Shapeshifter: Dynamically Changing Pipeline Width and Speed to Address Process Variations*

Eric Chun
Intel Corporation
Austin, Texas, USA
eric.y.chun@intel.com

Zeshan Chishti
Intel Corporation
Hillsboro, Oregon, USA
zeshan.a.chishti@intel.com

T. N. Vijaykumar
Purdue University
West Lafayette, Indiana, USA
vijay@ecn.purdue.edu

***Abstract*---Process variations are a manufacturing phenomenon that result in some parameters of the transistors in a real chip to be different from those specified in the design. One impact of these variations is that the affected circuits may perform faster or slower than the design target. Unfortunately, only a small fraction of chips speed up whereas the vast majority incur slow downs. While die-to-die variations have been addressed by clock binning, within-die variations are increasing in importance with scaling. Clock binning in the presence of within-die variations results in slow clock speeds for dies with many components that can operate at higher clock speeds. A recent paper addressing within-die variations proposes variable-latency functional units and register file so that the fast instances of these components take fewer clock cycles to operate than the slower instances. However, in pipeline stages where the instances are interdependent, the fast instances would be held up by the slow instances. Also, variable latency may complicate timing-critical instruction scheduling.**

Instead of varying the number of clock cycles, we advocate varying the clock speed. Our scheme, called Shapeshifter, maintains high clock speeds during low-ILP program phases by using a narrower pipeline of only the faster instances, and reduces the clock speed only in the high-ILP phases which use all the instances. Shapeshifter simply turns off the slow instances, removing them from any interdependence among all the instances. Also, Shapeshifter requires minimal additions to the pipeline because almost all pipelines already support varying the clock speed for power management purposes. Using simulations, we show that Shapeshifter performs better than clock binning and the variable-latency approach.

1 INTRODUCTION

Technology scaling has led to an increased number of smaller and faster on-chip transistors in successive technology generations, allowing improved microprocessor performance. However, one negative effect of technology scaling is *process variations* which are a manufacturing phenomenon that result in some parameters of the transistors in a real chip to be different from those specified in the design. These variations typically occur in parameters such as the gate length, threshold voltage, and oxide thickness. One impact of the variations is that the affected circuits may perform faster or slower than the design target. While faster circuits are the positive outcome of process variations which enables some of the chips to be clocked faster than the intended speed, the negative outcome of slower circuits lower the clock speed well below the target (e.g., as much as 30%). Unfortunately, all the critical paths in the chip have to be sped up for a faster clock whereas even one slow critical path slows down the clock. Therefore, only a small fraction of chips speed up whereas the vast majority incur slow-downs. As scaling continues, process variations are expected to increase (as transistors get smaller, variations are more likely *and* increase in relative magnitude), threatening to eat away the benefits of scaling. Therefore, it is important to address the negative impact of process variations.

Until recently, process variations have caused mostly die-to-die variations and not much within-die variations. In this scenario, transistors on one die do not deviate much from each other but may deviate from those on another die and from the target speed. Die-to-die variations have been addressed by *clock binning* which determines the clock speed of each individual die (or a set of dies) allowing faster dies to be clocked at higher speeds rather than clocking all the dies at the worst-case speed. Clock binning is a simple and effective solution; it is simple because it does not require modifying the microarchitecture or system software in any way. However, continued scaling is now resulting in increased within-die variations where some components of a die deviate from the other components and from the target speed. Because clock binning varies the clock speed of the whole die and because the slowest component determines the die's clock speed, clock binning in the presence of within-die variations would result in slow clock for dies with many components that can operate at higher clock speeds. One may think that because architects have recently stopped deepening processor pipelines, future-generation pipelines would have more slack to absorb within-die variations. However, while pipeline depths have stagnated, clock speeds continue to increase, though at a slower rate than before — at a rate allowed by faster transistors every generation. Thus the same pipeline in later generations is clocked faster. Therefore, there is no increase in the slack and process variations will continue to be a problem.

There have been some proposals to address within-die variation. Previous work [22, 33] has proposed forward body biasing (FBB) the slow components to bring them to the same speed as the faster components so that the die can be clocked faster. While the previous papers use coarse-grain FBB requiring one or two bias voltages for the whole die, a recent paper [31] proposes to apply multiple fine-grain bias voltages to different sections of a die. While coarse-grain FBB may be practical, fine-grain FBB requiring many, precisely-controlled bias voltages is hard to implement. Unfortunately,

* This work was done while Eric Chun and Zeshan Chishti were at Purdue.

while low within-die variations can be tackled by coarse-grain FBB, scaled technologies will have significant within-die variations which would require the hard-to-implement fine-grain FBB.

While FBB is a circuit-based approach, there are two architecture-based schemes to address within-die variations. The first scheme [20] proposes variable-latency functional units and register file to accommodate both fast and slow instances of these components in the same die. In this scheme, if one of the ALUs of a processor is slow then that ALU is set at testing time to operate at two-cycle latency (with optional pipelining) while the rest operate at the usual one-cycle latency. Because only one of the ALUs is slow, performance does not degrade much. However, there are two shortcomings: First, this approach works because functional units and registers have independent, redundant instances of components (e.g., ALUs) so that one instance's latency does not affect those of the others. However, in stages that have interdependent, redundant components (e.g., the select trees in the issue queue and dependence check in the renamer, as we explain later), the slowest instance slows down even the faster instances due to the interdependence (such stages are not targeted by [20]). Thus, the entire stage becomes slow and if the stage is involved in a tight pipeline loop (e.g., select-wakeup loop in the issue stage) then the stage would prevent back-to-back issue of dependent instructions causing significant performance loss. Second, while the paper carefully avoids complexity by restricting the combinations of coupling any fast register port to any fast functional unit, variable-latency functional units may complicate the instruction scheduler which is often a clock-critical component in an out-of-order issue processor. The second scheme, ReCycle [32], proposes applying the well-known circuit-level technique of time borrowing to the entire pipeline where slower stages borrow time from faster stages so that the clock speed remains high. Unfortunately, pipeline loops [4, 5, 17, 21] severely limit time borrowing, as we explain in Section 4.

Instead of varying the number of clocks, we advocate varying the clock speed. Our key idea is to keep the clock fast when the slower instances are not in use and to slow down the clock dynamically only when all the instances, including the slower ones, are used. We call our scheme as *Shapeshifter* which is based on the key observation that because instruction-level parallelism (ILP) varies within and across programs, all the instances are needed only in the high-ILP phases while fewer instances suffice for low-ILP phases. Consequently, Shapeshifter maintains higher clock speeds during the low-ILP phases by using only the faster instances (i.e., narrower pipeline), and reduces the clock speed only in the high-ILP phases which use all the instances. Shapeshifter addresses both the shortcomings of the variable-latency approach. First, because Shapeshifter turns off the slower instances in low-ILP phases, these instances are removed from any interdependence with the faster instances, allowing the faster instances to remain fast. Second, the key mechanism used by Shapeshifter is varying the clock speed, support for which is already present in almost all processors for the purpose of power management. However, Shapeshifter does add hardware to detect phases and to decide the clock speed and the appropriate number of component instances to use. Fortunately, these additions are well off the pipeline critical path.

Though Shapeshifter dynamically reconfigures the pipeline to achieve its end, Shapeshifter's novelty is not in dynamic reconfiguration which is a well-explored topic but in the idea that pipeline width can be traded off for clock speed to alleviate process variations. Having said that, there are some differences from previous reconfiguration work. Most of those papers trade-off performance for power or energy where either only one of the pipeline width [4, 26, 15, 18] or clock speed [27] is changed while the other is kept constant, or both width and clock speed are increased or decreased *together* and are not traded-off for *each other* [26]. In contrast, Shapeshifter trades-off pipeline width for clock speed. Previous work on clustered microarchitecture [6] improves performance by trading-off pipeline width (i.e., number of clusters) for communication latency (more clusters implies longer inter-cluster communication latencies). Clustering and process variations are orthogonal issues with no straightforward implications that one's solutions are applicable to the other. Because logic is much more susceptible to process variations than wires [8], trading off pipeline width for computation latency (i.e., clock speed) instead of communication latency is more appropriate for process variations. Moreover, that work does not (intend to) capture performance opportunities due to process variations across clusters. Finally, the dynamic trade-off of pipeline width for communication latency does not exist in conventional, non-clustered processors whereas Shapeshifter's trade-off is applicable to clustered as well as non-clustered processors.

Our simulation results using SPEC2000 benchmarks show that the variable-latency approach [20] improves upon clock binning by 4%, and Shapeshifter achieves a significantly better improvement of 11% over clock binning,

The rest of the paper is organized as follows. Section 2 provides a qualitative characterization of process variations. Section 3 describes Shapeshifter in detail. In Section 4 we discuss the interplay between time borrowing and pipeline loops. Section 5 describes our experimental methodology. We present our results in Section 6 and draw some conclusions in Section 7.

2 PROCESS VARIATIONS: A QUALITATIVE CHARACTERIZATION

There are two types of within-die process variations: random variations due to noise in the fabrication system and systematic variations due to spatial correlation among the transistors in a circuit as dictated by layout and lithography. Previous work [9, 1, 23] uses probabilistic models to analyze the factors affecting within-die variations. Here, we summarize the qualitative implications of relevance to architects.

The impact of within-die variations depends on the tightness of circuit timing and the length and number of critical paths in the circuit. Slack in the timing makes variations less likely to affect circuit speed. However, the impact of the length and number of critical paths is a little more involved. For long critical paths, the positive and negative random variations tend to cancel each other so that the circuit speed is affected primarily by systematic variations (i.e., spatial correlation). But for short critical paths, both random and systematic variations affect circuit speed. In such paths, the effects of random variations of the transistor threshold voltage (V_t) do not cancel out because the delay is inversely proportional to the difference between the supply voltage, V_{dd}, and V_t, causing positive V_t variations to affect the delay differently than negative V_t variations. We note that in processors with high clock speeds, pipelines are deep and therefore, critical paths are not long (e.g., around 12-14 FO4). Therefore, the case of not-long critical paths is more impor-

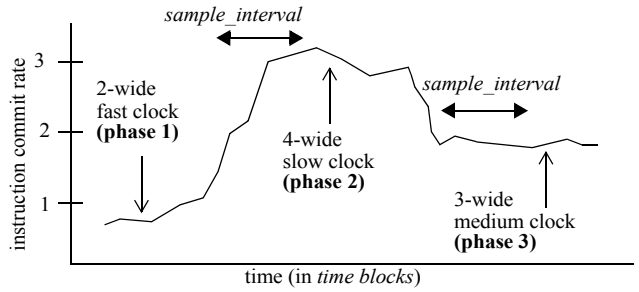

Figure 1. Shapeshifter

tant than that of long critical paths. Consequently, we model both random and systematic variations in our simulations.

Next, if the number of critical paths is large then the chances of at least one of the critical paths being affected is high (e.g., RAM and CAM circuits as in register files, rename table, and issue queue, have many critical paths because of their many wordlines and bitlines being similarly critical). Consequently, if we consider the speed distribution of the instances of the circuit under within-die variations, then the distribution shifts from the nominal target speed because every instance of the circuit is likely to be affected. Also, because almost all instances are affected similarly (i.e., the variability across the instances is small), the spread of the distribution is thin. Therefore, Shapeshifter which relies on speed variability amongst instances of components (i.e., using faster instances for high-speed clock) would not be effective for such circuits.

Finally, if the number of critical paths is small then not every instance is likely to be affected (e.g., logic circuits have fewer critical paths than RAM and CAM circuits). Consequently, the distribution does not shift as much from the nominal speed as the previous case of many critical paths. Also, because some instances are affected more than the others, the distribution spreads out more than the previous case. For such circuits, both slow and fast instances would exist and Shapeshifter can use the fast instances in low-ILP program phases to maintain high clock speed. this category of circuits includes decode, rename dependence check logic, select trees, ALUs, FPUs, structures where individual entries can be turned on or off such as individual registers (but turning off individual issue-queue entries is not straightforward, as we explain in Section 3.3). Thus, Shapeshifter can cover most of the pipeline (for the rest, we lose clock speed if the circuit falls below the nominal speed). We explain the exact coverage details in Section 3.3.

3 SHAPESHIFTER

As mentioned in Section 1, our key idea for accommodating slow instances of components is to keep the clock fast when the slower instances are not in use and to slow down the clock dynamically only when all the instances, including the slower ones, are used. We base our scheme, called *Shapeshifter,* on the key observation that because the amount of instruction-level parallelism (ILP) varies within and across programs, all the instances are needed only in the high-ILP phases while fewer instances suffice for low-ILP phases. Consequently, Shapeshifter maintains higher clock speeds during the low-ILP phases by using only the faster instances (i.e., narrower pipeline), and reduces the clock speed only in the high-ILP phases which use all the instances. Figure 1 illustrates the idea (*sample_interval,* and *time blocks* are explained later).

Shapeshifter detects phase changes by monitoring the changes in ILP. Upon detecting a new phase, Shapeshifter determines the pipeline configuration — i.e., the number of instances of each component and the clock speed — that would be appropriate for the new phase. This determination involves Shapeshifter's key trade-off of pipeline width versus clock speed. The processor remains in a pipeline configuration until a new phase is detected. To enable dynamic reconfiguration, Shapeshifter's pipeline components are modified so that slower instances can be turned off while allowing the faster instances to operate at high clock speed by removing any interdependence between the slower and faster instances. The other part of the reconfiguration is the clock speed change which is common in current processors. Clock speed change incurs PLL settling-time overhead (typically 10-20 microseconds [27, 16] or 30,000-60,000 cycles at 3 GHz). Transmeta Crusoe [14] allows discrete frequency steps of multiples of 30 MHz with 20 microseconds PLL settling time, while Intel XScale allows continuous frequency scaling.

We next describe the steps of phase detection, configuration determination, and pipeline modification.

3.1 Phase Detection

Program phases defined on the basis of ILP are characterized by two aspects: ILP magnitude (counted in instruction commit rate) and phase length (counted in number of instructions or cycles). Because we are interested in ILP changes that are large enough to warrant adding or removing instances of pipeline components, our phase detector need not detect small ILP changes. Also, because changing the clock speed incurs PLL settling-time overhead, our phase detector need not detect short phases where the PLL overhead would overwhelm the benefits of pipeline reconfiguration. Finally, because Shapeshifter detects phases at runtime, we need a scheme that is simple enough to be implemented in hardware. Previously-proposed hardware schemes for on-line phase detection monitor dynamic branch counts [5] and instruction working sets [13, 28, 15] to detect phase changes. In contrast, Shapeshifter needs to detect phases on the basis of ILP changes. Consequently, the previous schemes are not directly applicable.

Keeping these arguments in mind, we propose a phase detector that looks for a change of *ILP_delta* or more in the commit rate. By analyzing SPEC2K programs, we found that changes of about 0.5 or more in the commit rate denote a new sustained phase prompting addition or removal of component instances whereas changes of about 0.1 or less in the commit rate occur even within a phase. Accordingly, choosing *ILP_delta* from the range of 0.2-0.8 works well in practice. Such a choice acts as a low-pass filter that prevents changes smaller than *ILP_delta* from triggering pipeline reconfiguration. Because we are not interested in changes in the instantaneous commit rate, our detector samples the average commit rate of instructions over some *time block* (e.g., 100,000 cycles). Because we are not interested in short phases, a change in commit rate of just one time block is insufficient to qualify as a phase. Accordingly, the detector declares a new phase only if the commit rate of *each* of *sample_interval* consecutive time blocks changes by *ILP_delta* or more as compared to the running average commit rate of the current phase (e.g., *sample_interval* = 10). In Figure 1, we see that as the commit rate changes by *ILP_delta* or more for the *sample_interval* time blocks, phase changes are detected. By adjusting the value of *sample_interval*, we can control the length of the phases being detected.

To implement this detector, we need latches to hold the total number of instructions committed in a time block and the current phase, the number of time blocks in the current phase; and a shared divider to compute both the average commit rate per time block and the average commit rate of the current phase. The divider can be quite slow because of the large time scales involved (e.g, a division could take hundreds of cycles). As such, the divider should be optimized for area and power at the cost of speed. We also need a subtractor to determine whether the difference between the time-block average and phase average lies within *ILP_delta*. Alternatively, because the average phase length for most programs is in hundreds of millions of cycles (Section 6.1.2), all of these calculations can be done in software to avoid hardware cost. However, because phase detection occurs every 100,000 cycles though phases changes are much less frequent, there may be some overhead.

Finally we note that our phase detector merely counts and compares the number of committed instructions per cycle, does not put pressure on the clock by requiring slow operations to fit within the clock period. and is well off the pipeline's critical path.

3.2 Determining Appropriate Pipeline Configuration: Novel Trade-off of Pipeline Width versus Clock Speed

Upon detecting a new phase, Shapeshifter chooses the best configuration for this phase from among the set of configurations allowed by the pipeline.

If we assume that each pipeline stage can allow anywhere from one to its maximum number of component instances and that each stage's number is independent of those of the others, then there would be a combinatorial explosion in the number of configurations. Also, some combinations are not useful. For instance, with the asymmetric configuration of four decoders and three renamers, the front-end would be constrained by the rename stage anyway, and using more decoders would mean introducing some buffering which may complicate the front-end. To avoid the combinatorial explosion and the useless configurations, we restrict the set of configurations to a few configurations. First, within the front-end (fetch through rename) we allow only symmetric stages (i.e., all front-end stages have the same width); similarly for the back-end (issue through write-back). Second, even across the front-end and the back-end, we allow only symmetric configurations with widths of two, three, and four each (denoted as *2-2*, *3-3*, and *4-4*). We do allow one asymmetric configuration of *3-2* which does not need any extra buffering because the front-end and the back-end are already separated by the issue queue's buffering. We found that other asymmetric configurations (e.g., *4-3*, *3-4*, *2-3*) perform close to the ones in our above set, and therefore do not include them in our set. Also, we do not consider *1-1* as it performs close to *2-2*. We describe the details of implementing our configurations in Section 3.3.

To determine the best configuration, we make the key observation that the configurations differ in the clock frequency and the pipeline width (we use pipeline width in a broad sense to include front-end and back-end widths) which determines the instruction commit rate. Consequently, we would like to compare the product of the clock frequency and the commit rate (performance = commit rate x clock frequency) of the different configurations to choose the best. This comparison would allow us to perform the novel trade-off of pipeline width versus clock speed. However, the commit rate is not a simple function of the pipeline width (i.e., the relationship between the commit rate and pipeline width varies from one program or phase to another). As such, it is hard to guess the actual commit rate that will be achieved by each of the configurations in the new phase. The only exception is that if the current commit rate is extremely low (e.g., 0.1) then the commit rate will remain mostly unchanged irrespective of the configuration; such low commit rates are typically the result of high cache miss rate which does not change with pipeline width. In most other cases where the commit rate is moderate or high, the commit rate will change with the configuration. Trying out every configuration and comparing the resulting commit rates would incur performance loss by delaying the deployment of the best configuration and by incurring the PLL overhead for every configuration change. Also, performing these trials on the fly may increase complexity.

We avoid these issues by using *pre-computed gross* estimates of commit rates that would be achieved by the various configurations. We back up these estimates by a safety net of occasionally perturbing the current configuration to correct any bad configuration choices made due to inaccuracies in the estimates.

3.2.1 Pre-Computed Gross Commit Rate Estimates

We wish to pre-compute gross estimates of the typical commit rates that would be achieved by the new phase in all the allowed configurations. However, estimating the commit rate of an arbitrary phase using an arbitrary configuration is difficult if no further information were available. Fortunately, the commit rates of a given phase using different configurations fall within a small neighborhood (e.g., within 30%). Accordingly, we use the current configuration's commit rate in the just-begun new phase to locate the neighborhood where the other configurations' commit rates are likely to fall. The commit rate for the current configuration is the commit rate averaged over the last *sample_interval* time blocks at the end of which the new phase has been detected. Note that these time blocks still use the current configuration as the next configuration is yet to be decided.

However, using a single percentage for the entire range of possible commit rates, to estimate a phase's commit-rate degradation on going from a wider configuration to a narrower one, would result in loose estimates. The actual degradation depends on the amount of instruction-level parallelism in the phase. For instance, a high-ILP phase will likely lose a larger fraction of commit rate on going from *4-4* to *3-3* than a low-ILP phase; the low-ILP phase does not need a wide pipeline anyway and therefore does not degrade as much. Consequently, we partition the current commit rate into ranges (e.g., less than 0.5, 0.5-1, 1-1.5, 1.5-2, and greater than 2) and provide a separate estimate for each range. These estimates are *not specific* to any program and are based on the *gross* average commit-rate degradation over *all* SPEC2K programs (using train inputs) experienced by each of our configurations while running similar-ILP phases (i.e., their commit rates fall within the same range). We assume that the degradation is the same for all the commit rates within a range. Because our assumptions may result in inaccurate estimates for some phases, we provide a back-up safety net of periodic sampling in Section 3.2.2.

An obvious way to provide these pre-computed estimates would be by using a table indexed by the current commit-rate's range. Each row would hold the estimated commit rates and the clock frequencies for the various configurations. Table 1 shows an example

TABLE 1. UNOPTIMIZED

Commit rate Interval	Normalized Commit Rate			
	4-4	*3-3*	*3-2*	*2-2*
0.0 to 0.5				
0.5 to 1.0				
1.0 to 1.5	1.0	0.92	0.85	0.78
1.5 to 2.0				
> 2.0				

TABLE 2. OPTIMIZED

Commit rate Interval	Best configuration
0.0 to 0.5	
0.5 to 1.0	
1.0 to 1.5	3-2
1.5 to 2.0	
> 2.0	

Figure 2. Estimate table examples

with only the commit rate estimates (we show only one row for clarity). The clock frequencies are observed quantities, not estimates, and are determined at the time of testing (as explained in the beginning of Section 3.3). Then, Shapeshifter would compare the products of the estimated commit rate and the clock frequency of each configuration and choose the configuration with the best product. However, we can greatly optimize both the table and the process of choosing the best configuration. Because the commit rate estimates are known at design time and the clock frequencies at testing time, the comparison of the products in a given row can be done at testing time. Therefore, the best configuration for each row can be determined at testing time. This optimization has two implications. First, the configuration identifier can be simply looked up during execution without computing and comparing commit-rate and clock-frequency products. Second, the table becomes a small read-only structure where each row holds just the configuration identifier, and not the commit rate estimates for all the configurations. Table 2 shows the optimized version of Table 1 assuming the frequencies for *4-4*, *3-3*, *3-2*, and *2-2* are 1.0, 1.15, 1.3, and 1.35, respectively. Note that the best configuration for a commit-rate range may be different in different chips because of differences in the chips' clock frequencies for the same configuration.

It may seem that a narrow configuration may artificially reduce the commit rate, so that looking up the estimate table using only the commit rate may lead us to an incorrect row and result in us choosing a narrow configuration when a wider configuration may be better. In practice, however, our commit-rate ranges are large enough that all our configurations' commit rates fall within the same range, allowing the index to be independent of the current configuration.

3.2.2 Safety Net

As mentioned before, our estimates may be inaccurate because they are averages that are binned into ranges and may not match a given program's behavior. We found that in practice the inaccuracies are usually minor. Nevertheless to correct any bad configuration choices made due to inaccuracies, we use a safety net of periodically perturbing the current configuration (i.e., try out a different configuration). Such perturbations degrade performance if the current configuration is appropriate and needs no change, and incur the PLL overhead for changing clock speeds. Therefore, we trigger perturbations infrequently so that the PLL overhead for a perturbation and a revert back is about 0.5% of execution time (e.g., if the PLL overhead is 20 microseconds then for a 2.5 GHz clock speed, the perturbation should occur once every 20 million cycles). Fortunately, because our phases are typically hundreds of millions of cycles, our safety net despite being triggered infrequently can correct the configuration for most of the cycles in a phase.

At every safety net trigger, Shapeshifter alternately tries out a configuration that is one step narrower or one step wider than the current configuration (i.e., one row above and below the current row in the estimate table). Thus, even in the rare case that the current configuration is many steps away from the best configuration, Shapeshifter will eventually reach the best configuration. Shapeshifter compares the commit rate of the trial configuration for *sample_interval* time blocks to that of the current configuration's last *sample_interval* time blocks, and stays in the trial configuration if it is better. While our phase detection scheme uses the commit rate averaged over the entire current phase (Section 3.1), the safety net uses only the last *sample_interval* time blocks' commit rate of the current configuration. We incorporated this difference because we use the safety net not only for correcting major mistakes in our configuration choices but also for making minor adjustments to the configuration in response to any local ILP changes in the current phase. Because such changes span much fewer time blocks than a phase, averaging over all the time blocks of the current phase would overwhelm the contribution of such local changes and prevent the safety net from detecting such changes.

3.3 Enabling Dynamic Reconfiguration

The remaining issue is the detail of how the pipeline reconfigures itself. We assume that at testing time the clock speed of each chip is determined by clock binning which can be extended to determine the clock speed at which each component operates correctly (described in detail in Section 3.4). Using this information, the narrower configurations simply turn off the slower components.

3.3.1 Restricting the combinations of turned-off components across stages: reducing reconfiguration complexity

Allowing *any* combination of components across stages to be turned off (e.g., way 1 of decode, way 3 of rename, and way 2 of EX) would increase testing time (Section 3.4) and would complicate the pipeline due to a subtle issue in the routing of instructions in the narrower configurations. The fast components of all the stages may not be aligned across the stages. For instance, the fast decoders' (select trees') way numbers may be different from those of the fast renamers (functional units). Consequently, we would need shifters at the end of every stage to route instructions from a fast way in one stage to a fast way in the next stage. Because any of the ways in a stage could be fast, we would need a *n*-to-*n* crossbar at the end of every stage for an *n*-wide pipeline.

We avoid these crossbars by using an observation and by restricting the combinations of turned-off ways. We observe that because the front-end and back-end are decoupled naturally by the issue queue (i.e., instructions from any front-end way can be issued to any back-end way), there is no need for crossbars to route instructions from the front-end to the back-end. Therefore, crossbars are needed only *within* each of the front-end and back-end — e.g., between decode and rename, and between issue (select) and functional units (issue and EX are separated by register read but because we do not distinguish among register ports, our routing choice is from a fast select way to a fast functional unit).

Nevertheless, because this requirement still implies large crossbars, we propose restrictions which altogether eliminate the crossbars. In the front-end, we allow n^{th} decode way to feed only either the $n\text{-}1^{th}$ or n^{th} rename way, requiring only a 2-to-1 multiplexor at

the end of each decode way. This restriction implies that a fast decode way is not clocked fast if neither of the two allowed rename ways are as fast (even if some other rename way is as fast). The front-end requires no other modifications. In the back-end, however, *we disallow all flexibility* and restrict the n^{th} issue way to feed only the n^{th} functional unit, treating the two as an atomic unit whose frequency is the lower of the two components' frequencies (and hence the two are enabled or disabled together). We constrain the back-end more than the front-end because the back-end is more timing critical and may not have the timing slack to allow any multiplexor. We note that the front-end multiplexor is optional and removing it results in about 2% clock frequency loss.

We note that the variable-latency approach [20] adds latches to pipeline the extra latency of every slow component. Also, time borrowing [32] adds extra latches to create "nop" donor stages. Such latches may increase area and power. In contrast, because we vary the clock speed and do not deepen the pipeline except for the above 2-to-1 multiplexors, we need significantly fewer extra latches.

Finally, if a pipeline resource is shared by two pipeline ways, then we simply accept that the two ways cannot be faster than the shared resource. This approach is simple yet effective as shown by our evaluations which use 4 issue ways where a pair of pipeline ways share an FP unit (which is the most common example of resource sharing in real processors).

The above restrictions greatly simplify Shapeshifter's pipeline reconfiguration. Back-end reconfiguration amounts to a binary on/off decision for each way irrespective of the number of back-end stages, where each way can be turned off simply by not issuing any instruction to that way. Front-end reconfiguration amounts to setting the 2-to-1 muxes in the decode stage and providing the correct increment to the PC incrementor in the fetch stage (for narrow configurations, the increment is smaller than the full pipeline width).

3.3.2 Handling dependence among off and on components within a stage

While the above restriction simplifies pipeline reconfiguration, turning off a component instance in a stage raises the following issue about the other instances in the stage. A stage where the component instances are independent of each other (e.g., decode and EX), going from wider to narrower configurations and vice versa by turning off and on the slower instances is straightforward. For stages where the instances are interdependent, extra care is needed which we explain for each such stage. While the rename ways are independent in the table lookup, correcting the source tags for dependencies among the instructions fetched in the *same* cycle introduces interdependence among the rename ways (i.e., the n^{th} way may use the destination tag from any of way *1* through way *n-1*). Similarly, the select trees in the issue stage are interdependent because trees *1* through *n-1* have to communicate their selection choices to the n^{th} tree so that it does not select an instruction already selected by the other ways. In the variable-latency paper which varies the number of clocks to accommodate slower instances of components [20], a slow rename/select way would hold up the faster way due to this interdependence. To turn off a slow way in a narrower configuration so that the slow way does not hinder the faster ways' operation due to the interdependence, we modify each way to provide a NULL output whenever the way is turned off (NULL destination tag in rename and NULL select choice in issue). The NULL output is not affected by process variations because it involves no circuitry and at the same time satisfies the interdependence so that the faster ways can proceed without being held up.

In register read and write stages, we turn off slow registers in the high-clock-speed configurations by leveraging rename and removing them from the free pool of physical registers. In slow clock-speed configurations we turn on all registers. The variable-latency paper differentiates between slow and fast ports for the same register. The paper optimizes for speed by recognizing that an instruction reads two registers using two ports and each register read could be directed to the faster port for that register. In contrast, we do not differentiate between slow and fast ports for simplicity and turn off a register if any of its ports is slow. For all our configurations except *4-4* we turn off slow registers up to 20% of the registers.

The variable-latency paper applies its approach to floating-point units in addition to integer units. We apply our approach to only floating-point adders (in narrow configurations, one of our two adders is turned off if it limits the clock speed). However, we argue that floating-point multiplies and divides have enough ILP or are rare enough that their latency can be increased by one cycle to create timing slack to absorb the effects of any variations. We found that this extra one cycle results in less than 0.5% degradation.

Because we focus on process variations in the pipeline, we do not address process variations in caches. We assume that caches use other schemes such as [19, 2] to handle variations so that caches run at the nominal speed. In addition, there are a few pipeline stages that we do not address in our paper, such as the wakeup logic in the issue queue and the load-store queue. Wakeup can become slow due to a slow issue-queue entry. However, while we turn off slow registers by leveraging register renaming, turning off an issue-queue entry is not easy because there is no rename for issue. Furthermore, issue queue insertion and compaction would no longer be simple and sequential, and would need to skip over slow entries. While we do not rule out the possibility that these problems can be solved, we simply assume that variations in wakeup are not addressed. Apart from wakeup, we do not address the load-store queue whose relative area is small enough that the chance of being affected by process variations is low. However, unlike caches we do not assume that wakeup and the load-store queue operate at the nominal speed. Instead, we assume that they are affected by variations and that they constrain the clock speed for *all* our configurations.

We end this discussion with an important yet subtle difference between [20] and Shapeshifter. The cause of this difference is that issue associates a static priority order among the select trees. This priority is hard-wired into the select trees in which a lower-priority tree is prevented from selecting an instruction already selected by a higher-priority tree. This static prioritization greatly simplifies the arbitration among the select trees. The select tree order imposes an ordering among the functional units and register ports because each select tree is tied statically to a functional unit which is tied statically to a pair of register ports. Thus, if there are n instructions to issue ($n <$ maximum issue width) then the instructions are issued to the top n units in the priority order. In the variable-latency paper, this priority order implies that instructions would be issued to the highest-priority unit even if the unit happens to be slow while other faster units remain idle (e.g., in low-ILP cycles where fewer instructions than the maximum width are issued). In contrast, Shapeshifter turns off the slow units in the narrower configurations even if these units have a higher priority than the faster units so that

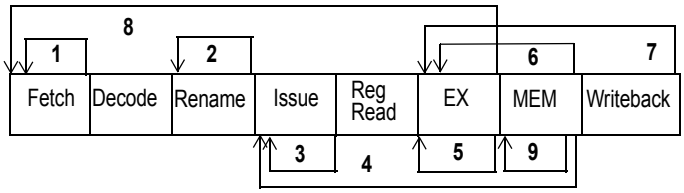

Figure 3. Pipeline loops

only the faster units are used in low-ILP phases (because of our back-end routing restriction, a select way and the corresponding functional unit are turned on or off together). While turning off a slow unit is easy, dynamically assigning low priority to whichever units happen to be slow (known only at test time) is hard to fit in fast clocks (which is why static prioritization is used).

3.4 Testing

As mentioned in Section 3.3, we assume that at testing time the clock speed of each chip is determined by clock binning which can be extended to determine the clock speed for correct operation of each component. A typical test involves scanning in a test pattern, clocking at a given speed for some cycles, and scanning out the result for comparison against the gold standard. While testing for each *individual* component's speed would considerably increase testing time requiring more testers to maintain the production rate, the pipeline reconfiguration restrictions from Section 3.3.1 greatly reduce Shapeshifter's testing combinations and thereby testing time. Because our configurations run *entire* (front-end or back-end) way at the same clock speed, we need to test the speeds of *entire ways* instead of various *combinations of individual components*. Furthermore, only a few speeds need to be tried similar to conventional speed binning (the range of speeds is pruned based on the idle leakage current in IDDQ test). While multiple ways can be tested in parallel for the same clock speed (parallel testing is common today: scan in patterns for all the ways and clock the ways in parallel, and scan out results), such testing verifies that each way works correctly in isolation but does not cover the circuits involved in interdependence among the ways (e.g., inter-way bypass circuits or the interdependence circuits in renaming and select logic described in Section 3.3.2). To cover the interdependence circuits at the few test speeds, we need to run additional tests. Because the number of pipeline configurations is only a few (e.g., *4-4, 3-3, 3-2,* and *2-2*) and because only the interdependence circuits need to be tested in the additional tests, the additional testing will likely add only a little to the overall testing time. However, all our tests — of front-end and back-end ways and of interdepedence circuits — can use existing scan chains and do not need any additional hardware.

4 INTERPLAY BETWEEN TIME BORROWING AND PIPELINE LOOPS

Tiwari et al. [32] propose ReCycle to address process variations by using the well-known time borrowing technique in the whole pipeline where fast stages donate time to slow stages. The recipient stage's clock is skewed to arrive later while the donor stage's clock is skewed to arrive earlier so that the overall clock speed is not slowed down.

We make two key observations about time borrowing. The first is that there is an interplay between time borrowing and pipeline loops [4, 5, 17, 21] which, shown in Figure 3, carry information involved in a dependence. ReCycle assumes that "time borrowing can be applied as long as the donor and recipient stages are in the same loop". This condition is unclear and *may be* insufficient in nested loops (most loops are nested, as seen in Figure 3). As stated, the condition allows a recipient stage from an inner loop and a donor stage from an outer loop because both stages are in the same outer loop. However, because the donor stage is not in the recipient's inner loop, such borrowing may not satisfy timing for the loop-back path of the inner loop. For instance, in Figure 3, borrowing from *Reg Read* to donate to *Issue,* which are both in loop *8*, may not satisfy timing for loop-back path *3*. While producer instructions exit *Issue* at a rate slower than the nominal clock rate due to borrowed time, information for consumer instructions would have to enter *Issue* at the nominal rate, which is impossible.

As such, time borrowing can be applied only if the set of loops enclosing the recipient is a subset of the set of loops enclosing the donor. This condition is more stringent than that of ReCycle which *seems* to allow borrowing if some, but not all, of the loops enclosing the stages is the same (i.e., if the two sets have a non-null intersection). Our condition implies that no borrowing can occur from outside a loop to inside the loop, which, in turn, implies that time borrowing cannot benefit single-stage loops (e.g., loops *2*, *3*, and *5*) because borrowing would be limited from the stage to itself which is pointless. ReCycle avoids this problem by using a pipeline model where single-stage loops take one cycle for the stage and another for loop-back wire delay. In this model, time slack from the short loop-back wires can be donated to the stage. However, this model precludes back-to-back issue of dependent instructions, a key requirement for high instruction commit rate in the face of the frequent dependencies in sequential code. As such, high-performance pipelines cram the delays of both the stage and the loop back wire in one cycle. These tight single-stage loops often limit the clock speed, and when affected by variations slow down the clock. Consequently, time borrowing cannot alleviate a major part of the problem, as we briefly show in Section 6.1.1. However, if future pipelines give up back-to-back issue for some reason then time borrowing may be feasible in single-stage loops.

Our second observation about time borrowing is that it requires the recipient stage to be wave-pipelined, as also briefly noted by ReCycle. Because the latency of the recipient stage is longer than a clock period, instructions enter the stage at a rate faster than that at

TABLE 3. SYSTEM PARAMETERS

Decode Width	4
Issue Width	4 Int + 2 FP
Issue Queue	32-entry Int, 16-entry FP
Reorder Buffer	128 entries
LSQ	64 entries
Physical Registers	256
L1 i-cache	64KB, 2-way, 32Bblock, 2-cycle
L1 d-cache	64KB, 2-way, 32Bblock, 2-cycle
L2 cache	2 MB, 8-way, 128B block, 13-cycle
Functional Units	4 Int + 2 FP
Branch Predictor	2-level hybrid, 8K entries
Mispredict Penalty	9 cycles
Memory Latency	300 cycles

TABLE 4. BENCHMARKS

Program	IPC	Program	IPC	Program	IPC	Program	IPC
mcf	0.1	applu	0.7	equake	1.3	eon	1.8
ammp	0.3	mgrid	0.7	apsi	1.4	mesa	2.0
art	0.4	twolf	0.9	facerec	1.5	crafty	2.1
lucas	0.4	parser	0.9	bzip	1.6	fma3d	2.3
swim	0.4	galgel	1.0	vortex	1.7	sixtrack	2.5
perlbmk	0.6	gap	1.0	gcc	1.7		
vpr	0.6	wupwise	1.1	gzip	1.7		

which they are processed by the stage, requiring more than one instruction to be present simultaneously (i.e., wave-pielined) in the stage. Because *any* stage may end up being a recipient, all stages have to be designed to be amenable to wave pipelining. This requirement, however, is hard to satisfy in practice because wave pipelining requires timing control that is significantly more precise than normal pipelining. In fact, process variations introduce timing uncertainties making wave pipelining even harder. In view of these limitations, we take the route of changing the pipeline width and speed to address process variations.

5 METHODOLOGY

5.1 Architectural Simulation

We extend SimpleScalar 3.0 [4] to simulate a high-performance, out-of-order processor in 50 nm technology. Table 3 shows the baseline parameters for the simulated processor, which are similar to those of the Alpha 21264 [9]. Shapeshifter addresses decode, rename, select, register file, and functional units (all the relevant stages in both integer and floating-point pipelines). Recall from Section 3.3 that though Shapeshifter does not address wakeup and LSQ, we do account for variations in these components in deciding the clock speed for each configuration. Like [19], we found LSQ to be the least affected component due to its small area. As mentioned in Section 3.3, we assume that caches use other schemes such as [19, 2] to handle variations so that caches run at the nominal speed.

Table 4 shows the benchmarks and their IPCs for the baseline processor. We evaluate the entire SPEC2K suite. For each program, we use *ref* inputs, fast-forward the first 2 billion instructions and then execute the next 2 billion instructions. We do not use SimPoints [12] because the 100-million instruction intervals prescribed by SimPoints are not long enough to capture the phase behavior of the programs.

To generate the commit rate estimate tables (Section 3.2.1), we profile the benchmarks using *train* inputs to obtain the average IPCs for our pipeline configurations. The estimates are shown in Table 5 in which the rows represent IPC ranges while the columns show the average IPC values for each configuration normalized to the 4-4 baseline. By combining these numbers with the configurations' clock frequencies (known at testing time for each chip), we find the best configuration for each IPC range for each chip.

5.2 Modeling Process Variations

We model both the random and systematic components of on-die variations for each stage of the pipeline. We use projections from [9] to estimate the total number of critical paths (N_{cp}) in the simulated processor. Based on real processor data, the projections estimate N_{cp} to be 1000 (including L1 and L2 caches) for 100 nm technology. Assuming that the number of transistors per processor and the processor pipeline depth would keep increasing with technology generations, the projections suggest that N_{cp} would increase beyond 1000 for post-100-nm generations. However, we note that the diminishing returns from ILP, increased complexity of scaling processor resources, and increased power dissipation has led to chip multiprocessors where processor complexity (both the number of transistors and pipeline depth) has stopped scaling in post 100-nm generations. Consequently, we assume N_{cp} to stay unchanged at 1000 from 100 nm to 50 nm technology. N_{cp} for each pipeline stage is assumed to be proportional to the number of transistors in that stage [21, 19]. We further assume that the number of transistor per stage is roughly proportional to the area of the stage. This assumption is common and reasonable because high-performance processors are area-optimized and do not have much area slack. We calculate the N_{cp} for each stage by using the area estimates from the Alpha 21264 floorplan file in Hotspot [29]. We distribute each stage's N_{cp} into individual components within that stage (such as between wakeup and select in issue queue) on the basis of area breakdown. We assume a pipeline depth of 13 FO4, consistent with the 3 GHz clock speed for the current 70 nm technology.

We obtain the following parameters for process variations from the ITRS 2006 roadmap [3]. Like [21, 19, 20, 32], we assume equal contribution from systematic and random variations in gate length (L_{eff}), thus keeping $\sigma_{rand}/\mu = \sigma_{sys}/\mu = 0.03$. We capture the effect of gate length on the (random and systematic) threshold voltage (V_t) variations using the gate-length-to-delay curves from [20, 12]. In addition, we account for the effect of random dopant variations on V_t by modeling V_t as a normal random variable, similar to [2, 20]. We set the nominal V_t for 50 nm technology to be 300 mV and its variability (3σ) to be 42% *exactly* as per the ITRS 2006 roadmap for 45 nm [3]. (V_t variability is much higher than L_{eff} variability as shown in the ITRS roadmap.) To simplify our analysis, we ignore the spatial correlation effects of systematic variations and assume that all the components within a stage are equally affected by systematic variations (this correlation is too weak across stages to have much impact). This assumption is conservative because more accurately varying correlation with distance would enhance within-stage component variability, creating more opportunity for Shapeshifter.

Using the above-mentioned parameters, we generate 100 test chips, each with a unique profile for gate lengths and threshold voltages. We then use the delay curves to calculate the critical path delays in each pipeline stage. Finally, we use the FMAX methodology [9] to obtain the frequency of each pipeline configuration for each individual chip. We found that increasing the number of chips beyond 100 does not have a significant impact on the average clock frequencies for different configurations. Consequently, we use 100 chips in our architectural simulations.

TABLE 5. COMMIT RATE ESTIMATE TABLE

IPC Interval	Normalized IPC			
	4-4	3-3	3-2	2-2
0.0 to 0.5	1.00	0.96	0.93	0.91
0.5 to 1.0	1.00	0.95	0.91	0.87
1.0 to 1.5	1.00	0.92	0.85	0.78
1.5 to 2.0	1.00	0.89	0.79	0.73
> 2.0	1.00	0.83	0.69	0.63

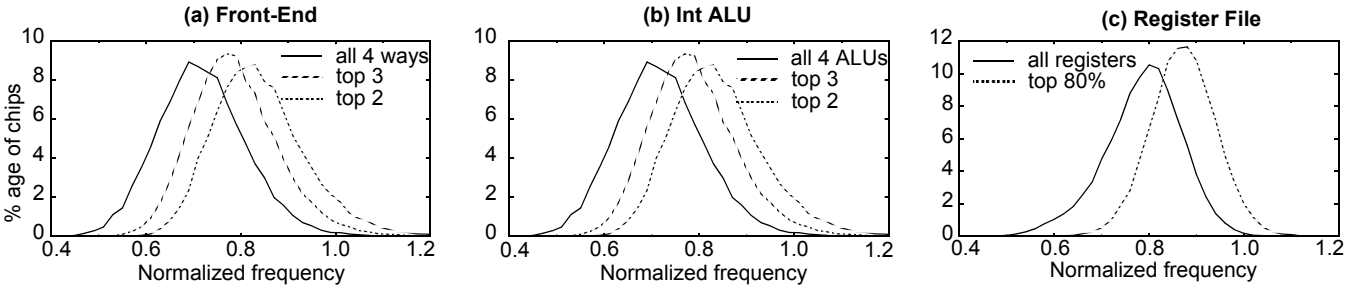

Figure 4. Frequency distribution of pipeline stages for different configurations

6 RESULTS

Section 6.1 analyzes the frequency of Shapeshifter's configurations. Section 6.2 compares Shapeshifter with other schemes in terms of performance. Section 6.3 shows the time breakdown of pipeline configurations for different programs and Section 6.4 analyzes the sensitivity of Shapeshifter to phase detection thresholds.

6.1 Analysis of Frequency and Phase Behavior

6.1.1 Pipeline frequencies

In this section, we analyze the frequency distribution of Shapeshifter's pipeline configurations. Because the maximum operable frequency of the pipeline depends on the operable frequencies of the individual stages. we first discuss the frequency distributions for the individual stages.

Figure 4 (a through c) shows the frequency distribution for the front-end (*decode* + *rename*), *integer ALU* and *register file* stages, respectively. In the interest of space, we do not show distributions for all the pipeline stages. We obtain these distributions by simulating process variations in 10,000 randomly-generated chips (while we need 10,000 chips to capture the *distribution*, the 100 chips mentioned in Section 5.1 are enough to capture the *average*). For each graph, the X-axis represents frequencies normalized to the nominal frequency under no process variations. The Y-axis shows probability distributions of the stage's maximum operable frequencies as percentage of total number of chips. The maximum frequency is that of the stage and not of the whole chip. For each stage, we show multiple distributions — one with all the instances of the stage's components, and others with a fastest subset of instances (e.g., top 3 and top 2 instances in Figure 4(a,b) and top 80% in Figure 4(c)).

The frequency distribution for each stage shifts away from the nominal frequency as we increase the number of instances in the stage to include slower instances. For example, in going from 2-way to 4-way front-end, the mean frequency decreases from 0.83 to 0.7. Note that because we show *fastest* subsets and not *random* subsets, our distributions get narrower with fewer instances instead of wider as would be the case with random subsets (as discussed in Section 2 and shown in [9]).

Using the frequency distributions for individual pipeline stages and the FMAX model, we show the frequency distribution for the entire processor in Figure 5. Like Figure 4, the X-axis in Figure 5 represents frequencies normalized to the nominal frequency under no process variations and the Y-axis shows probability distributions of maximum operable frequencies in terms of percentage of total number of chips. We show separate distributions for our base configuration of clock binning, the variable-latency scheme (*VarLat*) from [20], *3-3*, *3-2*, and *2-2* configurations. Note that clock-binning's frequency, like *4-4*'s, is constrained by the slowest pipeline component and thus *4-4* (not shown) and clock binning have identical distributions. Similar to [20], we set the slower functional units and the slowest 20% of registers to take one extra cycle in *3-3*, *3-2*, and *2-2*. Based on the probability distributions of Figure 5, Table 6 shows the mean frequencies for the various schemes and pipeline configurations.

The narrower Shapeshifter configurations exhibit significant frequency improvements over clock binning. On average, the normalized frequencies of clock binning, *3-3*, *3-2*, and *2-2* are 0.65, 0.74, 0.77, and 0.79, respectively (Table 6). Thus, *3-3*, *3-2*, and *2-2* achieve 13%, 18%, and 21% improvements in frequency over clock binning, respectively. As discussed in Section 3.2, the frequency advantage for narrower configurations is due to the shutdown of slower pipeline components.

VarLat shows only 6% frequency improvement over clock-binning (Table 6). *VarLat's* improvement is less than that of 3-3 because *VarLat* does not address variations in *rename*, *integer issue*, and *FP issue*, and these stages constrain *VarLat's* frequency (we assume that caches do not constrain *VarLat's* frequency, as we do for Shapeshifter). In contrast, *3-3* shuts down slower components in these stages. Note that our *VarLat* results are different from those in [20] in which the authors assume that components other than the functional units and register file do not have variations, overstating the speedups achieved by addressing only these components.

For each of the configurations, the number of chips operable at or beyond the nominal frequency is negligible. Accordingly, the mean frequencies for clock binning, *3-3*, *3-2*, and *2-2* are 35%, 26%, 23%, and 21% worse than the ideal case of no process variations (Table 6). These numbers show the severity of the problem. Specifically, the frequency for even the narrowest configuration is

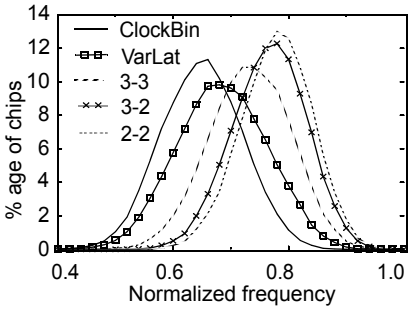

Figure 5. Processor frequency distributions

Config	Mean Frequency
NoVar.	1.0
ClockBin	0.65
3-3	0.74
3-2	0.77
2-2	0.79
VarLat	0.69
ReCycle	0.67

TABLE 6. MEAN FREQUENCIES

considerably less than the nominal value. This observation suggests that any technique (including Shapeshifter), barring for a no-variation process, is likely only to mitigate the problem and not solve it entirely. However, by mitigating the problem, Shapeshifter tries to prevent process variations from eating away the benefits of scaling.

The frequency distribution of *3-2* is closer to *2-2* than to *3-3*. This trend arises because the back-end has more critical paths than the front-end and thus the back-end limits the frequency more often than the front-end. As *3-2* is similar to *2-2* but different from *3-3* in terms of the back-end, the frequency distribution of *3-2* is closer to that of *2-2* than to that of *3-3*.

In the case of ReCycle [32], our simulations show that pipeline stages with single-stage loops (rename, issue, and EX) limit the clock speed in 70% of the simulated chips. Because time borrowing cannot alleviate such loops (Section 4), ReCycle achieves an average frequency improvement of only 3.1% over clock binning (Table 6), even under the ideal assumption that enough slack is available for time borrowing to make the other stages non-critical in the remaining 30% chips. We do not show ReCycle in Figure 5 because it is close to clock binning.

While frequency differences between configurations create opportunity for Shapeshifter, switching configurations according to phases is desirable only if the phases are long enough so that the performance benefit of phase switching amortizes the PLL overhead. Next, we characterize our programs based on phase behavior.

6.1.2 Phase characterization

Table 7 shows the phase behavior of the programs. For each program, we show the number of phases and the average phase length in terms of millions of cycles for the baseline *4-4* configuration. We use *sample_interval* = 10 and *ILP_delta = 0.5* for phase detection.

Most of the programs have a small number of long phases. 9 of the 26 programs have only one phase while 12 of the remaining 17 programs have average phase lengths of more than 100 million cycles. In contrast, the PLL overhead is ~ 20 microseconds [27, 14] or 60000 cycles for a 3 GHz clock frequency. Thus, program phases are usually long enough to amortize the PLL overhead.

TABLE 7. PHASE CHARACTERIZATION

Program	No. of Phases	Avg. Phase Length (10^6 cycles)	Program	No. of Phases	Avg. Phase Length (10^6 cycles)
mcf	1	18250.0	wupwise	7	252.0
ammp	8	938.0	equake	6	249.2
art	1	5389.0	apsi	10	140.4
lucas	1	5226.9	facerec	209	6.4
swim	1	5090.9	bzip	23	54.7
perlbmk	9	362.1	vortex	11	109.7
vpr	12	287.6	gcc	3	382.5
applu	20	140.5	gzip	1	1170.0
mgrid	21	133.6	eon	1	1120.2
twolf	1	2195.7	mesa	17	58.5
parser	98	21.6	crafty	1	975.6
galgel	222	8.8	fma3d	1	856.2
gap	13	149.3	sixtrack	2	405.5

6.2 Performance

In this section, we compare the performance of Shapeshifter against other schemes and analyze the relationship between Shapeshifter performance and the amount of ILP in each program.

Figure 6 compares the performance of variable-latency [20] (*VarLat*), *3-way*, Shapeshifter, and *perfect-phase* schemes. The X-axis in the graph lists the benchmarks in order of increasing ILP to show performance trends. The performance of each scheme is given in terms of Instructions Per Second (IPS) normalized to the IPS of the base clock-binning scheme. *VarLat* uses one extra cycle for slow components as described in Section 6.1.1 and Shapeshifter uses *sample_interval* = 10 and *ILP_delta = 0.5*, as before. We explain *3-way* and *perfect-phase* next. One simple option to address variations is to achieve higher clock speeds by statically (i.e., permanently) turning off the slower components. This option is a competing alternative to Shapeshifter and shows the need for Shapeshifter's dynamic reconfiguration. *3-way* is such a scheme that always stays in the *3-3* configuration. Note that *3-way* has the same frequency as *3-3* in Table 6. *perfect-phase* is a hypothetical scheme which provides an upper bound on Shapeshifter's performance. While inaccuracies in commit-rate estimates may cause Shapeshifter to switch to a sub-optimal configuration for a program phase, *perfect-phase* uses a priori knowledge to accurately decide the optimal configuration for each program phase. Furthermore, unlike Shapeshifter, *perfect-phase* instantaneously switches to the optimal configuration without incurring any PLL overhead.

The rightmost bars of Figure 6 show the normalized mean IPS for different schemes. Caution is in order while looking at these means. The normalized mean for a scheme is not the harmonic mean of all the other bars shown in the graph for that scheme, but rather it is the harmonic mean of that scheme's IPS (over all the benchmarks) divided by the harmonic mean of clock-binning's IPS.

Shapeshifter significantly outperforms clock binning and improves performance over *VarLat* and *3-way* in most of the benchmarks. On average, *VarLat*, *3-way*, and Shapeshifter have speedups

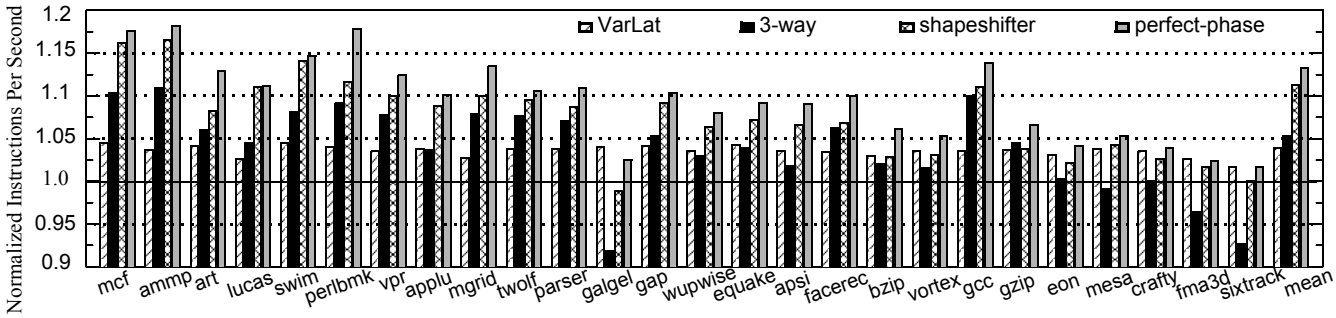

Figure 6. Performance comparison

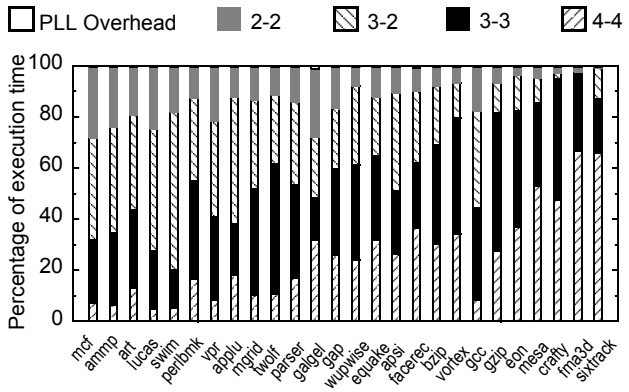

Figure 7. Execution time breakdown

of 3.9%, 5.3%, and 11%, respectively, over clock-binning. *VarLat* performs worse than Shapeshifter in most of the benchmarks because *VarLat's* frequency is significantly worse than those of the narrower Shapeshifter configurations, as shown in Table 6. Also, these configurations are better than *VarLat* in giving higher priority to faster functional units, as discussed in Section 3.3.2.

While there are some exceptions, the performance gains of Shapeshifter over both clock binning and *VarLat* generally increase as the amount of ILP in benchmarks decreases (towards the left end of the graph). This trend occurs because narrower and faster Shapeshifter configurations lose less IPC in low-ILP programs than in high-ILP programs. As expected, the two lowest ILP benchmarks, *mcf* and *ammp*, show the maximum improvements of 16% and 17% over clock-binning, respectively.

While *3-way* suffers IPC loss in comparison with clock-binning, the net performance of *3-way* is better than that of speed-binning in most of the programs. We found that going from clock binning to *3-way* decreases the harmonic mean of the IPC by 8%. However the 13% frequency advantage of *3-way* over clock-binning (Table 6) results in a net speedup of 5%. This speedup is higher than that of *VarLat* which, unlike *3-way*, is not applicable to all pipeline stages that may be susceptible to process variations. However, *3-way*'s speedups being considerably lower than those of Shapeshifter highlight the advantage of Shapeshifter's dynamic approach.

3-way's speedup may suggest that *3-way* should be preferred over clock binning. However, *3-way* performs significantly worse than clock-binning in 4 high-ILP benchmarks (*sixtrack*, *fma3d*, *mesa*, and *galgel*). These programs require all the pipeline components due to their high ILP. In contrast to 3-way, Shapeshifter dynamically switches to the 4-4 configuration in these benchmarks to perform close to or better than clock-binning.

Shapeshifter performs reasonably close to *perfect-phase* in most of the benchmarks. On average, Shapeshifter performs within 1.7% of *perfect-phase***.** The performance gap is due to imperfections in choosing the best configurations and the PLL overhead. Benchmarks with frequent short phases (such as *facerec* and *galgel*, as seen in Table 7) exhibit larger performance gaps. There are two main reasons for this trend: (1) PLL overhead is relatively more substantial in short phases and may overwhelm the performance advantage of configuration switches. (2) The safety net mechanism is rarely invoked for short phases, thus preventing any bad configurations from being corrected. Nevertheless, the performance gap is still small indicating the high accuracy of the estimate table.

6.3 Execution Time Breakdown

In this section, we analyze Shapeshifter's ability to adapt the pipeline width according to program behavior. We show execution time breakdowns for different programs and relate them to the programs' ILP. We expect narrower (wider) pipeline configurations to dominate the execution time of lower-ILP (higher-ILP) programs.

Figure 7 shows the time spent by programs in different Shapeshifter configurations. Like Figure 6, the X-axis lists the benchmarks in the order of increasing ILP. For each program, going from bottom to top, the portions of the stacked bar represent the times spent in *4-4*, *3-3*, *3-2*, *2-2*, and PLL overhead, respectively, as percentages of execution time for the runs shown in Section 6.2.

For low-ILP benchmarks (at the left end of the graph), execution time is dominated by *3-2* and *2-2*. By adapting to narrower configurations for low-ILP programs, Shapeshifter is able to achieve significant performance advantages over clock-binning (Figure 6).

Going from low-ILP to high-ILP programs (from left to right in Figure 7), the relative contribution of *4-4* to the runtime increases and that of *2-2* (and *3-2*) decreases. For high-ILP benchmarks (such as *fma3d* and *sixtrack*), most of the time is spent in *4-4*, as expected.

It is worth noting that most of the benchmarks spend more time in *3-2* than in *2-2*. This trend arises because *3-2*'s frequency is very close to that of *2-2* (Table 6) and *3-2* has an IPC advantage over *2-2*, making *3-2* usually better than *2-2*, even in low-ILP programs.

Because of long phases, the time lost due to the PLL overhead is negligible. Averaging across all the benchmarks, the PLL overhead is only 0.6% of the total execution time. Also, recall from Section 3.2.2 that the safety net is triggered only every 20 million cycles, thereby limiting the safety net's PLL overhead to 0.5%. Only two programs, *galgel* and *facerec* incur PLL overheads of more than 1%. As shown in Table 7, both these programs have large number of phases, resulting in more frequent PLL switching.

Finally, though some programs have just one phase (e.g., *mcf* and *crafty* in Table 7), they use multiple configurations (Figure 7), because of the following two reasons: (1) Figure 7 shows *average* distribution over *all* our simulated chips where the optimal configuration for a given phase may be different from one chip to another (because the clock speed differences may work out that way). (2) Our safety-net's perturbations can result in multiple configurations.

6.4 Sensitivity Analysis

We analyze the sensitivity of Shapeshifter's performance to phase detection thresholds and the use of safety net. We vary *ILP_delta* as 0.3, 0.5 (default), and 0.7, and *sample_interval* as 10 (default), 30, and 100. We found that performance is mostly insensitive to these variations (within 2% of each other across all programs). To isolate the impact of the safety net, we run Shapeshifter with and without the safety net. On average, removing the safety net degrades performance by only 0.2%. However, for *perlbmk,* removing the safety net results in 6% degradation. The commit rates for some phases in *perlbmk* are significantly different from the estimates of Table 5, resulting in sub-optimal configuration choices. Removing the safety net leaves these choices unchecked.

7 CONCLUSIONS

Process variations result in a vast majority of microprocessor chips to perform slower than the design target and are a serious

impediment to improving performance through technology scaling. While clock binning is effective against die-to-die variations, applying clock binning to chips with within-die variations results in slow clock for dies with many components that can operate at higher clock speeds. We proposed a novel scheme, called Shapeshifter, which maintains high clock speeds during low-ILP program phases by using a narrower pipeline of only the faster instances of components, and reduces the clock speed only in the high-ILP phases which use all the instances. Shapeshifter is based on the key observation that because instruction-level parallelism (ILP) varies within and across programs, all the instances are needed only in the high-ILP phases while fewer instances suffice for low-ILP phases. While previous approaches are limited to a few stages of the pipeline, Shapeshifter can be applied to the entire pipeline. Also, Shapeshifter requires minimal additions to the pipeline because almost all pipelines already support varying the clock speed for power management purposes. Our simulation results using SPEC2000 benchmarks showed that while the previous variable-latency approach achieves 4% performance improvement over clock binning, Shapeshifter achieves a significantly better improvement of 11%.

REFERENCES

[1] A. Agarwal, D. Blaauw, and V. Zolotov. Statistical timing analysis for intra-die process variations with spatial correlations. In *Proceedings of the 2003 International Conference on Computer-aided Design*, pp. 900–907, 2003.

[2] A. Agarwal, B. Paul, H. Mahmoodi, A. Datta, and K. Roy. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE Transactions on Very Large Scale Integration Systems*, 13(1):27–38, 2005.

[3] S. I. Association. International technology roadmap for semiconductors, 2006.

[4] R. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 218–229, 2001.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 245–257.

[6] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 275–287, 2003.

[7] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Eighth International Symposium on High Performance Computer Architecture*, pp. 299–310, 2002.

[8] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *Keynote speech: 37th Annual International Symposium on Microarchitecture*, 2004.

[9] K. Bowmann, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid State Circuits*, 37(2):183–190, 2002.

[10] D. Burger and T. Austin. The SimpleScalar Tool set, version 2.0. Technical report, University of Wisconsin-Madison, 1997.

[11] Z. Chishti and T. N. Vijaykumar. Wire delay is not a problem for SMT (in the near future). In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 40–51, 2004.

[12] U. B. Device Research Group. *BPTM homepage*. http://www-device.eecs.berkeley.edu/ ptm/mosfet.html.

[13] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 233–244, 2002.

[14] M. Fleischmann. Longrun power management. Technical report, Transmeta Corporation, 2001.

[15] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 157–168, 2003.

[16] T. Kan, G. Leung, and H. Luong. A 2-V 1.8-Ghz fully integrated CMOS dual-loop frequency synthesizer. *IEEE Journal on Solid State Circuits*, 37(8):1012–1020, 2002.

[17] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[18] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pp. 81–92, 2003.

[19] X. Liang and D. Brooks. Microarchitecture parameter selection to optimize system performance under process variation. In *Proceedings of the 2006 International Conference on Computer-aided Design*, pp. 429–436, 2006.

[20] X. Liang and D. Brooks. Mitigating the impact of process variations on processor register files and execution units. In *Proceedings of the 39th International Symposium on Microarchitecture*, pp. 504–514, 2006.

[21] D. Marculescu and E. Talpes. Variability and energy awareness: a microarchitecture-level perspective. In *Proceedings of the 42nd Annual Conference on Design automation: DAC'05*, pp. 11–16, 2005.

[22] S. Narendra, A. Keshavarzi, B. Bloechel, S. Borkar, and V. De. Forward body bias for microprocessors in 130-nm technology generation and beyond. *IEEE Journal of Solid State Circuits*, 38(5):696–701, 2003.

[23] M. Orshansky, L. Milor, P. Chen, K. Keutzer, and C. Hu. Impact of systematic spatial intra-chip gate length variability on performance of high-speed digital circuits. In *Proceedings of the 2000 International Conference on Computer-aided Design*, pp. 62–67, 2000.

[24] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, 1997.

[25] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 244–255, 2003.

[26] R. Sasanka, C. Hughes, and S. Adve. Joint local and global hardware adaptations for energy. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 144–155, 2002.

[27] G. Semeraro, et al. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pp. 29–40, 2002.

[28] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 336–349, 2003.

[29] K. Skadron, et al. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 2–13, 2003.

[30] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 25–36, June 2002.

[31] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Mitigating parameter variation with dynamic fine-grain body biasing. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, pp. 27–42, 2007.

[32] A. Tiwari, S. Sarangi, and J. Torrellas. Recycle: Pipeline adaptation to tolerate process variation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 323–334, 2007.

[33] J. W. Tschanz, et al. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid State Circuits*, 37(11):1396–1402, 2002.