynamic voltage and frequency scaling (DVFS).

Processor cores in idle cycles are continuously consuming power, which is highly undesirable in systems, especially in real-time systems. In addition to meeting deadlines in real-time systems, our MIPS rate stabilization framework can be applied on top of it to reduce power and energy by avoiding idle cycles. If processors are equipped with MIPS rate stabilization, the execution time can be predicted. Because the MIPS rate remains steady, a stabilized processor meets deadlines on time in real-time systems or in systems with quality-of-service execution latency requirements at the lowest possible frequency.

To demonstrate and evaluate this capability, we have selected a subset of the MiBench benchmarks with the widest execution rate variations. We stabilize their MIPS rate on a 1GHz Pentium III-like OoO single-thread microarchitecture, a 1.32GHz StrongARM-like IO microarchitecture, and the 1GHz OoO processor augmented with two-way and four-way simultaneous multithreading. Both IO and OoO cores can take advantage of the stabilization framework, but the energy per instruction of the stabilized OoO core is less because it runs at a lower frequency to meet the same deadlines.

The MIPS rate stabilization of complex processors using a PID feedback control loop is a general technique applicable to environments in which lower power or energy coupled with steady, predictable performance are desirable, although we target more specifically real-time systems in this article.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-Time and Embedded Systems

General Terms: Processor Architectures, Performance of Systems

Additional Key Words and Phrases: Power, energy, EPI

This work was supported by NSF grant NSF-1219186.

Extension of a Conference Paper: This article is based on a paper published in the proceedings of the International Symposium on Computer Architecture (ISCA '09) and titled "Dynamic MIPS rate stabilization in out-of-order processors [Suh and Dubois 2009]. Besides significant rewrite, we have included new material, specifically (1) results for the stabilized IO processor (Section 6.2); (2) extension of the stabilization framework to OoO SMT processors (Section 7); (3) port of all applications from SimpleScalar to SESC; (4) more MiBench benchmarks compiled by the SESC-supported MIPS compiler for our experiments; and (5) re-evaluation of all previous work in [Suh and Dubois 2009] with the SESC simulator, including controller design, nonstabilized or stabilized OoO processors, nonstabilized IO processors, various cache hierarchies, and power comparisons. The major insight gained by the port to SESC is that the controller design is not affected by the ISA, the compiler, or the simulator.

Authors' addresses: J. Suh, Intel Corporation, Hillsboro, OR 97124; email: jinho.suh@intel.com; C.-T. Huang and M. Dubois, Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-2560; emails: chiehti@usc.edu, dubois@paris.usc.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/04-ART4 \$15.00

DOI: <http://dx.doi.org/10.1145/2714575>

ACM Reference Format:

Jinho Suh, Chieh-Ting Huang, and Michel Dubois. 2015. Dynamic MIPS rate stabilization for complex processors. *ACM Trans. Architec. Code Optim.* 12, 1, Article 4 (April 2015), 25 pages.
DOI: <http://dx.doi.org/10.1145/2714575>

1. INTRODUCTION

In this article, we see a processor or a core as a supplier of quality of service (QoS). The service is to maintain a constant instruction execution rate. Predictable execution rate is useful in real-time systems because it can help reduce power consumption by minimizing processor idle cycles. We illustrate how mega instructions executed per second (MIPS) rate stabilization can work on top of a real-time system flow and demonstrate how to stabilize the performance of out-of-order (OoO) processors and in-order (IO) processors with hardware-controlled cache hierarchies at a predetermined, steady instruction throughput (MIPS) level with a proportional, integral, and differential (PID) gain feedback control loop. Different applications or different phases of the same application may have different amounts of instruction-level parallelism (ILP) or different cache miss rates. To meet the preset target MIPS rate, the frequency is reduced in application phases where ILP is abundant and memory access locality is high, and it is throttled up in phases with low ILP and high cache miss rates using dynamic voltage and frequency scaling (DVFS). DVFS is very effective at regulating power because dynamic power is roughly proportional to the cubic power of frequency [Childers et al. 2000], which is why many processors for mobile applications implement a DVFS scheme. The stabilization of the MIPS rate of a processor is desirable in environments in which low power or energy and steady, predictable performance levels are desirable or required.

One way to achieve higher performance within a given power budget is to exploit parallelism at all levels. Very long instruction word (VLIW) processors are often preferred for real-time applications because their performance comes from ILP rather than from higher clock rates. Moreover, execution times on a VLIW are highly predictable because instruction scheduling is done statically by the compiler. However, as mobile applications become more complex, exploiting ILP statically in VLIWs becomes more difficult.

An alternative to VLIWs is dynamic OoO processors. Unfortunately, the MIPS rate of OoO processors is very hard to predict statically and is highly variable and program- and data dependent, because their performance depends on execution speculation and nonblocking hardware-managed cache hierarchies. Additionally, simultaneous multithreading, in which multiple threads are executed concurrently on a core, introduces even more uncertainty and variability because concurrent threads interfere with each other's executions. This uncertainty and variability makes it hard to estimate execution times accurately with techniques such as worst-case execution time (WCET) analysis, on which software developers of real-time systems typically rely. The average case profile information is applicable to OoO processors, but it cannot optimize power/energy per instruction (EPI), which is important to mobile applications.

To stabilize the MIPS rate of processors, we advocate a control feedback loop designed to keep the MIPS rate constant. We show how to estimate the parameters of the components in the loop. The resulting design of the feedback loop to control the MIPS rate is very robust and effective for various core and cache architectures, as evidenced by the large number of simulation results of various systems reported in this article.

The major contribution of this work is to demonstrate that the performance of complex processors can be stabilized so that their MIPS rate is predictable and constant. As a result, they can be deployed in real-time applications or in environments with execution latency requirements and meet their deadlines with lower power and energy

by minimizing the clock rate to meet deadlines and execution latency requirements. To exploit MIPS rate stabilization, we also propose a new framework to develop real-time applications on IO or OoO processors, which gives the system the power to control the execution rate (MIPS rate) of a processor. We simulate an IO processor and an OoO processor (with and without SMT) with various cache sizes, with and without stabilization. We demonstrate the methodology on a large set of execution samples from the MiBench benchmark suite and evaluate the power and energy consumption of the stabilized designs. By exploiting parallelism instead of frequency to reach a given performance level, stabilized OoO processors turn out to be more power/energy efficient than stabilized IO processors—a nonintuitive result since OoO processors exercise much more hardware to manage the processing of each instruction.

The article is structured as follows. In Section 2, we introduce our proposed framework for real-time application development. Section 3 overviews some prior work related to our topic. In Section 4, we demonstrate that the variability of instruction throughput in OoO processors is such that stabilization is required for real-time systems even with soft deadlines. In Section 5, we introduce the PID controller integrated with a throughput-to-frequency mapper in a control feedback loop to stabilize the MIPS rate of processors by DVFS; then we evaluate the effect of the controller parameter settings on the quality of the stabilization and on power in the context of OoO single-thread processors; finally we show through extensive simulations that the design of the controller works well with various cache hierarchies. In Section 6, the stabilized OoO processor is compared to the stabilized IO processor with the same MIPS rate target; we show that the stabilized OoO processor is more energy efficient than the stabilized IO processor. Section 7 explores the stabilization of the OoO processor running two and four threads in SMT mode; this evaluation demonstrates that the feedback loop with the same PID controller is robust enough to maintain a steady target MIPS rate for various mixes of concurrent threads. The discussion in Section 8 is followed by the conclusion in Section 9.

2. MIPS RATE STABILIZATION-AIDED FRAMEWORK FOR REAL-TIME APPLICATIONS DEVELOPMENT

A fundamental requirement for any real-time system is that the execution rate (MIPS rate) of the processor must meet the deadline of the most demanding task in worst-case situations. This determination can be done by applying WCET analysis or average case profile information. These techniques can ensure that tasks' deadlines are met for a given processor and that the overall task set is schedulable. The problem is that they do not take care of power/energy consumption. If tasks finish before the deadlines, the processor wastes energy in idle cycles. To avoid idle cycles, the processor should run at a lower execution rate (MIPS rate) so that tasks finish right at the deadline. Our MIPS rate stabilization framework can assist the system to achieve this by controlling the execution rate (MIPS rate) of the processor. Once the proper MIPS rate is obtained, our framework delivers predictable and constant MIPS rate at the lowest possible frequency and without wasting energy in idle cycles.

Figure 1 illustrates the framework for developing a real-time application with the assistance of MIPS rate stabilization to achieve lower power/energy consumption. First, a processor capable of supporting the target applications must be selected. This can be done by either WCET analysis or average case profile information. If a mathematically bounded estimated time is needed, WCET analysis is preferred. In the case of OoO (with or without SMT) processors, WCET analysis is too complex [Hergenhan and Rosenstiel 2000; Tan and Mooney 2007], and thus it may be preferable to use average case profile.

With a processor powerful enough to support all applications, some applications may finish close to the deadline and others may finish far earlier than the deadline. To

Fig. 1. Work flow to develop real-time applications.

avoid wasting energy during idle cycles and to reduce dynamic power, it is preferable that the task completes right on time rather than before the deadline. If the required (lowest) MIPS rate meets all deadlines, the MIPS rate can be stabilized at that lowest value and the power and energy are minimized automatically by the control feedback loop. This strategy can be applied to various microarchitectures, including complex OoO processors.

Knowing the number of executed instructions and the deadline, one can determine the target MIPS rate. The number of executed instructions can be obtained by performing WCET analysis or by gathering average case profile information. The highest MIPS rate that the processor must provide for a task is then computed by the worst-case number of instructions to execute the task (I) divided by the WCET (T).

$$HighestMIPS = \frac{I}{T} \quad (1)$$

To make the task finish right on its deadline, the target MIPS rate must be set equal or lower than the highest MIPS rate. It is calculated by the worst-case number of instructions to execute the task (I) divided by the deadline (D) of the task.

$$TargetMIPS = \frac{I}{D} \quad (2)$$

As long as the throughput of a processor is higher than the given target MIPS rate, a task can complete before its deadline. With our MIPS rate stabilization framework, the throughput of a processor remains stable at the given target MIPS rate. After one has chosen a processor powerful enough to sustain the highest MIPS at the normal (highest) frequency, the processor will be capable of producing the required throughput at the given target MIPS rate. The target MIPS rate should be lower than the highest MIPS rate required by any task because the deadline must be longer than the WCET.

Throughout this article, the instruction set is a RISC-like ISA (the MIPS ISA), in which instructions are so simple that an entire instruction can be executed directly by an execution unit. For complex ISAs such as the Intel x86, in which instructions may take multiple RISC-like micro-ops to execute, the micro-op execution rate must be

stabilized instead of the instruction execution rate. The application developer must be aware of the number of micro-ops per complex instruction to estimate the target micro-op rate. This target micro-op rate is then used instead of the MIPS rate to regulate frequency and voltage.

3. RELATED WORK

Applying WCET analysis to OoO processors [Hergenhan and Rosenstiel 2000; Tan and Mooney 2007] is a very hard problem because of all the complex and unpredictable features of OoO processors, such as speculative execution and nonblocking cache hierarchies. Profile information [Kumar et al. 2007] is often used in OoO processors for soft real-time systems. MIPS rate stabilization provides a way to sustain a predictable and steady MIPS rate without knowing the microarchitecture of a processor. In conjunction with any real-time analysis technique, the execution time can be controlled to reduce the number of idle cycles and the operating frequency and save power/energy.

Other studies have tackled the same problem in different ways [Hughes et al. 2001b; Kondo et al. 2007; Rotenberg 2001; Varma et al. 2003]. In Rotenberg [2001], the execution predictability of a single-issue OoO processor is improved by throttling the processor's frequency to meet deadlines. Performance is speculated by slowing down the processor at first and then assessing the slack to the deadline to set the operating frequency to catch up with the remaining work on time. In Hughes et al. [2001b], tasks are at first scheduled optimally after compiler profiling. As in Rotenberg [2001], the framework is based on a measure of slack to the deadline. In Varma et al. [2003], a PID controller manages the DVFS hardware to suppress jitter and improve energy efficiency in the context of a simple IO processor, where MIPS rate variability is much less severe than in OoO processors.

The task scheduler can help real-time systems meet their deadlines. The study in Cazorla et al. [2005] showed that the execution time of a real-time high-priority thread in a multithreaded system is predictable when it is scheduled in conjunction with smart resource management. In Zhu and Mueller [2004], feedback control is also applied to reach an energy-optimal real-time schedule of tasks in the OS scheduler, given that WCET is known a priori. Feedback control tunes the DVFS to let parts of each task run at a lower power level, and task scheduling makes sure that hard deadlines are met. Instead of integrating DVFS with real-time schedulers, we advocate isolating DVFS from the schedulers. The schedulers can treat the processor as a MIPS QoS provider as long as the target MIPS rate is set below the highest MIPS rate. In general, our stabilization framework, which is at the hardware level, can be deployed with software or thread scheduling techniques to improve the overall outcome in various contexts. We will show this in Section 7.3 in the context of multithreaded processors.

Power and energy consumption brings up many issues, such as hotspots, cooling costs, and energy budget. In Skadron et al. [2002], the problem of finding the optimal trade-off between heat and performance is resolved by an RC model-based PID controller such that the processor always runs within the envelope of the power (heat) budget while maximizing performance as much as possible. Since the mathematical RC model is well defined and the effect of heat dissipation on temperature is a rather slow process, PID control works very well in this context. In GALS [Iyer and Marculescu 2002; Kondo et al. 2007] and MCD [Dropsho et al. 2004], a microarchitecture is partitioned into several, independent domains, and energy efficiency is improved by adapting the frequency and voltage of each domain separately inside the processor. Currently, our approach is limited to the control of the clock and Vdd in the entire processor, but its extension to architectures partitioned into independent clocking domains to further optimize the performance/power/energy trade-offs should be explored in future work.

Table I. Architectural Parameters for the Baseline OoO Processor Model Clocked at 1GHz

Parameter	Configuration
Machine width	3-wide fetch, 3-wide issue, 3-wide commit
Window size	128 entry ROB, 48 entry INT issue queue, 24 entry FP issue queue, 48 entry load queue, 32 entry store queue
L1 I-cache	16 KB, 4-way set-associative, 32 byte line, 2 cycles
L1 D-cache	16 KB, 4-way set-associative, 32 byte line, 2 cycles
L2 cache	512 KB, 8-way set-associative, 32 byte line, 8 cycles
BTB/Predictor	2,048 entry, 2-way set-associative, 16K 2 level
Functional units (latency)	2 INT ALU (1); 1 INT MULT (1); 1 INT DIV (1) 1 FP ALU (2); 1 FP MULT (5); FP DIV (32)
Memory	88-cycle latency
TLB	64 entry, 4-way set-associative

The studies in Childers et al. [2000] and Hughes et al. [2001a] were starting points for our research. In Childers et al. [2000], energy is saved when the processor targets a certain throughput with DVFS. We address the same problem in this article, but with a different approach to achieve throughput stabilization in the context of real-time systems. In Hughes et al. [2001a], the high performance variability of OoO processors is explored, and one of the conclusions is that IPC is practically independent of on-chip frequency because caches running at the same frequency as the processor hide most of the impact of off-chip memory accesses that could change IPC—a key observation that motivated our interest in adaptive techniques to suppress the high performance variability of modern processors. If IPC is practically independent of frequency, then frequency can effectively control the MIPS rate.

4. PERFORMANCE VARIABILITY IN OoO SINGLE-THREAD PROCESSORS

4.1. Baseline Processor Model

Our baseline processor is an OoO single-thread processor without DVFS. We have modified the SuperEScalar (SESC) simulator [Ortego and Sack 2004] to model it. SESC simulates the MIPS instruction set architecture. Power is estimated with McPAT 1.0 [Li et al. 2009]. SESC and McPAT are separate software tools; thus, we snapshot the runtime statistics in SESC every time the operation frequency is changed. Offline, McPAT estimates the power for each execution snapshot, and overall power is obtained by combining all of the snapshots. We estimate the power of processor cores including L1 caches. The microarchitecture is Intel P6-like (with a reorder buffer keeping speculative register values); it fetches, dispatches, and issues up to three instructions per clock. The power model is consistent with a 90nm technology. Vdd values for different operating frequencies are adopted from a technical report on Intel's 90nm logic technology for the Pentium 4 processor [Mistry et al. 2004]. The minimum Vdd is selected for each operating frequency to reach the best power efficiency possible (see Table III). Table I gives the major architectural parameters for the baseline processor.

4.2. Benchmarks

To explore the MIPS rate variability of the baseline processor, we have run a large number of execution samples from the MiBench benchmark suite. MiBench is a set of programs representative of real-time embedded applications [Guthaus et al. 2001]. Among them, we have selected 18 programs, a mix of integer and floating-point intensive programs. To have execution samples with a rich mix of behavior, we sliced the dynamic execution of the 18 programs into 315 execution samples of 40 million consecutive instructions each, which form the set of tasks to execute within a time deadline in our evaluations. The task size was chosen based on a study in Xu et al. [2005] showing

Table II. Programs Chosen from MiBench

Category	Programs	Input
automotive	basicmath/50 (bm), bitcount/18 (bc), qsort/9 (qs), susan:edges/1 (se), susan:smoothing/10 (ss)	ref: large data
network	dijkstra/7 (di), patricia/12 (pa)	
security	blowfish:encode/26 (be), blowfish:decode/26 (bd), rijndael:encode/8 (re), rijndael:decode/9 (rd)	
telecomm	adpcm:rawaudio/15 (ac), adpcm:rawaudio/12 (ad), crc32/45 (crc), fft:forward/8 (ft), fft:inverse/4 (ift), gsm:toast/38 (gt), gsm:untoast/17 (gut)

Fig. 2. Dynamic MIPS rate (a) and average MIPS rate distribution (b) of all 315 tasks on the 1GHz baseline processor.

that the processing of a MPEG-4 frame requires 30 to 42 million instructions. The total number of instructions in all of the tasks is more than 1.2 billion. Table II lists the 18 programs selected from the MiBench suite. The number in the table after each program is the number of tasks obtained from each program.

4.3. MIPS Rate Variability in the Baseline Processor

Figure 2(a) plots the dynamic MIPS rates for all 315 tasks run on the baseline OoO processor clocked at 1GHz as a function of the number of retired instructions. The dynamic MIPS rate is computed as the total number of retired instruction (in millions of instructions) divided by the time (in seconds) since the beginning of each task execution. The dynamic MIPS rate varies widely across tasks and sometimes within the same task.

Figure 2(b) shows the distribution of the average MIPS rates of all 315 tasks run on the 1GHz baseline processor. The average MIPS rate of a task is computed as 40 divided by the execution time of the task (in seconds). This distribution is very wide. Two tasks have an average MIPS rate of 390 MIPS. About 5% of the tasks have an average MIPS rate of about 579 MIPS. The rest of the tasks have an average MIPS rate between 700 MIPS and 1.3 GIPS. The large standard deviation of the average MIPS rates (194.13 MIPS) indicates wide variations around a mean average MIPS rate of 1.05 GIPS. Variable memory access latencies, program dependencies, speculative execution, and variable ILP all contribute to this high variability. In the next section, we describe our approach to stabilizing the performance of the baseline processor.

5. MIPS RATE STABILIZATION IN OoO SINGLE-THREAD PROCESSORS

In our stabilization framework, we assume a conservative off-chip DVFS controller with a typical switching frequency less than 5MHz. We adapt the clock rate and voltage on-chip only and keep the main memory at the same frequency at all times.

Fig. 3. Generic closed-loop feedback controlled system.

5.1. Control Framework

Among all feedback control schemes, the PID gain feedback controller [Franklin et al. 1986] illustrated in Figure 3 is widely adopted. The input is the reference signal. The control loop forces the output of the plant (which is the system to control) to track the reference signal. The error signal (the difference between output and reference) is relayed by the feedback loop and is continuously applied to a controller whose function is to bring the error down to zero. Mathematically, the PID controller is characterized by the following equation:

$$U = K_P E + K_I \int E dt + K_D \frac{dE}{dt}, \quad (3)$$

where K_P , K_I , and K_D are the proportional gain, the integral gain, and the differential gain, respectively, and U is the controlled input applied to the plant. Feedback corrections are continuously applied until the error reaches zero; thus, ideally, the output faithfully tracks the reference input after some delay. Equation (4) shows the discrete model for the PID controller obtained after a bilinear transformation of Equation (3).

$$\begin{aligned} U_n &= U_{n-1} + \Delta U_n \\ \Delta U_n &= K_P(E_n - E_{n-1}) + K_I E_n \\ &\quad + K_D\{(E_n - E_{n-1}) - (E_{n-1} - E_{n-2})\} \end{aligned} \quad (4)$$

When the plant is mathematically well defined, the PID controller can be formally designed with the aid of control theory. When the mathematical model of the plant is unknown, the parameters of the PID controller must be fine-tuned empirically. Unfortunately, the characteristic of an OoO processor is neither mathematically well defined nor constant. The plant model changes constantly and depends on the characteristics of program phase execution, such as dependencies, memory behavior, execution speculation, and resource conflicts. In fact, in our framework, the input signal to the system (the reference) is fixed but the plant characteristic keeps changing. This is very different from the traditional control system design problem.

The values of the three PID gains directly affect the performance of the controller. If the controller reacts too fast to the error, it may amplify the error even if the error is very small; thus, the system moves erratically and may not settle down. If the controller reacts too slowly, it cannot track the reference input fast enough.

Typically, the controller is tuned for the step input. In the case of a step input, the control system is designed to output the same step function as faithfully as possible. The characteristics of system output under a step input, such as overshoot, rising time, settling time, and steady-state error, are observed to help tune the controller properly. We adopt this approach in this article. The processor starts “cold” at the beginning of each task, and we observe the rapidity of convergence toward the target MIPS rate.

The description of the PID controller so far assumes that all control variables are in the same domain. However, the variable to control is the MIPS rate, but the variable affecting the control is the clock rate. To convert MIPS rate error into frequency differential, a throughput-to-frequency mapper must be included in the feedback loop.

Fig. 4. Controller with throughput-to-frequency mapper.

Fig. 5. Configuration of the discrete frequency mapper.

Figure 4 shows the system with the mapper added. Each frequency change causes a resynchronization penalty charged to the performance of the plant. To compute the next frequency, the mapper predicts that the processor will have the same behavior in the next phase as in the current phase and that the IPC does not change with the frequency. Thus, it computes the next on-chip operating frequency as such:

$$Next_Freq = Current_Freq \times \frac{Target_MIPS}{Current_MIPS} \quad (5)$$

In a DVFS scheme, the number of possible frequencies is limited. Thus, the mapper must map the next operating frequency to one of the operating frequencies provided by the DVFS scheme. Figure 5 shows the mapping of frequencies in our DVFS scheme. Each discrete DVFS frequency is mapped to a range of target frequencies. The frequency of the DVFS scheme is not necessarily aligned at the center of its associated range of target frequencies. By slanting the DVFS frequency toward the higher end of its associated frequency range (as shown in Figure 5), a given target frequency tends to map to a higher DVFS frequency, which improves the response time because the system response is faster when the MIPS rate falls below its target and is slower when the MIPS rate is above its target.

5.2. Controller Design

In this section, we show how we arrived at the final controller design for the baseline processor using a reduced set of 16 carefully selected tasks called the *training set*. The reason for the training set is that the exploration of the design space for the PID controller over all 315 tasks is not practically feasible. To shrink the search space for the controller design, the most representative 16 tasks are selected using SimPoint [Hamerly et al. 2005] from the five programs (automotive:bitcount, automotive:qsort, telcomm:fft-forward, telcomm:fft-inverse, and network:dijkstra) with the highest performance variability among the MiBench programs shown in Table II. The first step is to explore the effects of the controller parameters on the MIPS rate stabilization given a DVFS framework.

5.2.1. DVFS Framework. The four voltage and frequency pairs in our DVFS framework are given in Table III. These values are drawn from Intel data [Mistry et al. 2004]. At each voltage/frequency transition, the processor pauses for 20 microseconds due to PLL

Table III. DVFS Frequency and Voltage Pairs

Frequency (MHz)	Vdd (V)
1,000	0.825
800	0.772
500	0.694
300	0.641

resynchronization [Dropsho et al. 2004]. In Dropsho et al. [2004], the DVFS transition speed was measured to be 10mV/microsecond. In our DVFS configurations, the voltage changes by 200mV steps at a time, hence the 20-microsecond pause at each transition. Furthermore, we assume that the energy consumption due to each voltage/frequency transition is negligible based on observations made in Shen [2006] and Stanley-Marbell et al. [2003]. Similar assumptions were made in other DVFS studies [Eeckhout et al. 2003; Dropsho et al. 2004; Hughes et al. 2001b; Kondo et al. 2007; Rotenberg 2001; Skadron et al. 2002; Varma et al. 2003; Zhu and Mueller 2004].

The execution of a task is divided into nonoverlapping control windows. During each control window, the controller measures the core throughput and determines the operating point for the next control window. In this article, the control window is set to 50K retired instructions. Whenever a control window ends after 50K retired instructions, the error communicated to the PID controller is the difference between the reference throughput and the average instruction throughput during the window. The controller then determines the next target throughput, and the throughput-to-frequency mapper maps it to the next operating frequency and voltage.

The size of the control window is a design trade-off. When it is small (such as our window of 50K instructions), the controller reacts faster. The potential performance cost is more resynchronization pauses. Each task of 40M retired instructions contains 800 control windows. Our final controller design causes an average of 140 resynchronizations per task—that is, an average of $3.5\mu\text{s}$ of resynchronization stall per control window. Each control window in the baseline running at maximum frequency with no stabilization takes an average of $47\mu\text{s}$. Thus, the total resynchronization overhead is at most 8% of the execution time of the baseline running at maximum frequency. It is much less under stabilization given the lower average frequency.

The most important problem of PLL resynchronization is that the processor is stalled during the resynchronization and cannot react to interrupts. This may cause a real-time interrupt loss if the interrupt arrives during the resynchronization period. Intel's XScale-style resynchronization can avoid this problem [Clark 2001], as it does not stall the processor while the frequency and voltage change.

5.2.2. Effects of Control Parameters on Stabilization. If the controller responds faster and often changes the frequency, the processor throughput is more stable; however, every time the frequency and voltage change, the processor is stalled for PLL resynchronization, which lowers the maximum stable MIPS rate attainable by the stabilized processor. On the other hand, if the controller is too slow, the system takes longer to converge and may deviate from the target throughput. The parameter settings that we have explored for the controller are shown in Table IV. As the mathematical model of the plant is unknown, it is safer to keep K_D small to prevent unexpected divergence, because the differential gain parameter amplifies any fast phase jitter.

To compare the performance of the six controllers, we have modified the SESC simulator to include the DVFS scheme, the throughput-to-frequency mapper, and the PID controller. The stabilized OoO processor is exactly the same as the baseline OoO processor, except that it is equipped with DVFS and PID control. We simulate eight target MIPS rates from 300 MIPS to 1,000 MIPS for the 16 tasks in the training set.

Table IV. Six Different Parameter Settings for Controller Gains

	K_P	K_I	K_D
Setting 1: Slowest	1	10	0.1
Setting 2	10	50	0.1
Setting 3	50	50	0.1
Setting 4	75	50	0.1
Setting 5	100	50	0.1
Setting 6: Fastest	100	100	0.1

Fig. 6. Dynamic MIPS rate for the training set with MIPS targets of 300, 400, 500, 600, 700, 800, 900, and 1,000 MIPS (stabilized OoO single-thread processor).

Figure 6 shows the dynamic MIPS rate of the stabilized OoO processor for the six controller parameter settings with target MIPS rates of 500, 600, 700, and 800 MIPS (the four columns in the box). We observe that the throughput of the stabilized OoO processor converges rapidly to its target MIPS rate, except for the slowest controller with parameter setting 1. Settings 2 through 6 all stabilize the MIPS rate for all tasks in the training set. We also ran simulations of the stabilized processor with MIPS rate targets of 300, 400, 900, and 1,000 MIPS for the tasks in the training set shown in Figure 6. We observe that in these cases, the throughput of several tasks does not converge to the target MIPS rate. With a target of 300 or 400 MIPS, the stabilized processor keeps running at the minimum frequency of 300MHz all the time for some tasks, but because of their high ILP, the processor exceeds the 300 MIPS target. When the target is 900 or 1,000 MIPS, some tasks with very low ILP cannot reach the target MIPS rate, although the processor continuously runs at its maximum frequency of 1GHz. Clearly, the stabilized processor cannot consistently sustain target processor throughputs of 300, 400, 900, or 1,000 MIPS across all tasks in the training set.

It is remarkable and very encouraging that the controller design is valid for a wide range of parameter values across a wide range of target MIPS rates and a wide range of tasks. This shows that the design of the controller is robust and the same design can be reused in different environments without repeated empirical tuning. We will see

Fig. 7. Power and EPI relative to the baseline processor (training set).

more evidence of the robustness of the controller design when we later evaluate the sensitivity of the stabilization to core architectures and cache sizes.

5.2.3. Effects of Control Parameters on Power and EPI. A major advantage of throughput stabilization, besides the predictability of program execution time, is the reduction of power and energy. Figure 7(a) shows the power and Figure 7(b) shows the EPI (energy per retired instruction) for the six controller settings under the eight target MIPS rates relative to the baseline processor. For higher target MIPS rates, power and EPI are higher because the processor operates at higher frequencies. Faster controllers (closer to setting 6) trigger more frequency/voltage changes because of faster responses to throughput changes and because the processor needs to operate at higher frequencies to compensate for the overhead of DVFS transitions. Therefore, a faster controller consumes more power and has higher EPI than a slower controller.

5.3. Stabilization Results for All Tasks

Based on our evaluations on the training set, parameter settings 2 through 6 are equally acceptable. In this section, we show detailed simulation results when the operating point of the controller is set to $(K_P, K_I, K_D) = (75, 50, 0.1)$ —setting 4 in Table IV.

First we need to select a target MIPS rate. Our original workload of 315 tasks is made of all 40 million instructions execution slices of 18 programs a priori selected from the MiBench suite. This selection has resulted in a population of tasks with wide execution time variability, from 390 MIPS all the way to 1.3 GIPS. Although 95% of the average MIPS rates are between 700 MIPS and 1.3 GIPS, the target MIPS rate must be at least equal to the minimum MIPS rate of all tasks shown in Figure 2, which is 390 MIPS, even though only 2 out of 355 tasks cannot run faster than 390 MIPS. As shown in Figure 6, some high-ILP tasks do not converge to the low target MIPS rates of 300 or 400 MIPS even when the stabilized processor keeps running at the minimum frequency of 300MHz all the time. Therefore, they complete early and do not maximize energy/power savings. To widen the range of target MIPS rates, we have removed the two outlying tasks that run at 390 MIPS and 396 MIPS on the baseline (two tasks from automotive:bitcount out of the 315 preselected tasks) from the workload. Without these two tasks, the maximum target MIPS rate becomes 579 MIPS, and we can set the target MIPS rate anywhere from 1 to 579 MIPS. All evaluations done in the balance of this section are for the 313 remaining tasks. A target throughput of 550 MIPS is selected. The next “sweet spot” for the target MIPS rate would be 706 MIPS, provided that 5% of the tasks are removed from the workload, in which case the results on power and energy would be much better.

The effectiveness of the PID controller is best demonstrated by comparing Figures 2 and 8. In Figure 2, the dynamic MIPS rates are all over the map. By contrast, in Figure 8, the dynamic MIPS rates of most tasks settle at the target throughput of

Fig. 8. Dynamic MIPS rate (a) and average MIPS rate distribution (b) of all 313 tasks on the stabilized OoO processor.

550 MIPS within the first 5 million committed instructions. At the end of every task, after 40 million instructions, the MIPS rates of all tasks settle within a ± 3 MIPS band around the target throughput of 550 MIPS, as illustrated in Figure 8(a). Figure 8(b) shows that the distribution of average MIPS rates across all 313 tasks is very narrow. Overall, the average frequency of the stabilized processor across all tasks is 603MHz.

Misses per instruction (MPI) and IPC are practically identical in the baseline and in the stabilized processors. This is because the number of cache misses is small and the possible IPC gains due to smaller L2 miss penalties (in cycles) when the processor throttles down to lower operating frequencies are less than 1%. As pointed out previously in this article and in Hughes et al. [2001a], the IPC of an OoO processor is practically independent of its operating frequency, which is why the stabilization works so well.

Figure 9 shows the fraction of tasks whose dynamic MIPS rate falls within a margin of the target MIPS rate (550 MIPS) as a function of the number of retired instructions during execution. Within a margin of 0.5% of the target MIPS rate, the dynamic MIPS rate of all tasks converges to the target throughput after the first 5 million instructions. To stabilize the MIPS rate as fast as possible, we suggest targeting at least a 0.5% marginal offset from the target MIPS rate. For example, one should set the target MIPS rate to 552.8MIPS instead of 550MIPS so that the task reaches the 550MIPS target faster.

One major contribution to EPI in the baseline processor is the power consumed after the completion of a task and until its deadline expires. During the time between task completion and its deadline, the processor can be in either IDLE or SLEEP mode. During IDLE time, both static and dynamic powers are consumed, and the processor operates at its lowest frequency (300MHz). In Contreras et al. [2007], it is shown that the idle power consumption of the XScale PXA255 microcontroller is about 29.6% of the average CPU power consumption. To avoid wasting energy in IDLE mode, the baseline processor can be put in SLEEP mode. Unfortunately, in some processors, waking up the processor from SLEEP mode requires long transition times (e.g., 160ms for StrongARM SA-1100), thus making SLEEP mode impractical in the context of high-performance real-time applications [Benini et al. 2000]. However, in more recent processors, the wakeup latencies are much smaller as compared to XScale PXA225. Current processors reduce power by providing several sleep levels with different wakeup latencies. The Intel Atom processor Z5xx series [Intel 2009] has six operation states (C0, C1, C1E, C2, C4, and C6). The state with the higher number has lower power consumption and longer wakeup latency. For example, in the C4 state, the processor takes about 35us to wake up and enter C2 state. If a processor is in the C6 state, the wakeup latency to enter the C4 state is increased to 100us. In the C6 state, Intel enhanced deepest sleep

Fig. 9. Fraction of tasks falling within a margin of the target as a function of time.

Fig. 10. MIPS rate, power, and EPI of the stabilized OoO processor relative to the baseline processor.

mode, the PLL is turned off and the L2 cache is gated off. All architectural states are saved in an on-die SRAM, and the core voltage is dropped to the lowest core voltage possible (0.3v).

Figure 10 shows (from left to right) the MIPS rate, power, EPI-to-task-completion, EPI-to-deadline (IDLE), and EPI-to-deadline (SLEEP) relative to the baseline processor. Each bar in the figure represents an average over all tasks in a benchmark program. The deadline is 72.7ms and is determined by the target MIPS (550MIPS) and the size of the task (40 million instructions). The baseline processor completes every task early in most cases, whereas the stabilized processor completes its tasks right on time to meet the deadline. EPI-to-task-completion measures the energy while executing tasks. EPI-to-deadline includes the power consumed by the processor after the task is completed and before the deadline. If the processor is put in IDLE mode (all activities in the processor are stopped, and the voltage is set at 0.641v), the average MIPS rate of the stabilized OoO single-thread processor is 53.8% of the baseline, whereas its average power and EPI-to-deadline are 59.9% and 87.0% of the baseline, respectively, as displayed in Figure 10. The last column in Figure 10 shows the EPI-to-deadline when the processor is put in SLEEP mode (all activities in the processor are stopped, and the voltage is set at 0.3v) after the task is completed. The results show that our stabilization framework can still save 5% of energy as compared to the baseline using SLEEP mode. Note that because of a lack of information about the Intel Atom processor, we did not include the power consumed by the on-die SRAM. In addition, we did not include the impact of cache misses after the L2 cache has been turned off and the processor resumes normal operation.

Although the stabilized OoO processor saves dynamic power by operating at lower frequency and voltage levels, the time to execute a task takes longer. Therefore, the static energy increases. On average, the EPI-to-task-completion is 103.6% of the baseline, because in some current processors the wakeup latencies of lower power modes are much improved, whether slowing down the processors with DVFS is better than finishing tasks earlier and putting the processor into SLEEP mode is an open question [Le Sueur and Heiser 2011]. Either way, our stabilization framework provides a way to make the performance of superscalar processors reliably predictable. In some situations, a lower MIPS rate can be targeted so that tasks finish right before the deadline. In other situations, the target MIPS rate can be raised to finish tasks early and put the processor into SLEEP mode.

Table V. IPC and Cache MPKI in Three Different Cache Hierarchies on the Stabilized OoO Processor

		qsort	patricia	average
IPC	16K-16K-512K	1.2328	0.9663	1.1000
	8K-8K-256K	1.2130	0.9115	1.0623
	4K-4K-128K	1.0673	0.8668	0.9671
IL1 MPKI	16K-16K-512K	0.0083	54.4868	27.2476
	8K-8K-256K	1.6944	82.3929	42.0437
	4K-4K-128K	56.0044	93.4985	74.7515
DL1 MPKI	16K-16K-512K	3.3801	0.7516	2.0658
	8K-8K-256K	3.8790	2.7491	3.3141
	4K-4K-128K	5.2766	15.1513	10.2140
L2 MPKI	16K-16K-512K	1.3946	0.4404	0.9175
	8K-8K-256K	1.7785	0.4421	1.1103
	4K-4K-128K	2.6253	0.7015	1.6634

Fig. 11. Dynamic MIPS rate ((a) and (c)) and average MIPS rate ((b) and (d)) distribution with smaller caches on the stabilized OoO processor.

5.4. Sensitivity to Cache Size

Cache effects are a major cause of execution time variability. Caches in real-time systems usually are software controlled or even disabled to simplify WCET analysis. This approach has the undesirable side effect of reducing performance. Cache hierarchies affect IPCs and MIPS rates in ways that are hard to predict, but their effects can be stabilized with the same framework developed in this article.

To evaluate the sensitivity of the controller design to cache sizes, we ran simulations of our stabilized OoO processor with cache sizes reduced by factors of two and four. For this evaluation, we selected the two benchmarks (basicmath:qsort and network:patricia) from Table II that have the largest number of cache misses per kilo instructions (MPKI). The average IPC and cache MPKI for all tasks in the two benchmarks are shown in Table V. The average number of MPKI in the L1 instruction cache (IL1), the L1 data cache (DL1), and the L2 cache increases three-, five-, and twofold when the cache sizes are reduced by a factor of four. The IPC of the baseline OoO processor reduces by 3.05% and 12.1% when the cache sizes are reduced by 50% and 75%, respectively.

The average MIPS rates of the tasks in the two benchmarks are 1,254 and 966 MIPS in the baseline. When the cache sizes are reduced by 50%, the average MIPS rates of the tasks in the two benchmarks are 1,236 and 911 MIPS. The average MIPS rates become 1,078 and 867 MIPS when the cache sizes are reduced by 75%. Even with reduced IPC, the MIPS rates of the two benchmarks are still high enough to sustain the target MIPS rate of 550 MIPS in all tasks. Figure 11 shows the stabilized MIPS rates for the OoO processors with reduced cache sizes. With the same controller parameter settings, the variances of the average MIPS rates of the stabilized processors are similar to the

Fig. 12. Dynamic MIPS rate (a) and average MIPS rate distribution (b) for the IO processor running at 1.32Hz.

Fig. 13. Dynamic MIPS rate (a) and average MIPS rate distribution (b) for the stabilized IO processor.

variance obtained previously for the stabilized processor with original cache sizes. The MIPS rate still converges rapidly to its target within 5 million instructions of each task. This is further evidence that the design of the controller is robust, as it works well across different cache sizes even though cache sizes change IPCs.

6. COMPARISON WITH A SINGLE ISSUE IO PROCESSOR

6.1. Basic IO Processor

To achieve the same real-time throughput as the 1GHz OoO processor, a StrongARM SA1110-like single issue IO processor with the same cache hierarchy as our baseline OoO processor has to run at the nominal frequency of 1.32GHz. This operating frequency is selected such that the task with the lowest performance on the IO processor has the same throughput as the task with the lowest performance on the OoO processor. We simulated the same 313 tasks as for the baseline. Figure 12 shows the MIPS rates of a basic (nonstabilized) IO processor capable of maintaining a minimum throughput of 550 MIPS on all tasks. Most tasks execute at a throughput between 550 and 700 MIPS. Although the IO processor runs at a higher frequency, it consumes less power than the baseline OoO processor because its architecture is simpler. However, the OoO processor has better EPI. For the same technology, the average power and EPI-to-task-completion of the IO processor are 87.6% and 122.0% of those of the baseline OoO processor.

6.2. Stabilized IO Processor

The IO processor can be stabilized at a target of 550 MIPS with the same parameter settings as in Section 5 (setting 4 in Table IV), resulting in an average operating frequency of 1.132GHz. Figure 13 shows the MIPS rate of the stabilized IO processor. The throughputs of all tasks are within a margin of ± 3 MIPS of the target throughput. These results demonstrate further that the design of the PID controller is robust, as it works well across different types of processor architectures.

The detailed power and energy comparisons between stabilized OoO, nonstabilized IO, and stabilized IO processors are shown in Figures 14, 15, 16, and 17. EPI-to-task-completion of the stabilized IO processor is slightly higher as compared to the basic IO processor running at nominal 1.32GHz by 1.7% but saves power, EPI-to-deadline (IDLE), and EPI-to-deadline (SLEEP) by 17.3%, 4.1%, and 1.3% respectively. However, the stabilized OoO processor has the best savings for all metrics. It saves 12.4%, 12.7%, 12.2%, and 12.5%, respectively, as compared to the stabilized IO processor while having the same performance throughput.

Fig. 14. Comparison of power (baseline OoO = 100%).

Fig. 15. Comparison of EPI-to-task-completion (baseline OoO = 100%).

Fig. 16. Comparison of EPI-to-deadline (IDLE) (baseline OoO = 100%).

Fig. 17. Comparison of EPI-to-deadline (SLEEP) (baseline OoO = 100%).

These results demonstrate that for a given MIPS rate, a complex OoO processor can have better power/energy characteristics than a simple IO processor, even though the OoO processor exercises more hardware to manage the execution of each instruction. The reason is that ILP is traded off for higher operating frequencies, and the relation between power and frequency is superlinear.

7. OoO SMT PROCESSOR

In SMT cores, instructions from different instruction streams are executed concurrently. The resources are shared and contended for by the concurrent threads, which makes it more difficult to predict the execution time of each thread or set of threads. Execution times can be predicted by WCET analysis in some cases, as was done for an IO SMT processor [Mische et al. 2008, 2010] derived from an IO single-thread superscalar processor. However, it is practically impossible to apply WCET analysis to OoO SMT processors. On the other hand, the average case profile information may apply to OoO SMT processors, but it does not provide the real-time system a way to control the throughput of processors that results in saving power/energy. Our stabilization framework based on a PID MIPS rate controller provides a reliable approach to stabilizing the throughput of an OoO SMT processor. It shows the robustness of the controller that can be applied to various complex processors.

Table VI. Workloads Chosen for the SMT Processor

2 Threads			4 Threads		
se / re	be / gut	bm / ss	bm / bd / bc / be	bm / di / ac / ad	bm / ss / bc / be
bm / bd	bc / be	be / gt	bc / be / bc / be	bc / crc / ac / ad	bc / ss / bc / ss
di / gut	bc / crc	ad / ift	be / gt / pa / ift	be / gut / bm / gt	be / gut / bc / ss
bm / di	bm / gt	qs / ft	di / gut / bm / di	di / gut / bm / gt	pa / ift / ad / ift
pa / ift	ac / ad	bc / ss	ac / ad / ac / ad	ad / ift / ad / ift	se / ri / qs / fft

To evaluate the stabilization of the OoO SMT processor, we use SESC [Ortego and Sack 2004] to simulate two-thread and four-thread OoO processors with the same architecture parameters as in Table I. Resources such as functional units and caches are shared by all threads. The simulator implements a round-robin (RR) SMT fetch policy. In every cycle, the processor fetches at most three instructions by first fetching as many instructions as possible from the first thread. If the first thread cannot fill all instruction slots, the processor fetches instructions from the next thread. SESC can simulate SMT processors, but it only supports multithreaded programs. Because the MiBench benchmarks are all single-threaded programs, we picked sets of two and four benchmarks from the MiBench suite to form two-thread and four-thread workloads and modified the core of SESC to support simulations of multiprogrammed workloads.

7.1. Workload Selection

It is not possible to run all combinations of benchmark programs in multithreaded machine evaluations. To obtain acceptable simulation times, we start by choosing one representative sample from each benchmark program using SimPoint [Sherwood et al. 2002]. We select every possible pair of threads and add workloads with two copies of the same thread. The number of resulting workloads is $C_2^{18} + 18 = 171$. We run each of the 171 two-thread workloads until both threads have completed at least 40 million instructions. Note that now one workload executes at least 80 million instructions. We then select a representative subset of two-thread workloads by workload characterization methods [Eeckhout et al. 2003]. At first, a set of 15 metrics is extracted from each workload simulation: average committed IPC, ROB occupancy, load queue occupancy, store queue occupancy, instruction fetch rate, integer ALU instruction issue rate, integer MULT issue rate, floating point instruction issue rate, load/store issue rate, L1 instruction cache miss rate, L1 data cache miss rate, L2 cache miss rate, branch predictor accuracy, RAS prediction accuracy, and the number of instructions per branch. We then normalize every metric such that their mean is zero and their variance is one.

Since the correlation between metrics may bias workload selection, we identify the first 10 principal components obtained after principal components analysis [Eeckhout et al. 2003] using SAS [SAS Institute 1990]. Every workload is then projected onto the 10-dimensional space, and the workloads are clustered in that space. For each cluster, we select the most representative workload as the one whose Euclidean distance to the centroid of the cluster is smallest. The 15 selected two-thread workloads are listed in the left column of Table VI.

The four-thread workloads are made of all possible pairs of the two-thread workloads including two copies of the same (a total of $C_2^{15} + 15 = 120$). We follow the same procedure as for the two-thread workloads to obtain 15 representative four-thread workloads. The right column of Table VI lists the selected four-thread workloads. When evaluating the SMT processor, each thread is executed for at least 40 million instructions. Due to the characteristic of each program and resource interferences between threads, the throughput of each thread might be different. To capture interferences, if a thread finishes earlier than other threads, the SMT processor will keep executing the early thread until all threads have executed 40 million instructions.

Fig. 18. Dynamic MIPS rates (a) and average MIPS rate distribution (b) for the two-thread OoO processor.

Fig. 19. Dynamic MIPS rates (a) and average MIPS rate distribution (b) for the stabilized two-thread OoO processor.

7.2. Stabilized OoO SMT Processor

In SMT processors, multiple threads share processor resources in a fine-grain fashion to boost processor utilization. Relative to the average throughput of the baseline single-thread OoO processor, the average throughputs of the basic (nonstabilized) two-thread and four-thread OoO processors increase by 12.38% and 47.49% for the selected workloads. Due to resources sharing between threads, some workloads experience severe high number of branch miss prediction. For example, in the two-thread OoO processor, the rate of correct predictions of qsort/fft drops to 68.6%, whereas the correct prediction rates are 97.8% and 96.4% for qsort and fft running separately. Consequently, the average MIPS rate of qsort/fft becomes 1,076 MIPS, which is lower than when running each task separately (1,317 MIPS for qsort and 1,168 MIPS for fft). In the four-thread OoO processor, the branch misprediction rate does not increase any further and resource utilization is increased, which compensate for poor branch predictions.

We can stabilize the overall MIPS rate (combined MIPS rate of all threads) of the SMT processor with the same PID control scheme (including the same parameter values) as for the IO and OoO single-thread processors of Sections 5 and 6.

Figure 18 shows the MIPS rates of the 15 workloads selected for the two-thread OoO processor. We have verified that the execution times of the 15 selected two-thread workloads have similar statistical characteristics as the set of 171 two-thread workloads: the relative differences between the means and between the standard deviations are 3.67% and 0.98%, respectively.

The target MIPS rate of the baseline OoO single-thread processor was set to 550 MIPS after discarding two outlying tasks. The throughput of 5% of the remaining tasks (at 570 to 580 MIPS) lags behind the throughput of most tasks (95% at 700 MIPS to 1.3 GIPS). Once tasks are paired together, their combined throughput tends to average out, and they have different MIPS rate characteristics. All two-thread workloads fall between 800 MIPS and 1.5 GIPS in the two-thread OoO processor, as shown in Figure 18(b). Figure 19 shows the dynamic MIPS rate of the two-thread OoO processor stabilized for a target MIPS rate of 800 MIPS. The dynamic MIPS rate converges to the target 800 MIPS rate rapidly, before the first 10 million instructions, which is consistent with the single-thread case. The stabilized MIPS rate of every selected workload falls within a ± 3 MIPS band of the target.

We also applied the PID framework to the four-thread OoO processor for which the execution time is harder to predict. Each thread executes for at least 40 million instructions. The resulting workloads now have at least 160 million instructions. We simulated the 15 four-thread workloads listed in Table VI. Figure 20(a) shows their dynamic MIPS rates. The selected 15 four-thread workloads are representative of the

Fig. 20. Dynamic MIPS rates (a) and average MIPS rate distribution (b) for the four-thread OoO processor.

Fig. 21. Dynamic MIPS rates (a) and average MIPS rate distribution (b) for the stabilized four-thread OoO processor.

full set of 120 four-thread workloads, as the relative differences of the average MIPS rates and of the standard deviations are 0.16% and 11.42%, respectively, as compared to the entire set of four-thread samples.

Because of its high throughput and mix of workloads, we can boost the target MIPS rate of the four-thread OoO processor to 1 GIPS to support the most demanding task (see Figure 20(b)). With the PID controller, the throughputs of all workloads are stabilized quickly within the first 10 million instructions, and the stabilized MIPS rates fall within a band of ± 5 MIPS of the target rate, as shown in Figure 21.

Figure 22 displays the MIPS rate, power, EPI-to-task-completion, EPI-to-deadline (IDLE), and EPI-to-deadline (SLEEP) of the stabilized two-thread OoO processor relative to the basic (nonstabilized) two-thread OoO processor. Due to interferences between the two threads, the tasks may take different times to complete. In this context, the EPI-to-task-completion is the time to completion of the slowest task. The deadline is computed as the total number of instruction of both threads (80 million instructions) divided by 800 MIPS. The average MIPS rate of all workloads on the stabilized processor is 69.7% of the MIPS rate of the nonstabilized processor continuously operating at the highest frequency (i.e., 1GHz). The power saving versus the nonstabilized processor is 27.7%. The EPI-to-task-completion is only reduced by 1.7% because of the energy consumed by static power due to longer execution times. The EPI-to-deadline (IDLE) is reduced by 8.8% because of the idle power consumed by the basic processor. If we put the processor into sleep mode, the EPI-to-deadline (SLEEP) is reduced by 5.2% of the baseline. The results are similar for the stabilized four-thread OoO processor, where power, EPI-to-deadline (IDLE), and EPI-to-deadline (SLEEP) are reduced by 31.0%, 8.7%, and 3.1% of the basic four-thread OoO processor, respectively, as displayed in Figure 23. The EPI-to-task-completion is only reduced by 2.5%.

7.3. Performance of Each Thread in the OoO SMT Processor

We have shown that the stabilization framework can make the overall throughput of OoO SMT processors predictable. However, if different real-time applications execute on a SMT processor, it is necessary to make the performance of each individual thread predictable as well. To achieve this, processor bandwidth must be shared equitably among all threads running concurrently. Many studies [Cazorla et al. 2004; Tullsen et al. 1996; Tullsen and Brown 2001; Cazorla et al. 2003; Luo et al. 2001] have focused on throughput fairness of non-real-time applications by controlling either instruction fetch policies or dynamic resource usage. ICOUNT [Tullsen et al. 1996] prioritizes threads with few instructions residing in a processor. LC-BPCOUNT [Luo et al. 2001] gives a thread higher priority if it has low confidence branch predictions. Some studies

Fig. 22. MIPS rate, power, and EPIs of the stabilized two-thread OoO processor relative to the basic two-thread OoO processor.

Fig. 23. MIPS rate, power, and EPIs of the stabilized four-thread OoO processor relative to the basic four-thread OoO processor.

[Barre et al. 2008; Mische et al. 2008; Yamasaki et al. 2007] have shown that hard real-time tasks can be supported in a SMT processor by guaranteeing the performance of one thread in the SMT processor while the rest of threads execute soft real-time or non-real-time tasks. Jain et al. [2002] schedules soft real-time tasks in all threads of a two-thread OoO processor by profiling all possible co-scheduled tasks to obtain co-schedule performance.

An equitable throughput balancing fetch policy can be applied on top of our stabilization framework. Ideally, if the fetch policy can guarantee that each thread receives the same amount of compute cycles, the target MIPS rate will be shared evenly by the threads. We propose a hybrid fetch policy combining RR and CICOOUNT. RR is a common fetch policy. It treats threads equally, but it does not monitor the resource utilization of threads. CICOOUNT is a new fetch policy that we propose to equalize the throughput for all threads in an SMT processor. CICOOUNT prioritizes threads with few committed instructions. Different from ICOUNT, which monitors the number of on-the-fly instructions, CICOOUNT monitors the exact number of committed instructions of a thread. Using CICOOUNT alone will significantly lower performance because fast threads cannot fetch instructions until slow threads can catch up. Given a target throughput, T , and a threshold λ , the fetch policy is switched from RR to CICOOUNT to speed up slow threads and slow down fast threads whenever the throughput of any thread is out of the range between $T(1 - \lambda)$ and $T(1 + \lambda)$. Once all threads' throughputs are within the range, the fetch policy is switched back to RR. Figure 24 shows the dynamic MIPS rates of each thread for the stabilized two-thread OoO processor. The overall throughput is set to 800 MIPS. The ideal throughput, T , of each thread is expected to be 400 MIPS. With the RR fetch policy (Figure 24(a)), the throughput of each thread is in the range of 296 to 506 MIPS. It is about $\pm 26.3\%$ of the ideal balanced throughput. Figure 24(b) shows the throughput of threads with our hybrid fetch policy. The threshold λ is set to 5%. The throughputs are stabilized in the range of 379 to 422 MIPS—that is, within $\pm 5.3\%$ of the ideal balanced throughput. To have the MIPS rate of each thread close or higher than the ideal throughput, we can slightly increase the target MIPS rate. For example, instead of the target MIPS rate of 800 MIPS, one could set it to 840 MIPS to compensate for the variance.

For the four-thread OoO processor, the threshold λ is again set to 5% and the stabilized MIPS rate of each thread T is 250 MIPS. The range of dynamic MIPS rate of each thread for RR fetch policy is about $\pm 35.4\%$ of the ideal throughput shown in

Fig. 24. Dynamic MIPS rates of each thread with RR fetch policy (a) and hybrid fetch policy (b) for the stabilized two-thread OoO processor.

Fig. 25. Dynamic MIPS rates of each thread with RR fetch policy (a) and hybrid fetch policy (b) for the stabilized four-thread OoO processor.

Figure 25(a). In Figure 25(b), with the hybrid instruction fetch policy, the dynamic MIPS rates are in the range of 237 to 268 MIPS ($\pm 5.2\%$ of the ideal throughput).

8. DISCUSSION

We have demonstrated in a realistic situation the design of robust PID controllers to maintain a predictable MIPS rate for tasks of more than a few million instructions. Our MIPS rate stabilization combined with the real-time application development framework of Section 2 provides a way for tasks to meet deadlines while saving power/energy in the processor. Once a processor powerful enough to support the slowest task is selected by either applying WCET analysis or the average case profile information, the processor throughput can be stabilized within a very narrow band around the target throughput in a few million instruction executions to meet deadlines. Stabilization enhances the performance predictability of the processor and reduces heat dissipation by cutting power consumption. Stabilization degrades performance since the target MIPS rate must be sufficient to execute the most demanding tasks on time, but it yields power and energy savings on top of improved performance predictability. Comparing stabilized IO and OoO processors, we have shown that the stabilized OoO processor can have superior power/energy savings over the stabilized IO processor while targeting the same real-time QoS, mostly because executing instructions in parallel is more power efficient than increasing the frequency. Demonstrating that OoO processors can be more power efficient than IO processors for a given MIPS rate is an interesting and possibly counterintuitive result. The MIPS rate stabilization scheme makes the execution rate more predictable so that compilers can optimize the generated code more easily by simply minimizing the number of instructions.

In SMT processors, the processor throughput can also be stabilized, although predicting the execution time for such processors is more complex than for applications running on single-thread OoO processors because of resource interferences. The stabilization of the MIPS rate of SMT processors combined with hybrid instruction fetch policies provides a multithreaded hardware platform with a highly predictable, stable MIPS rate so that task schedulers can more easily optimize the schedule to meet the deadlines of individual tasks at lower power costs.

By its nature, the downside of stabilization is that the baseline processor is always faster, as it can freely take advantage of all ILP opportunities with the highest clock rate possible. However, if the maximum performance of the baseline processor is ever needed in an environment, the stabilization can be turned off.

Finally, this article is based on a previous paper [Suh and Dubois 2009] and adds results for OoO SMT processors. To achieve this, we ported all of our applications from

SimpleScalar using the PISA instruction set and its compiler to the SESC simulator using the MIPS instruction set and a MIPS compiler. Additionally, power measurements are made with McPat instead of Wattch. Interestingly, it turns out that the PID control parameters and the design choices for the feedback loop are still the same even with the different compiler, instruction set, and simulators. The target MIPS rates and the number of tasks in the selected benchmark programs had to be changed because of different instruction counts, but conclusions on processor stabilization obtained in Suh and Dubois [2009] remain valid.

9. CONCLUSION AND FUTURE WORK

We have demonstrated how to stabilize the throughput of complex processors with hardware-managed cache hierarchies to improve their performance predictability, power dissipation, and energy efficiency. The framework to stabilize processor throughput is a simple PID feedback controller that dynamically adjusts the frequency and voltage of the processor (DVFS). A simple DVFS scheme with only four steps is sufficient for a wide range of stabilization target throughputs, not only for IO processors but also for less predictable OoO processors running one or multiple threads. The simple PID control-based stabilization framework that we have introduced and demonstrated in this article can be implemented by software in the OS layer or by hardware with simple discrete time controllers widely available in the control market.

Research avenues to exploit the results presented in this article are still open. Better hardware support, such as adaptive feedback controller and finer-grain DVFS steps exploiting Intel's XScale frequency and voltage resynchronization technology, would most probably improve the stabilization with minimal penalty. We limited our target MIPS rate to 550 MIPS in this work because we wanted to demonstrate that a single controller design could be deployed for a very wide set of tasks, noting that the processor must meet the MIPS rate for the task with the lowest ILP. In practice, it is customary that a real-time system runs only a few target programs repetitively, possibly with less variability. In this case, the target MIPS rate may be higher. The framework is flexible enough to let the thread scheduler arbitrarily change the target MIPS rate on-the-fly. PID parameters could even be tuned adaptively on-the-fly as well, depending on how critical the settling time is. The target MIPS rate could be different for different tasks or programs, based on their own requirement. Further research is also warranted into how to adopt the framework in GALS systems or MCD architectures to manage individual components or domains effectively. Finally, with the port to SESC of the stabilization scheme, we will be able to explore MIPS rate stabilization in chip multiprocessors.

REFERENCES

- Jonathan Barre, Christine Rochange, and Pascal Sainrat. 2008. A predictable simultaneous multithreading scheme for hard real-time. In *Architecture of Computing Systems—ARCS 2008*. Lecture Notes in Computer Science, Vol. 4934. Springer, 161–172.
- Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. 2000. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8, 3, 299–316.
- Francisco J. Cazorla, Enrique Fernandez, Alex Ramírez, and Mateo Valero. 2003. Improving memory latency aware fetch policies for SMT processors. In *High Performance Computing*. Lecture Notes in Computer Science, Vol. 2858. Springer, 70–85.
- Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. 2005. Architectural support for real-time task scheduling in SMT processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. ACM, New York, NY, 166–176.
- Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. 2004. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*. IEEE, Los Alamitos, CA, 171–182.

- Bruce Childers, Hongliang Tang, and Rami Melhem. 2000. Adapting processor supply voltage to instruction-level parallelism. In *Proceedings of the Kool Chips 2000 Workshop*.
- Lawrence T. Clark. 2001. Circuit design of XScale microprocessors. In *Proceedings of the 2001 Symposium on VLSI Circuits*.
- Gilberto Contreras, Margaret Martonosi, Jinzhang Peng, Guei-Yuan Lueh, and Roy Ju. 2007. The XTREM power and performance simulator for the Intel XScale core: Design and experiences. *ACM Transactions on Embedded Computing Systems* 6, 1, 4.
- Steven Dropsho, Greg Semeraro, David H. Albonesi, Grigorios Magklis, and Michael L. Scott. 2004. Dynamically trading frequency for complexity in a GALS microprocessor. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*. IEEE, Los Alamitos, CA, 157–168.
- Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. 2003. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism* 5, 1, 1–33.
- Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. 1986. *Feedback Control of Dynamics Systems*. Prentice Hall.
- Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*. IEEE, Los Alamitos, CA, 3–14.
- Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4, 1–28.
- André Hergenhan and Wolfgang Rosenstiel. 2000. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of the Conference on Design, Automation, and Test in Europe*. ACM, New York, NY, 552–559.
- Christopher J. Hughes, Praful Kaul, Sarita V. Adve, Rohit Jain, Chanik Park, and Jayanth Srinivasan. 2001a. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 254–265.
- Christopher J. Hughes, Jayanth Srinivasan, and Sarita V. Adve. 2001b. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 250–261.
- Intel. 2009. *Intel Atom Processor Z5XX Series Datasheet*. Intel Corporation.
- Anoop Iyer and Diana Marculescu. 2002. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 158–168.
- Rohit Jain, Christopher J. Hughes, and Sarita V. Adve. 2002. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. IEEE, Los Alamitos, CA, 134–145.
- Masaaki Kondo, Hiroshi Sasaki, and Hiroshi Nakamura. 2007. Improving fairness, throughput and energy-efficiency on a chip multiprocessor through DVFS. *ACM SIGARCH Computer Architecture News* 35, 1, 31–38.
- Tushar Kumar, Jaswanth Sreeram, Romain Cledat, and Santosh Pande. 2007. A profile-driven statistical analysis framework for the design optimization of soft real-time applications. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 529–532.
- Etienne Le Sueur and Gernot Heiser. 2011. Slow down or sleep, that is the question. In *Proceedings of the USENIX Annual Technical Conference*.
- Sheng Li, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2009. McPAT 1.0: An integrated power, area, and timing modeling framework for microcore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 469–480.
- Kun Luo, Manoj Franklin, Shubhendu S. Mukherjee, and Andre Sezne. 2001. Boosting SMT performance by speculation control. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 9.
- Jörg Mische, Irakli Guliasvili, Sascha Uhrig, and Theo Ungerer. 2010. How to enhance a superscalar processor to provide hard real-time capable in-order SMT. In *Architecture of Computing Systems—ARCS 2010. Lecture Notes in Computer Science*, Vol. 5974. Springer, 2–14.
- Jörg Mische, Sascha Uhrig, Florian Kluge, and Theo Ungerer. 2008. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'08)*. IEEE, Los Alamitos, CA, 371–376.

- K. Mistry, M. Armstrong, C. Auth, S. Cea, T. Coan, T. Ghani, T. Hoffmann, A. Murthy, J. Sandford, R. Shaheed, K. Zawadzki, K. Zhang, S. Thompson, and M. Bohr. 2004. Delaying forever: Uniaxial strained silicon transistors in a 90nm CMOS technology. In *Proceedings of the 2004 Symposium on VLSI Technology*. IEEE, Los Alamitos, CA, 50–51.
- Pablo Montesinos Ortego and Paul Sack. 2004. SESC: SuperESCaLar simulator. In *Proceedings of the 17th Euro Micro Conference on Real Time Systems (ECRTS'05)*. 1–4.
- Eric Rotenberg. 2001. Using variable-Mhz microprocessors to efficiently handle uncertainty in real-time systems. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 28–39.
- SAS Institute. 1990. *SAS/STAT User's Guide: Version 6* (Vol. 2). SAS Institute.
- John Paul Shen. 2006. Lost in the Bermuda Triangle: Complexity vs. Energy vs. Performance. Retrieved January 22, 2015, from <http://www.csl.cornell.edu/~albonesi/wced06/shen.pdf>.
- Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News* 36, 5, 45–57.
- Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. 2002. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 17–28.
- Phillip Stanley-Marbell, Michael S. Hsiao, and Ulrich Kremer. 2003. A hardware architecture for dynamic performance and energy adaptation. In *Power-Aware Computer Systems*. Lecture Notes in Computer Science, Vol. 2325. Springer, 33–52.
- Jinho Suh and Michel Dubois. 2009. Dynamic MIPS rate stabilization in out-of-order processors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 46–56. DOI: <http://dx.doi.org/10.1145/1555754.1555763>
- Yudong Tan and Vincent Mooney. 2007. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems* 6, 1, 7.
- Dean M. Tullsen and Jeffery A. Brown. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 318–327.
- Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *ACM SIGARCH Computer Architecture News* 24, 2, 191–202.
- Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Bruce Jacob. 2003. A control-theoretic approach to dynamic voltage scheduling. In *Proceedings of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, New York, NY, 255–266.
- Ce Xu, Thinh M. Le, and Teng-Tiow Tay. 2005. H. 264/AVC codec: Instruction level complexity analysis. In *Proceedings of the 9th IASTED International Conference on Internet and Multimedia Systems and Applications*.
- Nobuyuki Yamasaki, Ikuo Magaki, and Tsutomu Itou. 2007. Prioritized SMT architecture with IPC control method for real-time processing. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, Los Alamitos, CA, 12–21.
- Yifan Zhu and Frank Mueller. 2004. Feedback EDF scheduling exploiting dynamic voltage scaling. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*. IEEE, Los Alamitos, CA, 84–93.

Received October 2013; revised December 2014; accepted December 2014