Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking*

Brian Greskamp and Josep Torrellas
University of Illinois at Urbana–Champaign
http://iacoma.cs.uiuc.edu

Abstract

Under current worst-case design practices, manufacturers specify conservative values for processor frequencies in order to guarantee correctness. To recover some of the lost performance and improve single-thread performance, this paper presents the *Paceline* leader-checker microarchitecture. In Paceline, a *leader* core runs the thread at higher-than-rated frequency, while passing execution hints and prefetches to a safely-clocked *checker* core in the same chip multiprocessor. The checker redundantly executes the thread faster than without the leader, while checking the results to guarantee correctness. Leader and checker cores periodically swap functionality. The result is that the thread improves performance substantially without significantly increasing the power density or the hardware design complexity of the chip. By overclocking the leader by 30%, we estimate that Paceline improves SPECint and SPECfp performance by a geometric mean of 21% and 9%, respectively. Moreover, Paceline also provides tolerance to transient faults such as soft errors.

1 Introduction

Increased transistor integration together with constraints on power dissipation and design complexity are motivating processor designers to build Chip Multiprocessors (CMP). While these platforms will work well for parallel applications, it is unclear to what extent they will be able to speed up legacy sequential applications. This is because many of these applications have proven to be hard to parallelize with current compiler technology.

An alternative approach is to keep the sequential software unmodified, and leverage multiple cores to speed up the application transparently using hardware mechanisms only. This approach often relies on running the application on two coupled cores, where one of them is used to speed up the execution on the other. Examples of this approach include Slipstream [26], Dual-Core Execution [34], and Future Execution [9]. In Slipstream [26], the Advanced stream speculatively skips predictable instructions from the application, while passing execution hints such as data values and branch outcomes to the Redundant stream which, as a result, runs the application faster. In Dual-Core Execution [34], the Front core fetches and preprocesses the instruction stream, without stalling for long-latency cache misses, while the Back core benefits from these prefetches and from the correct dynamic stream of instructions from the Front core. Finally, in Future Execution [9], a Prefetching core uses value prediction to execute future non-control instructions, prefetching data that will be needed by the Primary core.

To further improve the potential of this general approach, we propose to overclock the leader core — i.e., clock it at a frequency higher than its rating — so that it can run faster and provide hints and prefetched data earlier to the follower core. There are two reasons why a modest degree of overclocking is viable. First, current processors are rated with a non-negligible safety margin for frequency. This is a result of several factors, including the way processor chips are marketed in frequency bins, process variation within a CMP, and protection against aging effects, temperature peaks, and voltage variations. Second, even if the leader core is overclocked beyond what is safe, occasional timing errors can be tolerated, given than the follower core checks the results and guarantees correct execution.

In this paper, we present an architecture called *Paceline* that supports this approach. Paceline improves the performance of a single thread by running it redundantly on two cores of a CMP, called *Leader* and *Checker*. The leader is clocked at a frequency higher than nominal, exploiting the safety margin for frequency and even causing occasional timing errors. Meanwhile, the checker core runs at the rated, safe frequency. The leader prefetches data into the L2 cache that it shares with the checker, and also passes branch

*This work was supported in part by the National Science Foundation under grant CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; SRC GRC under grant 2007-HJ-1592; and gifts from IBM and Intel.

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

outcomes to the checker. This improves the checker's IPC and allows it to keep up with the accelerated leader. The result is that the thread executes faster than on a single baseline core.

The hardware periodically compares the architectural state of the leader and the checker, and is able to detect and recover from different types of errors. These errors include timing errors due to overclocking, incoherence caused by the same load from the leader and the checker returning different values and, importantly, transient faults such as soft errors.

As it executes, the overclocked leader dissipates higher power than a baseline core. Meanwhile, the checker, by leveraging the prefetched data and branch outcomes from the leader, expends less power than a baseline core. To be able to sustain higher than baseline speed without overheating, the two cores periodically alternate the leader position. The operation is analogous to a paceline of two bicycle riders where riders take turns to lead. The leader expends more effort while sheltering the other rider.

We envision a Paceline CMP as containing multiple of these leader–checker core pairs. Each core pair shares an L2 and includes simple hardware to periodically compare state and to pass branch outcomes. This hardware requires only small design modifications — mostly confined to the interface between the L1 and L2 caches — and can be easily disabled, returning the core pair to the standard CMP working mode.

Our simulation-based results show that Paceline substantially improves the performance of a thread without significantly increasing the power density or the hardware design complexity of the chip. By overclocking the leader by 30%, we roughly estimate that Paceline improves SPECint and SPECfp performance by a geometric mean of 21% and 9%, respectively.

This paper is organized as follows. Section 2 examines the potential for overclocking; Section 3 presents the Paceline architecture; Sections 4 and 5 evaluate Paceline; and Section 6 discusses related work.

2 Characterizing Overclockability

Paceline exploits three sources of overclockability in the leader core: *grading artifacts* arising from the way processors are binned and marked for sale; process and environmental *safety margins*; and *error tolerance* at frequencies beyond the safe one. This section characterizes these sources. Unless otherwise indicated, the discussion refers to a 16-core CMP in 32nm technology.

2.1 Grading Artifacts

After fabrication, each die is tested for functionality and speed. The latter process is called *speed binning*, where the objective is to assign one of several pre-determined speed grades to each part. For example, a manufacturer might offer 4, 4.5, and 5 GHz speed grades, and will label each part with the highest speed grade at which it can safely and reliably operate under worst case conditions.

The binning process introduces overclockability in two ways. The first arises from the fact that bin frequencies are discrete. For example, under the scheme described above, a part that passes all tests at 4.8 GHz will be placed in the 4.5 GHz bin. As of early 2007, the Intel E6000 series and the AMD Athlon 64 X2 series space their bin frequencies by 7-14% [3, 15]. If we assume a spacing of 10% as an example, a processor can safely run on average 5% faster than its binned frequency specification.

Secondly, binning contributes to overclockability because within-die process variation causes some cores on a given CMP to be faster than others. In current practice, the slowest core dictates the bin for the entire die. While it is possible to bin each core on the die individually, this may not be cost-effective. If CMPs continue to be binned according to the slowest core, each die will contain many underrated cores.

To get a feel for this effect, using the variation model of Teodorescu *et al.* [28] with $\sigma/\mu = 9\%$ for the threshold voltage, we find that the σ/μ of on-die core frequency is 4%. Monte Carlo simulations show that this corresponds to an average 16% difference in frequency between the fastest and the slowest cores on a 16-core die. As another example, Humenay *et al.* [13] estimate a 17% difference in frequency between the fastest and slowest core on a 9-core 45nm die.

2.2 Safety Margins

Process and environmental margins for device aging, operating temperature, and supply voltage can also be exploited for overclocking. For example, device aging due to Negative Bias Temperature Instability (NBTI) [20] and Hot Carrier Injection (HCI) [27] causes critical path delays to increase over the lifetime of the processor — and this is especially severe in near-future technologies. A typical high-performance processor's operational lifetime is seven to ten years [2]. According to [20], the delay increase due to NBTI alone over that period is 8% in 70nm technology, and HCI adds additional slowdown on top of that. Since processors are guaranteed to operate at the rated frequency for the entire design lifetime, significant overclockability exists in fresh processors where aging has not yet run full course.

Processors are typically rated for maximum device junction temperatures in the 85–100°C range (e.g., [14]) even though operating temperatures are often lower. At lower temperatures, transistors are faster, so overclocking is possible. For example, consider a chip where the hottest unit is at 65°C, about 20°C below the maximum temperature. In this case, the analysis in [12] and data from [16] show

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE

IEEE
COMPUTER
SOCIETY

that the safe frequency increases by approximately 5% for 180nm technology.

Finally, while a processor's off-chip voltage regulator typically has a tight tolerance, on-chip logic is subject to sizable local voltage drops. These drops are due to rapid current changes that cause *L dI/dt* supply noise and sustained periods of high activity that cause *IR* drops. Since these drops are difficult to predict at design time, designers assume conservative margins to guarantee safe operation. For example, the IBM POWER4 processor is designed to provide voltages within $\pm 10\%$ of nominal at all points in the worst case [32]. The logic must therefore be designed to operate correctly under a 10% supply voltage droop even though this is not the common case. This fact can offer some limited room to overclock the processor.

2.3 Error Tolerance

We can further increase overclocking and push the processor into a regime where it will experience occasional timing errors. This mode is called *Timing Speculation* (TS), and has been well studied [8, 29, 31, 33]. Empirical data demonstrates that error onset is only gradual [8]. This is because a processor has a variety of path delays, and not all paths are exercised with a given input. For modest clock rate increases, only the longest paths fail.

The Razor project fabricated 180nm in-order Alpha microprocessors and measured the error rates under different frequencies and voltages. In [6], they show the error rate versus voltage for two specimens at 120 MHz and 140 MHz. If the supply voltage is reduced so that the chip begins to experience timing errors at 120 MHz (i.e., all safety margins have been removed), increasing the frequency to 140 MHz yields less than one error per million instructions. This corresponds to a 17% frequency improvement in exchange for an error rate of 10^{-6} per instruction, which Paceline can easily correct.

2.4 Exploiting Overclockability

Exploiting the above factors for overclocking requires various levels of microarchitecture support. Removing the Grading Artifacts requires the fewest microarchitecture changes: The manufacturer must speed test each core individually and populate a one-time-programmable, architecturally-visible table with the core frequencies. The OS can then use this table to set operating conditions. We call this solution *Fine Grain Binning* (FGB).

At the next level of microarchitecture complexity, *Timing Error Avoidance* (TEA) schemes (e.g., [2, 4, 16, 30]) can estimate the maximum safe frequency dynamically as the processor runs. These techniques either embed additional "canary" critical paths [30], which have delays slightly longer than the actual critical paths, or they directly monitor the delay of the existing critical paths [2, 4, 16]. Either way, they are able to determine when the clock period is too close to the actual critical path delay. Although in theory TEA can push the clock frequency arbitrarily close to the zero-error point, practice requires that some safety margins be maintained to avoid errors. Consequently, TEA schemes can remove the Grading Artifacts and some but not all of the Safety Margins.

With an additional increase in microarchitecture complexity, we can exploit all three factors for overclocking, including timing speculation, where the processor experiences occasional errors. The required support is an error detection and correction mechanism. Some examples are Razor [8], Diva [33], TIMERRTOL [29], and X-Pipe [31]. In practice, all of these schemes require fairly invasive microarchitectural changes, either by modifying all processor pipeline latches and the corresponding control logic, or by adding a specialized checker backend for the processor core. In the next section, we describe the proposed Paceline microarchitecture, which exploits the three factors for overclocking while requiring minimal modifications to the processor cores and caches.

Figure 1 compares the factors for overclocking exploited by each approach. The figure shows only qualitative data because the factors described above are hard to quantify and not fully orthogonal. In the remainder of the paper, we consider a range of overclocking factors (10% to 40%) that we feel could be accessible to Paceline.

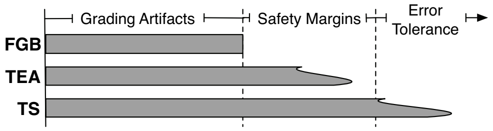

Figure 1. Qualitative comparison of the potential of the three architectural approaches to exploit overclocking.

3 Paceline Architecture

Paceline is a leader–checker architecture that improves the performance of a single thread by running it redundantly on two cores of a CMP. The leader core is clocked at a frequency higher than nominal, exploiting the three factors for overclocking described in Section 2. Meanwhile, the checker core runs at the rated, safe frequency. The leader prefetches data into the L2 cache that it shares with the checker, and also passes branch outcomes to the checker. This improves the checker's IPC and allows it to keep up with the accelerated leader. The hardware periodically compares the architectural state of the leader and the checker, and is able to detect and recover from errors due to overclocking or other effects. The result is that the thread executes faster than on a single baseline core.

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE

As it executes, the overclocked leader dissipates higher power than a baseline core. Meanwhile, the checker, by leveraging the prefetched data and branch outcomes from the leader, spends less power than a baseline core. To be able to sustain higher than baseline speed without overheating, the two cores periodically alternate the leader position. The result is that the chip's power density and maximum temperature are not expected to increase substantially over a baseline system. Intuitively, the operation is analogous to a paceline of two bicycle riders where riders take turns to lead. The leader expends more effort while sheltering the other rider.

A Paceline CMP contains multiple of these leader–checker cores as shown in Figure 2. Each core pair shares an L2 cache and includes simple hardware to periodically compare state and to pass branch outcomes. This hardware requires only very small core modifications and can be easily disabled, returning the core pair to the standard CMP working mode.

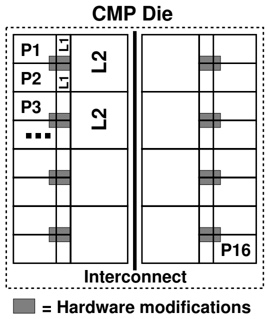

Figure 2. Paceline CMP with 16 cores.

Overall, Paceline speeds up a single thread (of a serial or a parallel program) without significantly increasing CMP power density or hardware design complexity. In the following, we first give an overview of the microarchitecture and characterize the types of errors it can encounter. Then, we present two different Paceline variations, each specialized for handling different types of errors.

3.1 Overview of the Microarchitecture

In Paceline, the leader and the checker cores operate at different frequencies, with the checker lagging behind and receiving branch outcomes and memory prefetches into the shared L2 from the leader. Figure 3 shows the microarchitecture. The region in the dashed boundary is overclocked, while everything else runs at the rated, safe frequency.

The shaded components are the new hardware modules added in Paceline. Specifically, the outcomes of the leader's branches are passed to the checker through the *Branch Queue* (BQ). Moreover, the hardware in both leader and checker takes register checkpoints every n instructions and saves them locally in ECC-protected safe storage. In addition, the hardware hashes the checkpoints into signatures and sends them to the ECC-protected *Validation Queue* (VQ). As execution continues, the VQ checks for agreement between the hashed register checkpoints of the leader and the checker. The VQ sits in the cache hierarchy between the L1 and L2 caches. Since the L1 caches operate in write-through mode as in the Pentium 4 [24], the VQ can potentially see all the memory writes in order. Such capability allows it to provide extra functionality that depends on the types of errors handled. We will see the details in Section 3.3.

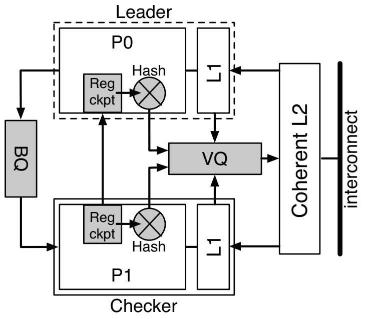

Figure 3. The Paceline microarchitecture.

Although the leader and checker cores redundantly execute the same thread, they do not execute in lock-step. Since they are out-of-order processors, they execute instructions in different orders and even execute different instructions — due to branch misprediction. However, in the absence of errors, their retirement streams are identical.

Given a dynamic write instruction in a program, the leader and the checker will issue the store to the L1 at different times, when each retires the write instruction. The VQ will only allow one of the two stores to propagate to the L2, possibly after performing some validation.

For reads, however, there is no such filtering. A load issued by the leader that misses in the L1 is immediately sent to the L2. If and when the checker issues the corresponding load, it may also send a read request to L2. The advantage of this approach is that it does not require any read buffering at all and, therefore, it is easy to support in hardware. However, it may result in the two loads returning different values — if the location being read is modified in between the reads by, for example, a write from another thread, a DMA action, or a fault. Smolens *et al.* [25] call this problem the *Input Incoherence Problem*. This approach is also used in Slipstream [22] and Reunion [25].

3.2 Types of Errors

To design Paceline, we consider the three potential sources of error shown in Table 1: timing errors due to overclocking, errors due to the input incoherence problem, and

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE

soft errors. Table 1 characterizes them based on (1) whether they re-appear as the code section with the error is rolled back and re-executed, and (2) what part of the Paceline architecture they can affect.

Type of Error	Repeats in Re-Execution?	Can It Affect...	
		Leader?	Checker?
Timing Error Due to Overclock	Likely	Yes: Register state Data read into L1 Data written to L1	No
Input Incoherence	Possibly	Yes: Register state Data read into L1 Data written to L1	No
Soft Error	No	Yes: Register state Data written to L1	Yes: Same as Leader

Table 1. Types of errors considered.

These errors behave differently. Timing errors are likely to repeat after rollback because the same critical paths are likely to be exercised during re-execution. Input incoherence errors may re-occur when other threads repeatedly update the location that the leader and checker are trying to read [25]. Indeed, one can construct a pathological scenario where, in each re-execution, a third processor updates the location between the leader and checker reads. On the other hand, soft errors do not re-occur.

From the last two columns, we see that both timing and incoherence errors can affect the same parts of Paceline. Specifically, they can affect the state in the leader's registers, and the state in the leader's L1 that has been read from memory or been written by the processor. However, they cannot affect the checker. On the other hand, soft errors can affect both the leader and the checker — both the register state and the data written by the processor into the L1.

A final key difference not shown in the table is that soft errors are much less frequent that the other two types. For example, based on data in [28], this paper will estimate in Section 5 that the expected timing error rate at optimal performance is approximately one per 10^5 instructions. A similar incidence of incoherence errors is shown in the environment considered by Reunion [25]. On the other hand, the soft error rate may be in the range of one per 10^{15}–10^{20} instructions.

3.3 Detailed Microarchitecture

Based on the previous discussion, we propose two different levels of microarchitecture support for Paceline. The first one is a simple design that targets only the frequent types of errors — namely the timing and input incoherence errors. The second level is a high-reliability design that targets all three types of errors. We call these designs *Simple* and *High-Reliability*, respectively. They can both be supported by a single microarchitecture, with some of the features being disabled for the *Simple* design.

There is a key difference between the two designs. Specifically, since timing and incoherence errors can only affect the leader, *Simple* can recover without rollback — it recovers simply by copying the checker's state to the leader's. On the other hand, since soft errors can affect both cores, *High-Reliability* must recover by rolling back. Unfortunately, the timing and incoherence errors will likely or possibly repeat during re-execution. To avoid this, the re-execution has to be performed under different conditions. As we will see, this complicates the *High-Reliability* design over the *Simple* one. Overall, these two designs correspond to different cost-functionality tradeoffs.

In this section, we describe the two designs. In both cases, the Paceline microarchitecture must provide two mechanisms: one to periodically compare the state of the leader and the checker, and another to repair the state of the leader-checker pair when the comparison mismatches.

3.3.1 The *Simple* Design

In this design, the checker is assumed to be always correct. Ensuring that the leader is also correct is a performance issue, not a correctness one. If the leader diverges from correct execution, it will not help speed up the checker: the leader will prefetch useless data into the L2 and pass useless branch outcomes to the checker, whose branch outcomes will not match the leader's predictions.

Since correctness is not at stake, the checkpoint signature can be short — e.g., four bits. The VQ has a *Signature Queue* that contains signatures that cores deposit when they take checkpoints and hash them. The VQ hardware compares signatures from the two cores corresponding to the same point in the program. Moreover, writes from the checker's L1 are immediately routed to the L2, while those coming from the leader's L1 are discarded. If the leader and checker checkpoints are different but hash into the same signature due to aliasing, correctness does not suffer. Moreover, since the divergence is likely to be detected in subsequent signature comparisons, the leader will not continue on a wrong path for long.

When a mismatch is detected, we know that a timing or incoherence error has happened and that its effect is confined to the leader's processor and L1 cache. At this point, the VQ state is invalidated and we need to repair the register and L1 cache states of the leader. To repair the register state, the checker's checkpoint is copied to the leader's registers, effectively rolling the leader forward past the error. To repair the L1 state, we have two options. The simplest is to invalidate the leader's L1 contents. As execution resumes, the leader will re-fetch the correct data from the L2 on demand. This approach does not involve a large performance overhead because all the lines invalidated from the L1 are in the L2.

A more advanced option is to have the VQ record the read miss and write addresses emanating from the leader L1 during each checkpoint interval and to selectively invalidate only those lines when a signature mismatches. This approach is not guaranteed to repair the L1 state, so an incoherence may resurface in future signature comparisons. If a certain number of signature comparisons in a row are found to mismatch, Paceline reverts to flushing the leader's L1, which eliminates all the lingering incoherence.

3.3.2 The *High-Reliability* Design

In this design, the leader or the checker may suffer an error. We need to make sure that any divergence between the leader and the checker states is detected. Missing a divergence leads to a correctness problem, since erroneous data can then propagate to the L2.

Paceline compares the register and L1 state of leader and checker. The register state is checked by comparing the checkpoint signatures of the cores, which are stored in the VQ's Signature Queue. In the *High-Reliability* design, the signature has more bits (e.g., sixteen) than in the *Simple* design in order to minimize aliasing. In addition, the L1 state is checked by buffering the leader writes in an in-order circular *Write Queue* in the VQ. Each Write Queue entry contains the address and data of a leader write, together with a Validated bit. When a checker write arrives, the hardware compares its address and data to the next non-validated entry in the Write Queue and, if they match, the Validated bit for the entry is set. Note that writes issued by each core arrive at the VQ in program order. For a checkpoint interval to check successfully, the leader and checker register signatures at the end of the interval must match, and the VQ must have successfully validated all writes in the interval. When this is the case, the Validated writes are removed from the VQ and released to the L2 cache.

If, instead, the signatures mismatch or the writes are not all validated, Paceline rolls the checkpoint interval back and re-executes it. Rolling back the interval involves invalidating the VQ's state, restoring the register state of the last successfully-compared checkpoint in both cores, and repairing the L1 state in both cores. To repair the L1 state, there are two alternatives. The simplest is to invalidate both L1 caches as in *Simple*. In this case, as the leader and checker re-execute, they will naturally re-populate their L1 caches with lines from the L2. A more advanced alternative is to re-fetch only those lines accessed during the failing checkpoint interval (i.e., those which could have contributed to the failure). This can be done by operating the two L1 caches in a special mode during the re-execution that explicitly forces all accesses to miss and re-fetch from the L2. At the next successful checkpoint comparison, re-execution is complete, and the L1s may return to normal mode.

Regardless of which L1 cleanup method is chosen, the VQ operation during re-execution is no different than its normal-mode operation; it buffers leader writes and sends load requests to the L2 as usual. However, some additional precautions are needed to handle timing errors and persistent input incoherence errors. Specifically, recall from Table 1 that a timing error will reappear in a normal re-execution. To avoid this, in a re-execution, the leader is clocked at a lower, safe frequency until the next checkpoint. We envision the high-frequency clock and the safe-frequency clock to be always available to the leader. When the leader enters re-execution mode, the high-frequency clock is disconnected from the leader and the safe-frequency one is fed to it.

A persistent input incoherence error is one that repeatedly occurs every time that the interval is re-executed [25]. Since this event is expected to be rare, the *High-Reliability* design uses a simple approach to handle it. Specifically, when the hardware detects that an interval has been re-executed more than a few times in a row, an interrupt is sent to the other cores to stall them until the local core pair successfully proceeds past the next checkpoint.

Overall, we see that the increased fault coverage of the *High-Reliability* design comes at a hardware cost relative to the *Simple* design.

3.4 Additional Issues

3.4.1 Special Accesses

There are a few types of memory accesses, including atomic Read-Modify-Write (RMW) operations such as atomic increment, and non-idempotent reads such as I/O reads, that require special support in a paired architecture such as Paceline. These accesses are called serializing operations in [25]. In this section, we briefly describe the basic support required.

At the simplest level, serializing operations can be implemented by checkpointing the leader before issuing the operation and then stalling the leader until the checker reaches the same point. Once this happens, the checker also checkpoints. Then, the checkpoint interval is validated, by comparing the checkpoint signatures and, in the *High Reliability* design, also comparing all the writes in the interval. After that, both cores issue the serializing operation, which is merged in the VQ and issued as a single access to memory. Let us call r the data that this operation returns, such as the data read in the RMW operation or in the non-idempotent read. The value r is provided to both cores and also temporarily buffered in the VQ. After that, both cores take a new checkpoint and the checkpoint interval is validated. In case of a match, the cores continue.

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE

If, instead, a mismatch occurs, an error has happened since the previous checkpoint — for example a soft error has affected one of the register files. In this case, the hardware uses the same recovery procedures as before except that the serializing operation cannot be redone. Instead, the buffered *r* value is provided to the processors.

As a simple performance optimization in the case of atomic RMW operations, the leader can issue an exclusive prefetch for the RMW variable into the L2 before stalling to wait for the checker. With this support, by the time the leader and checker together issue the RMW operation, the data is likely to be in the desired state in the L2. Note that this optimization cannot be applied for non-idempotent reads.

It is possible to design higher performance implementations where leader and checker do not need to wait for one another. For example, in an RMW operation, the leader could issue the read and buffer the write in the VQ, rather than waiting for the checker. The checker will later perform its RMW operation and, if both operations agree, a single operation is made visible to the memory. Otherwise, the recovery procedure is followed. Since this paper's evaluation focuses on compute-intensive sequential applications, we do not discuss these issues in depth, and leave them for future work.

3.4.2 Interrupts and Exceptions

Delaying interrupt delivery is correct and allowable as long as the delay is bounded. When an interrupt arrives, Paceline must find some point in the future when the leader and checker have identical architectural state and then deliver the interrupt simultaneously to both. Checkpoint boundaries provide ideal points at which to deliver interrupts. Consequently, when an interrupt signal arrives at the processor pair, it goes first to the VQ. The VQ marks a pending interrupt bit and waits until the next checkpoint is released. It then initiates a rollback to the just-released checkpoint and asserts the interrupt line on both cores before restarting them. Both cores then wake up in the same architectural state and see the interrupt simultaneously. The VQ remembers the interrupt that was just delivered so that it can redeliver the same interrupt in the event of a future rollback recovery.

We assume that synchronous, program-initiated exceptions have precise state semantics in the processor, as is the case in most current designs. Intuitively, an instruction with an exception of this type behaves like a conditional jump with the additional effect of setting one or more status registers. This does not demand any special handling from Paceline; exceptions are handled just as any other instruction. Barring an error, the exception code will execute identically in both cores just as all other instructions do. In the case of an error causing the execution of the exception code in one core only, the next checkpoint comparison will fail and initiate recovery.

4 Performance Analysis

In order to enjoy speedup under Paceline, an application must have two properties: (i) overclocking the leader must increase its performance; and (ii) the improved behavior of the branch predictor and cache subsystems in the checker must increase its IPC. A program that is totally memory-bound or limited by the L2 cache latency will not satisfy (i), while a program that fits completely within the L2 cache and already achieves high branch prediction accuracy without Paceline will not meet requirement (ii).

Here we present a fairly accurate method of using a standard (not Paceline-enabled) microarchitecture simulator to estimate the Paceline speedup for a given overclocking factor and application. To get the required data, we instrument the simulator to output an IPC trace that records the average IPC for each chunk of ten thousand dynamic instructions. We then run the application on the simulator under two configurations. The first represents an overclocked leader with a perfect (infinite performance) checker, while the second represents a checker in the presence of a perfect (infinite performance) leader.

To obtain these leader and checker configurations, we start with the baseline core configuration. The leader configuration differs from the baseline only in being overclocked, so that the latencies of all memory components beyond the L1 cache increase by the overclocking factor. The checker configuration differs from the baseline only in having a perfect branch predictor and a perfect L2 cache (thanks to the BQ and prefetching effects). Running the simulator with each configuration produces IPC traces. From these, we obtain performance traces by multiplying the entries in the leader IPC trace by the leader frequency and those of the checker trace by the checker frequency. Finally, we normalize these numbers to the performance of a baseline core. The two resulting speedup traces give the leader's speedup (L_i) and the checker's speedup (C_i) relative to a baseline core during the ith dynamic instruction chunk. For example, the leftmost two columns of Figure 4(a) show the L_i and C_i traces for the SPECint application `parser`. The overclocking factor used for the leader is 1.3.

We can use L_i and C_i to estimate the speedup of Paceline by observing that, at any given time, the *slowest* of the leader and checker cores determines the overall speed. Since the chunk size that we have chosen (10K instructions) is significantly larger than the typical lag between leader and checker ($<\ 1000$ cycles), both leader and checker will typically be executing the same chunk. Consequently, $P_i = min(L_i, C_i)$, shown in the next column of Figure 4,

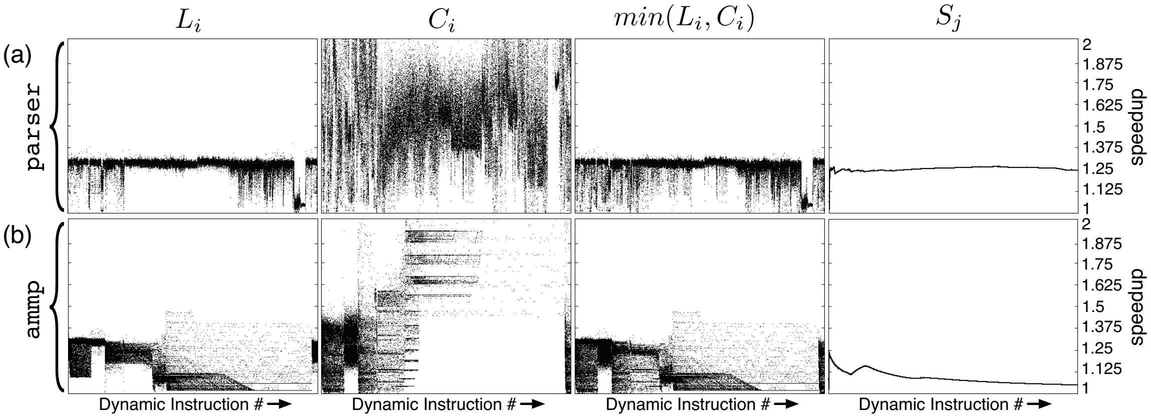

Figure 4. Speedup traces for `parser` (first row) and `ammp` (second row) with a leader overclocking factor of 1.3.

is a good estimate of the instantaneous speedup under Paceline. The speedup S_j over the interval from program start to chunk j is then given by the harmonic mean of all P_i in the interval as shown in Equation 1. Referring to the last column of Figure 4, we see how S_j evolves as the application executes.

$$S_j = \frac{j}{\sum_{i=1}^{j} \frac{1}{P_i}} \tag{1}$$

For an example of an application on which Paceline performs well, see `parser` in Figure 4(a). Here, the L_i speedup trace is consistently near the maximum possible value of 1.3 because the leader experiences few L1 cache misses, so almost all of the execution takes place inside of the overclocked domain. The C_i speedup in the checker from perfect branch and cache subsystems is high most of the time, allowing the checker to keep up with the leader most of the time. Consequently, the speedup converges to 1.25 with an overclocking factor of 1.3.

In contrast, `ammp` in Figure 4(b) does not show much potential for speedup. Early in the execution, the application has good cache and branch behavior in a baseline processor and, therefore, the C_i speedups are often modest. The result is that the checker limits the Paceline speedup. Later, the application enters a memory-bound phase, in which L_i speedups are low, and the leader limits the Paceline speedup.

We have predicted the Paceline speedups for our applications using this model. Table 2 compares the predicted speedups for the SPECint applications to speedups measured on the cycle-accurate Paceline-enabled simulator of Section 5. Given the strong agreement between the two sets of numbers, we believe that our trace-based model provides an attractive alternative to custom simulator implementation when evaluating Paceline.

Appl.	Estimated	Actual	Appl.	Estimated	Actual
bzip2	22%	21%	mcf	6%	9%
crafty	26%	24%	parser	25%	24%
gap	22%	21%	twolf	29%	29%
gcc	28%	27%	vortex	16%	12%
gzip	15%	17%	vpr	29%	29%

Table 2. Estimated versus measured Paceline speedups.

5 Simulation-Based Evaluation

This section uses a modified version of the cycle-accurate WATTCH-enabled [5] SESC [1] simulator to evaluate the performance and power impact of Paceline. Table 3 summarizes the architecture configuration used in all experiments. The cores are chosen to be representative of current and near-future high-performance microarchitectures like the Intel Core 2. Each core has aggressive per-core clock gating to reduce dynamic power consumption. The bottom of the table shows the default parameters for the Paceline features, not present in the baseline architecture.

General	16 OoO cores, 32nm, 5 GHz
Core width	6 fetch, 4 issue, 4 retire
ROB size	152
Scheduler size	40 fp, 80 int
LSQ size	54 LD, 46 ST
Branch pred	80Kb local/global tournament, unbounded RAS
L1 I cache	16KB, 2 cyc, 2 port, 2 way
L1 D cache	16KB WT, 2 cyc, 2 port, 4 way
L2 cache	2MB WB, 10 cyc, 1 port, 8 way, shared by two cores, has stride prefetcher
Cache line size	64 bytes
Memory	400 cyc round trip, 10GB/s max
Design evaluated	*High-Reliability*
VQ write queue	64 entries, 2 ports per core, 8 cyc
Checkpoint interval	100 instructions
Thermal managmt	Swap leader and checker every 200μs

Table 3. Architecture parameters.

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

We evaluate performance and power consumption using SPECint and SPECfp applications. Unless otherwise noted, the experiments assume the *High-Reliability* design with a leader overclocking factor of 1.3 and an error rate of zero.

5.1 Performance

For a given application and in the absence of errors, Paceline performance relative to a single baseline core increases monotonically toward an asymptote as the leader overclocking factor grows. Figure 5 shows the resulting geometric mean speedups of Paceline for SPECfp and SPECint applications with and without the BQ. The figure shows that the SPECint applications require the BQ to obtain any performance improvement. In fact, it can be shown that only `gap` and `mcf` achieve any speedup at all without a BQ; at an overclocking factor of 1.3, they reach speedups of 1.03 and 1.07, respectively. The remaining SPECint applications experience small (1–2%) slowdowns with all overclocking factors. The reason is that almost all of the SPECint working sets fit within the L2 cache. When there are few L2 misses, there can be no prefetching benefit. Consequently, the leader only wastes valuable L2 bandwidth.

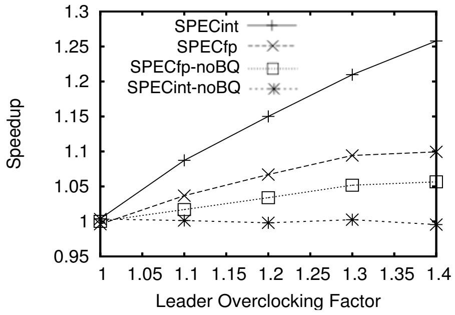

Figure 5. Paceline speedup versus leader overclocking factor in the absence of errors.

On the other hand, with a BQ, the SPECint applications experience large speedups. In this situation, their low L2 miss rates are an advantage, since the leader is able to do useful work when overclocked rather than stalling for main memory accesses. Moreover, in these applications, the dynamic branch count is relatively high and the branch prediction accuracy relatively low, meaning that the branch information can vastly increase the checker IPC (sometimes by a factor of two or more) — as was shown in the `parser` example of Figure 4.

In contrast, the SPECfp applications present behavior that is difficult for Paceline to exploit, whether a BQ is present or not. Most of the time, one of the following two conditions prevails: Either the program is memory-bound so that overclocking the leader has little effect, or the program is hitting the L2 cache and generating good branch predictions so that branch and cache hints do not help the checker keep up with the leader. This was the behavior seen from `ammp` in Figure 4.

Figure 6 shows the speedups for the individual applications with BQ as the leader overclocking factor (*oc*) varies from 1.1 to 1.4 in increments of 0.1. The figure also includes bars for the geometric mean of the integer and floating-point applications. The figure clearly shows that the benefits of Paceline are application-dependent, with speedups for the *oc-1.3* case ranging from 1.02 (`equake`) to 1.29 (`vpr`). Moreover, while some applications (e.g., `vortex`) top out at modest overclocking factors, others could benefit from factors in excess of 1.4.

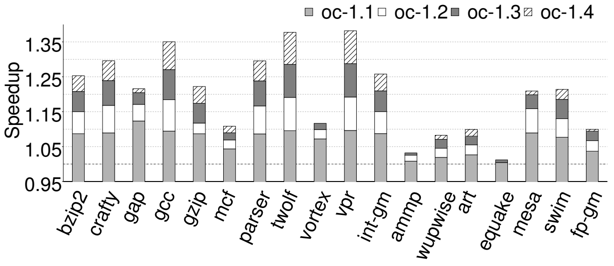

Figure 6. Application speedups with overclocking factors (*oc*) in the 1.1–1.4 range.

5.2 Dynamic Power

We compare the dynamic power consumed by a Paceline pair to that consumed by two baseline CMP cores sharing an L2 and independently executing the same program. Figure 7 shows the absolute dynamic power consumption for a pair of cores under five different configurations. The leftmost bar (*B*) represents the baseline case of two cores independently executing the same application. Moving to the right, the four *P* bars represent a Paceline system running with an overclocking factor of 1.1, 1.2, 1.3, and 1.4, respectively. The four segments within each bar show the power consumed in the leader core (*ldr*), checker core (*ckr*), and L2 cache (*L2*). The Paceline systems also include a *paceline* segment, indicating the power expended in the added structures (BQ, VQ, and checkpoint generation and hash logic). The core (*ldr* and *ckr*) powers include the L1 instruction and data cache access power. Off-chip power is not considered.

The total power of a Paceline system is usually approximately equal to that of a baseline core pair. Due to its improved branch prediction, the checker fetches and executes far fewer instructions in the Paceline system than it would in the base system. This is consistent with results from other work [19], which indicate that for the SPECint applications, perfect branch prediction reduces the number of fetched instructions by 53% on average. Using this infor-

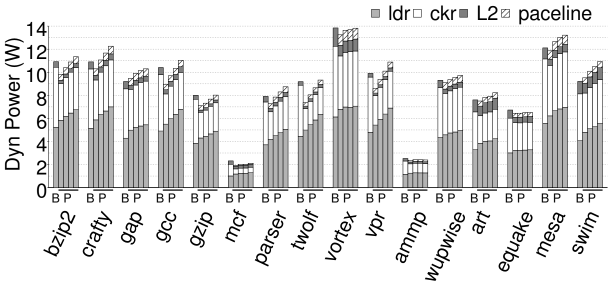

Figure 7. Dynamic power breakdown in Paceline (*P*) and two base cores sharing an L2 (*B*).

mation, we can obtain an intuitive feel for the relationship between baseline and Paceline power.

Consider `gzip`, which attains a speedup $S_{1.3} = 1.17$ with an overclocking factor of 1.3. Our simulations show that the checker fetches 51% fewer instructions than a baseline processor and executes 34% fewer. WATTCH shows that in total, it consumes 47% less energy than the baseline core. The leader core, meanwhile, is executing approximately the same number of total instructions as the baseline core. However, the number of wrong-path instructions increases slightly because the longer relative memory access time leads to slower branch resolution. Additionally, the lower IPC of the leader reduces its energy efficiency, as parts of the chip (especially the clock network) remain active even on idle cycles. The net result is that the leader consumes 4% more energy than the baseline core.

Given the preceding, we can easily compute the total dynamic core power for `gzip` under Paceline. When the application runs on the baseline core, let E_B be the dynamic energy consumed and T_B the time taken. According to the above, the energy of the checker is $E_C = 0.53\, E_B$, and the energy of the leader is $E_L = 1.04\, E_B$. However, due to the Paceline speedup, the leader and checker energies are dissipated over a shorter interval than the baseline execution. The total core power under Paceline is then $(E_L + E_C)\, S_{1.3}/T_B \approx 1.84\, P_B$, where P_B is the power of a baseline core. In this case, the total core dynamic power is less than that of two baseline cores executing the same application.

5.3 Power Density

Although total dynamic power may not increase under Paceline, the leading core power clearly does (by 22% in the case of `gzip`). This thermal imbalance is a concern because it could worsen hot spots that erode overclockability and reduce device lifetime. Fortunately, previous work [11, 21] has shown that *activity migration* can avoid these problems by periodically switching the core on which the hot thread runs. According to [11], swapping cores every $200\mu s$ is sufficient to equalize the leader and checker core temperatures and minimize hotspot formation. In Paceline, core swapping allows both the leader and checker core temperatures to roughly equal the temperature of a baseline core. This is because each core is dissipating approximately the same dynamic power as a baseline core *on average*, and the package's thermal RC network has an averaging effect on temperature. Since static power is dependent only on temperature, chip static power also does not change after applying Paceline.

Swapping the leader and checker is trivial in Paceline, and it has negligible effect on performance even if swaps are performed every $1M$ instructions. To see why, consider the simplest swapping mechanism: A swap starts with both cores rolling back to the most recent checkpoint as though an error had just been detected. Then, a mode bit change in the VQ and in the BQ effectively switches the identities of the two cores. Finally, switching the leader and core clock frequencies completes the swap. In this simple scheme, both cores begin with an empty L1 after the swap, but they repopulate it quickly from the shared L2 [18, 21].

5.4 Sensitivity to Errors

Until now, we have assumed a zero error rate. In this section, we make some assumptions and use data and trends from [28] to model timing errors; then, we estimate the resulting Paceline speedups. To model the rate of timing errors, we assume that the onset of timing errors occurs at an overclocking factor of 1.3, where we set the error rate to one per 10^{12} cycles. The data in [28] shows that a 9% frequency increase pushes the error rate from one per 10^{12} to one per 10^4 instructions. It also shows that the increase is approximately exponential. Consequently, using this information, we set the error rate at an overclocking factor of $1.3 \times 1.09 \approx 1.42$ to be one per 10^4 cycles, and use a straight line in a logarithmic axis to join the 10^{-12} and 10^{-4} error rate points. The resulting error rate line is shown in Figure 8.

To estimate the impact of these errors on Paceline speedups, we test three different recovery penalties. These penalties are modeled as a number of cycles that we force the checker to stall after every timing error. The recovery penalties modeled are 100, 1K, or 10K cycles. As a reference, we have measured in the simulator that a recovery where we roll back the Paceline cores to the last matching checkpoint and flush the L1 caches of both leader and checker, would correspond to approximately stalling the checker for $\approx$ 900 cycles due to the additional L1 misses in the checker after the restart.

Figure 8 shows the resulting Paceline speedups over the baseline processor for each of the three penalties. The speedups correspond to the geometric mean of the SPECint applications. From the figure, we see that at low overclock-

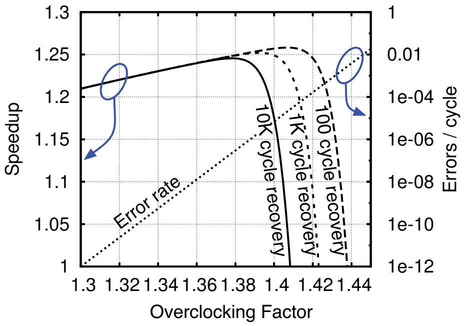

Figure 8. Error rate and geometric mean of the Paceline speedup in SPECint applications as the overclocking factor changes.

ing factors, the speedups are roughly proportional to the factors. However, the exponential increase in error rates causes the linear region to give way to a "cliff" where speedup plummets if the overclocking factor is pushed even 1–2% past the optimal point. We need, therefore, to be careful not to increase overclocking past the optimal point.

The figure also demonstrates that the peak Paceline speedup is only weakly dependent on the recovery penalty — the topmost point of the curves is similar across different recovery penalties. Consequently, we conclude that optimizing recovery time need not be a design priority.

5.5 Implementation Feasibility

Minimizing changes to the processor cores and the cache coherence system has been a key goal for Paceline, and we believe that the resulting microarchitecture can integrate easily in a commercial CMP. Note that the VQ, the principal Paceline structure, falls outside the core boundary and occupies the relatively non-critical path between the L1 and L2 caches. Moreover, cache coherence, which happens at the L2 level, is completely oblivious to Paceline. Although the core itself does require some modifications (shown shaded in Figure 3), they affect only the fetch and retirement stages. The renaming, scheduling, and execution logic of the core is completely unaffected. Even some of the hardware billed to Paceline may in fact already be present in the baseline system: several proposals for future core features (e.g., thread-level speculation and transactional memory) rely on the ability to perform frequent register checkpoints, using hardware like the register checkpoint box in Figure 3.

Perhaps the most demanding feature of Paceline in terms of implementation and verification effort is the requirement that, at a given time, different cores be able to run at different frequencies. Adding clock domains to a design complicates verification, so currently available CMP designs run all cores at the same frequency. However, pressure to improve power efficiency is already forcing chip makers to implement per-core frequency scaling. The upcoming Barcelona microarchitecture from AMD, for example, supports it [7].

6 Related Work

Better than Worst-Case Design: The Razor project [6, 8] examined a simple Alpha pipeline and found substantial voltage safety margins. Other recent work has shown that device aging [20, 27] and process variation [13, 28] dictate large safety margins as technology scales to 32nm and below. A large volume of prior work has focused on removing these margins to improve performance or reduce power. For example, CTV [23] and TIMERRTOL [29] showed how to design logic with dual-phase clocking that is capable of detecting and tolerating timing faults. Alternative proposals such as TEAtime [30] and [2, 4, 16] avoid timing faults altogether by dynamically monitoring path delays. Finally, the Razor [6, 8] and X-Pipe [31] projects showed how to apply timing error tolerance in microprocessor designs.

Core-Pairing Architectures: Paceline improves single-thread performance and, with its *High-Reliability* design, also provides fault tolerance to soft errors. There are several proposed architectures that couple two cores in the same CMP to either improve single-thread performance or to provide fault tolerance to soft errors.

Core-pairing for single-thread performance includes Slipstream, Dual-Core Execution, and Future Execution. In Slipstream [22, 26], one core speculatively skips predictable instructions from the application, while passing as hints all data values and branch outcomes to the other core. In Dual-Core Execution [34], one core fetches and executes instructions, without stalling for long-latency cache misses, passing the stream of dynamic intructions to the other core and also prefetching data for it. In Future Execution [9], one core performs value prediction to execute future non-control instructions, prefetching data that will be needed by the other core. Paceline differs from these designs in that it provides single-thread performance through overclocking, and in that it also provides fault tolerance to soft errors.

Core-pairing for soft-error tolerance includes several designs. For example, in CRTR [10], one core provides branch outcomes, load values and register results to the other core, so that the latter can detect and recover from faults. Madan and Balasubramonian [17] show how, by running the checker core at a lower frequency than the leader, they can provide energy savings. However, they do not suggest leader overclocking. Reunion [25] provides a loosely-coupled, single-frequency core-pair microarchitecture. Paceline differs from these designs in its focus on improving single-thread performance.

Paceline and Reunion have relatively similar microarchitectures. However, they make different design tradeoffs. First, to validate the correctness of instructions, the Reunion hardware checks the instructions' results before the instructions retire from the ROB, while in Paceline, the hardware checks the write streams after the L1 — making the hardware modifications in Paceline less intrusive. Second, the intervals between validations are shorter in Reunion (one instruction long for the data in [25]) than in Paceline (100 instructions between checkpoints). Finally, the lag between cores is smaller in Reunion than in Paceline.

7 Conclusion

We have shown that there is a substantial potential for overclocking in near-future CMPs. The Paceline microarchitecture provides a safe means of exploiting this overclockability to improve the performance of a thread (of a serial or a parallel application) without significantly increasing (i) CMP power density or (ii) hardware design complexity. With an overclocking factor of 1.3, we estimate that Paceline improves SPECint and SPECfp performance by a geometric mean of 21% and 9%, respectively. At the same time, the *High-Reliability* design provides tolerance to transient faults such as soft errors.

References

[1] The SESC simulator. http://sourceforge.net/projects/sesc/.

[2] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra. Circuit failure prediction and its application to transistor aging. In *VLSI Test Symposium*, pages 277 – 286, 2007.

[3] AMD Corporation. AMD NPT family 0Fh desktop processor power and thermal data sheet, June 2007. Datasheet.

[4] J. A. Blome, S. Feng, S. Gupta, and S. Mahlke. Online timing analysis for wearout detection. In *Workshop on Architectural Reliability*, 2006.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[6] S. Das et al. A self-tuning DVS processor using delay-error detection and correction. In *IEEE Symposium on VLSI Circuits*, June 2005.

[7] J. Dorsey et al. An integrated quad-core Opteron processor. In *ISSCC*, pages 102–103, 2007.

[8] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, pages 7–18, 2003.

[9] I. Ganusov and M. Burtscher. Future execution: A prefetching mechanism that uses multiple cores to speed up single threads. In *ACM TACO*, volume 3, pages 424–449, 2006.

[10] M. Gomaa, C. Scarborough, T. N. Vijaykumar, and I. Pomeranz. Transient fault recovery for chip multiprocessors. In *ISCA*, 2003.

[11] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *ISLPED*, pages 217–222, 2003.

[12] H. Hua et al. Exploring compromises among timing, power, and temperature in three-dimensional integrated circuits. In *DAC*, pages 997–1002, July 2006.

[13] E. Humenay, D. Tarjan, and K. Skadron. Impact of process variations on multicore performance symmetry. In *DATE*, 2007.

[14] Intel Corporation. Intel Core 2 Duo Mobile processor for Intel Centrino Duo Mobile technology, January 2007. Product manual.

[15] Intel Corporation. Intel Core 2 Extreme Processor X68000 and Intel Core 2 Duo Desktop Processor E6000 and E4000 sequences, April 2007. Datasheet.

[16] S. Krishnamurthy, S. Paul, and S. Bhunia. Adaptation to temperature-induced delay variations in logic circuits using low-overhead online delay calibration. In *ISQED*, pages 755–760, 2007.

[17] N. Madan and R. Balasubramonian. Power-efficient approaches to reliability. Technical Report UUCS-05-010, University of Utah School of Computing, December 2005.

[18] P. Michaud, A. Seznec, and D. Fetis. A study of thread migration in temperature-constrained multicores. *ACM TACO*, 4(2), June 2007.

[19] O. Mutlu, H. Kim, D. Armstrong, and Y. N. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54(12):1556–1571, December 2005.

[20] B. C. Paul, K. Kang, H. Kufluoglu, M. A. Alam, and K. Roy. Temporal performance degradation under NBTI: Estimation and design for improved reliability of nanoscale circuits. In *DATE*, pages 1–6, March 2006.

[21] M. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. In *ASPLOS*, pages 260–270, 2004.

[22] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream memory hierarchies. Technical report, North Carolina State University Department of Electrical and Computer Engineering, February 2002.

[23] T. Sato and I. Arita. Constructive timing violation for improving energy efficiency. In *Compilers and Operating Systems for Low Power*, pages 137 – 153, 2003.

[24] T. Shanley and B. Colwell. *The Unabridged Pentium 4*. MindShare Inc., 2005.

[25] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *MICRO*, pages 223–234, 2006.

[26] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *ASPLOS*, pages 257–268, November 2000.

[27] E. Takeda, C. Y. Yang, and A. Miura-Hamada. *Hot-Carrier Effects in MOS Devices*. Academic Press, 1995.

[28] R. Teodorescu, B. Greskamp, J. Nakano, S. Sarangi, A. Tiwari, and J. Torrellas. VARIUS: A model of parameter variation and resulting timing errors for microarchitects. In *Workshop on Architectural Support for Gigascale Integration*, June 2007.

[29] A. Uht. Achieving typical delays in synchronous systems via timing error toleration. Technical Report 032000-0100, University of Rhode Island Department of Electrical and Computer Engineering, March 2000.

[30] A. Uht. Going beyond worst-case specs with TEAtime. *Computer*, 37(3):51–56, March 2004.

[31] X. Vera, O. Unsal, and A. Gonzalez. X-Pipe: An adapative resilient microarchitecture for parameter variations. In *Workshop on Architectural Support for Gigascale Integration*, June 2006.

[32] J. Warnock et al. The circuit and physical design of the POWER4 microprocessor. *IBM JRD*, 46(1):27 – 51, 2002.

[33] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *DSN*, pages 411–420, July 2001.

[34] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT*, pages 231–242, 2005.

16th International Conference on
Parallel Architecture and Compilation Techniques (PACT 2007)
0-7695-2944-5/07 $25.00 © 2007 IEEE