Chapter 4

MICRO-ARCHITECTURE DESIGN AND CONTROL SPECULATION FOR ENERGY REDUCTION

Dirk Grunwald
Dept. of Computer Science
University of Colorado
Boulder, CO, 80309-0430
grunwald@cs.colorado.edu

Abstract Conventional wisdom states that the best way to design an energy-efficient microprocessor is to design it for high performance, since a high performance processor will complete a task quicker than an energy-conscious design. However, our research group has found ways to reduce energy without impacting performance by controlling the amount of speculation used by the processor in its quest for performance.

1. Introduction

As the issues and concerns facing power consumption grow to include high performance microprocessors as well as the pervasive hand-held and wireless computing market, novel approaches to power conservation at the architectural level are being sought. In the same vein, ubiquitous computing is becoming more prevalent, and thus the performance demands made upon these devices is increasing, leading toward more complex microarchitectures.

Although much work on energy reduction has taken place in the circuit and device technology domains [3], there has been an increasing emphasis on *designing* for power efficiency at the architectural level. This implies that the desired energy and power goals must be targeted early in the design cycle and that the system microarchitecture must work in concert with advances in circuit technology to reduce power demands. In this chapter, we describe some architectural techniques appropriate for reducing power and energy in processor designs; we will focus on designs appropriate for reducing *energy*. There is less challenge in reducing power, since power doesn't incorporate any notion of completing a given task – for example, we could minimize the power used by a processor simply by turning it off or running it very slowly. It is more difficult to reduce the *energy* needed for a computation since that reduction assumes the computation must still be completed (and, typically, completed in a timely fashion).

2. Background and Terminology

In current process technologies active power, or the power used by actively charging and discharging capacitance, dominates the total power consumed by processors. As process feature sizes decrease, static power will become a significant problem, but one that we will not discuss.

Since the active power for a CMOS circuit varies $P \propto CFV^2$, it's clear that the power can be reduced by: reducing the capacitance (C); reducing the processor frequency (F); or, reducing the voltage (V). However, we can typically only adjust capacitance and a combination of frequency and voltage to reduce *energy*. Improving program performance by reducing the number of instructions improves energy use because of a decrease in cycles, and thus optimizing algorithms and compilers are the first step to producing a truly energy efficient system [10]. Other works investigating power (and energy) reduction at the architectural level have concentrated mainly upon cache- or buffer-type structures which can be turned off during periods of inactivity [7, 24, 18]. In certain applications that must compute results at a fixed rate, such as media applications, the total execution time is driven by some exogenous factor, such as the duration of a sound or video image. In these systems, reducing the processor frequency can produce a linear reduction in energy. The power and total energy can be further reduced if some form of *voltage scaling* is available.

Speculation is a fundamental tool in computer architecture. It allows an architectural implementation to achieve higher instruction level parallelism, and thus performance, by predicting the outcome of specific events. Most processors currently implement branch prediction to permit speculative changes in the control-flow; other recent work has focused on predicting data values to reduce data dependencies [19]. When used poorly, speculation is a form of wasted work in the processor and can lead to considerable power or energy overhead We'll examine two forms of *speculation control* that attempt to trade off how aggressively speculation should be applied. The first technique uses *confidence estimation* to implement *Pipeline Gating*. Confidence estimation [14, 11] is a technique for assessing the quality of a particular prediction. Pipeline gating [20] is a form of speculation control that uses confidence estimation to reduce speculation in pipelined processors. Pipeline gating is a form of *activity reduction* – by reducing the number of instructions fetched by the processor, the total power can be reduced.

The second form of speculation control relies on *voltage scaling*. Dynamic voltage scaling reduces the power over time by varying the voltage and frequency of the entire processor; this process typically takes 10's of microseconds and is usually controlled by the operating system or BIOS layer. Another alternative is *clustered voltage scaling* [15, 25], where different parts of the processor run at different voltages. This is usually done during design time, and the results can be dramatic – Igarashi *et al* [13] applied this technique to a media processor and saw a 47% power reduction with 10% increase in area using two voltage levels. In that technique, critical path analysis is performed after the circuit is designed [25]. Circuits that are not on the critical path are said to have "scheduling slack" and are candidates for a lower voltage. The lower voltage drives the logic at a slower rate, but the voltage level is chosen to insure that the results are still computed no later than the circuits on the critical path. We will show how the "scheduling slack" can be applied to *instruction scheduling* in a processor as well as the underlying circuits. To controls speculation by slowing down instructions, executing them no earlier than necessary.

3. Pipeline Gating

We have applied confidence estimation to three areas of computer microarchitecture [20, 17, 21]; in this work, we'll focus on using confidence estimation for *pipeline gating*, which reduces the average power demands of an existing microarchitecture without reducing performance by avoiding unnecessary speculation. Before describing the how confidence estimators can be used to implement speculation control, we'll cover the basic properties of confidence estimators.

3.1 Confidence Estimators

It is more difficult to compare two confidence estimators than two branch predictors in part because confidence estimators can be used for a number of purposes while branch predictors are typically only used to predict the outcome of control-dependent instructions. Most architectures are designed to use speculation and the general assumption is that "you might as well be doing something", and thus each branch is predicted. By comparison, confidence estimators will normally be used for *speculation control*. For example, if a particular branch in a multithreading processor is of low confidence, it may be more cost effective to switch threads than speculatively evaluate the branch. A confidence predictor attempts to corroborate or assess the prediction made by a branch predictor. Each branch is eventually determined to have been predicted correctly or incorrectly. For each prediction, the confidence estimator assigns a "high confidence" or "low confidence" to the prediction. We have found that the following notation simplifies the comparison of different confidence estimators. We can draw a 2×2 matrix listing the frequency for each outcome of a test. When we apply this framework to architectural simulation, each of the quadrants can be directly measured during simulation or analysis. Typically, we normalize the values to insure that the sum equals one. Thus, our quadrant table for confidence estimation is:

		Prediction Outcome C	Prediction Outcome I
Confidence	HC	C_{HC}	I_{HC}
	LC	C_{LC}	I_{LC}

In this table, "C" and "I" refer to "correct" and "incorrect" predictions, respectively, and "HC" refers to "high confidence" and "LC" to "low confidence". During a simulation, we can measure C_{HC}, I_{HC}, C_{LC} and I_{LC} using a branch predictor for each branch and concurrently estimate the confidence in that branch predictor using a specific confidence estimator. When the branch is actually resolved, we classify the branch as belonging to class C_{HC}, I_{HC}, C_{LC} or I_{LC}.

When a confidence estimator is applied, the architectural feature using that confidence estimation will usually be used for "high confidence" *or* "low confidence" branches, but not both. For example, consider a simultaneous multithreading (SMT) processor that uses a confidence estimator to determine if a predicted branch is likely to be mispredicted. If the branch prediction is of "low confidence", the processor may switch to another available thread rather than fetch additional instructions from the current thread. The performance of such a processor is very sensitive to $P[I|LC] = I_{LC}/C_{LC} + I_{LC}$, the probability that the branch is incorrectly predicted if it was low confidence. A high value for $P[I|LC]$ indicates that the processor can switch contexts only when the following instructions will not commit. A low value of $P[I|LC]$ indicates that the SMT processor may needlessly switch threads, reducing the performance of the primary thread. A low value of the $P[LC|I]$ means that the processor will miss some opportunities to improve aggregate performance by switching threads.

Not all uses of confidence estimators will make the same kind of decisions, but we feel it is most useful to compare confidence estimators using metrics that reflect how the confidence estimators are used. For example, our work in pipeline gating relied on confidence estimators with a large $P[I|LC]$ and a large $P[LC|I]$. We have found in our own discussion that terms such as "accuracy" (*e.g.* $P[I|LC]$) and "coverage" (*e.g.* $P[LC|I]$) tend to cause confusion, because accuracy has an inherit implication about the application of a technique; thus, we have introduced a set of metrics that can be used to compare confidence estimation techniques [11]; we've also shown how confidence estimation can use existing branch predictor state and still yield good performance.

3.2 Using Confidence Estimation for Speculation Control

Much low power research has focused on reducing energy in the memory subsystem [8, 16, 10]. In embedded processors, such as the StrongArm [22], the memory subsystem is thought to be the dominant source of area and power because the rest of the processor has been simplified to reduce power. State-of-the-art microprocessors have a high degree of control complexity and a large amount of area dedicated to structures that are essential for high-performance, speculative, out-of-order execution, such as branch prediction units, branch target buffers, TLBs, instruction decoders, integer and floating point queues, register renaming tables, and load-store queues. For example, $\approx 30\%$ of the core die area on the DECchip 21264 is devoted to cache structures, while the StrongARM processor uses $\approx 60\%$ of the core die area for memory. Figure 4.1 shows a distribution of the power dissipated in a PentiumPro processor [12] during a test designed to consume the most power, which is when the processor is committing each instruction that it fetches. The fetch and decode stages, along with components necessary to perform dynamic scheduling and out-of-order execution, account for a significant portion of the power budget. It it clear that pipeline activity is a dominant portion of the overall power dissipation for many microprocessors.

Performance is the primary goal of state-of-the-art microprocessor design. Architectural improvements for performance have centered on increasing the amount of instruction level parallelism through aggressive speculation and out-of-order execution. Although these advances have increased the number of instructions per cycle (IPC), they have come at the cost of wasted work. Most processors use branch prediction for speculative control flow execution. Branch prediction is used to execute beyond the control boundaries in the code. With high branch prediction accuracy, most issued instructions will actually commit. However, many programs have a high branch misprediction rate, and these programs issue many instructions that never commit. Each of those instructions uses many processor resources. If we can decrease the percentage of uncommitted instructions actually issued, we can decrease the power demands of the processor as a whole [10, 4, 23]. This is the goal of pipeline gating.

We will use the schematic of the processor pipeline shown in Figure 4.2 to describe pipeline gating. Like many high-performance processors, such as the DEC AXP-21164 or Intel PentiumPro, our sample pipeline uses two fetch and decode cycles to allow the clock rate to be increased. We assume the fetch stage has a small instruction buffer to allow instruction fetch to run ahead of

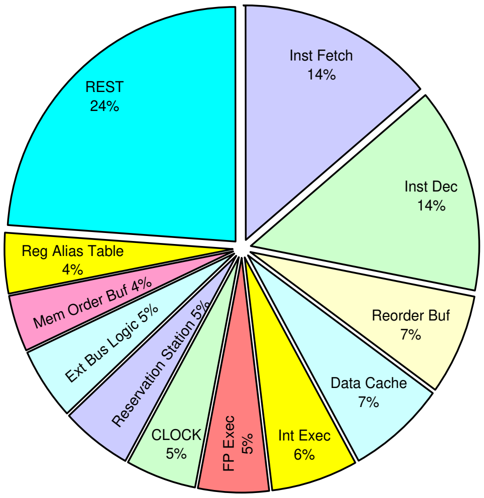


Figure 4.1. Power Consumption for PentiumPro Chip, Broken down by Individual Processor Components

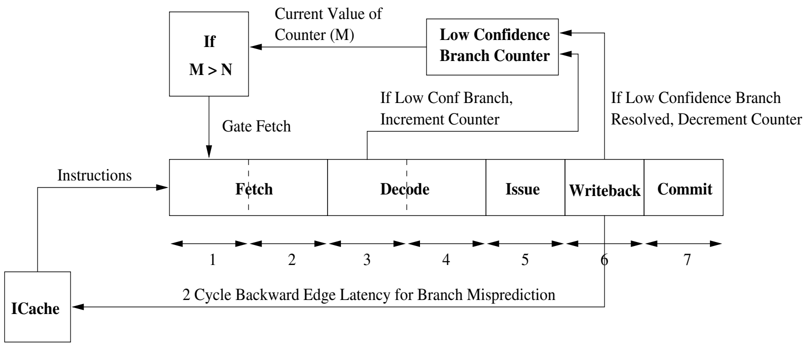


Figure 4.2. Pipeline with a Two Fetch and Decode Cycles Showing Hardware required for pipeline gating. The low-confidence branch counter records the number of unresolved branches that reported as low-confidence. The counter value is compared against a threshold value ("N"). The processor ceases instruction fetch if there are too many unresolved low-confident branches.

decode. Branch prediction occurs when instructions are fetched to reduce the misfetch penalty. The actual instruction type may not be known until the end of decode. Conditional branches are resolved in the execution stage, and branch prediction logic is updated in the commit stage. Since the processor uses out-of-order execution, instructions may sit in the issue window for many cycles, and there may be several unresolved branches in the processor.

We use a *confidence estimator* to assess the quality of each branch prediction. A "high confidence" estimate means we believe the branch predictor is likely to be correct. A "low confidence" estimate means we believe the branch predictor has incorrectly predicted the branch. We use these confidence estimates to decide when the processor is likely to be executing instructions that will not commit; once that decision has been reached, we "gate" the pipeline, stalling specific pipeline stages. In our study [20], we varied a number of system parameters, including the branch predictor, the confidence estimator, the stage at which a gating decision is made, the stage that is actually gated and the number of outstanding low-confident branches needed to engage gating. The decision to gate can occur in the fetch, decode or issue stages. Equally important is the decision about *what* to gate and *how long* to gate. Gating the fetch or decode stages would appear to make the most sense, and we examined both cases. We used the number of unresolved low confidence branches to determine when to gate. For example, if the instruction window includes one low-confident branch, and another low-confident branch exits the fetch (or, alternatively, decode or issue) stage, gating would be engaged until one or the other low-confident branch resolves. Figure 4.2 illustrates this process for a specific configuration. We add a counter that is incremented whenever the decode encounters a low-confident branch and is decremented when a low-confident branch resolves. If the counter exceeds a threshold, the fetch stage is gated. Instructions in the fetch-buffer continue to be decoded and issued, but no new instructions are fetched.

To properly understand the effects of stalling the pipeline, we developed a model of an out-of-order, speculative, wide-issue processor. We have found that gating the processor typically stalls the processor for a very short duration. Figure 4.3 shows the number of times a specific configuration of our pipeline model is stalled while executing different programs. Generally, gating stalls occur for about 2-4 processor cycles. Most processor configurations exhibit a similar distribution, and indicate that our mechanism is exhibiting fine control over the speculative state of the processor.

We explored moving the point of gating to the decode and issue stages rather than simply eliminating fetching. Gating at decode saved less power than gating at the fetch stage. This is reasonable since we are allowing the fetch stage to continue fetching until the fetch buffer is full. Therefore, more instructions will enter the pipeline; this would not happen if gating disabled instruction fetch. We expected to see an improvement in processor performance (CPI) with gating at decode since the recovery penalty for incorrect gating would be less than gating at fetch. It takes only three cycles for an instruction to "catch up" and issue after an incorrect gating event with gating at decode as opposed

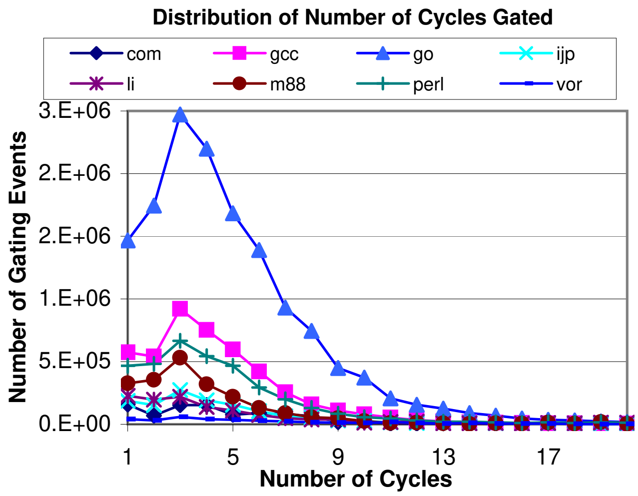


Figure 4.3. Distribution of Gating Events and the Number of Cycles Gated per Event

to five cycles with gating at fetch. However, results showed no real performance benefit from moving the gating point from fetch to decode. As shown in Figure 4.3, the pipeline is generally not gated for more than a few cycles. Our pipeline model used a 64-entry register update unit, and results show that it usually has enough instructions in the issue queue to keep the execution units occupied while the pipeline catches up from gating.

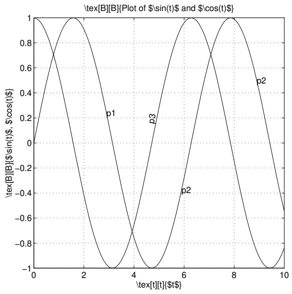
Overall, we found that gating the processor typically stalls the processor for a very short period and that we could vary the design of the confidence estimator to reduce the number of issued instructions by $\approx 20\%$ while reducing performance, measured by instructions per cycle, by typically less than 1%. As with most work on power-efficient microarchitecture, it's difficult to state how much power or energy is saved overall in an actual implementation, since those figures depend on numerous implementation decisions and higher level power analysis tools have poor accuracy [9]. Since the overheads for pipeline gating are so small, it should be effective across a wide range processor designs.

4. Speculation Control by Exploiting Scheduling Slack

We have explored how clustered voltage and frequency scaling techniques can be used to reduce processor energy. We assume the processor has independent clock domains and can, for example, run certain components such as functional units at $600Mhz$ and other functional units at $300Mhz$. There are a number of complications with such a design, since the different clock domains will require DC-DC power conversion and inter-domain communication may

add	r0,	r1,	r2;	(A)
sub	r3,	r4,	r5;	(B)
and	r9,	0x1,	r9;	(C)
ornot	r5,	r9,	r10;	(D)
xor	r2,	r10,	r11;	(E)

(a) Code Sequence



(b) Program Dependence Graph

Figure 4.4. Sample Code Sequence and Program Dependence Graph to Illustrate Scheduling Slack

have some overhead. However, it presents an interesting opportunity for power reduction: *if we are able to dynamically identify instructions that could use the slower components without impacting the overall rate of computation, we may be able to save additional power without impacting performance.* This design technique can be exploited at multiple levels. To date, we have focused on instruction-level scheduling but we are also investigating thread-level scheduling that can be exploited either by multithreaded or chip-multiprocessing designs.

To exploit functional units with different clock speeds, the critical issue is identifying those instructions that can be computed at a slower rate without impacting performance. These instructions must be identified early in the pipeline so that the information can be used at the scheduling/issue stage without impacting performance. Following the terminology of Johnson *et. al* [15], we call this property "scheduling slack". We will formally define scheduling slack shortly, but the simple example shown in Figure 4.4 provides most of the insight.

In Figure 4.4(a), we see a sample dynamic code sequence executing in the microprocessor pipeline with each instruction labeled *A* through *E*. Figure 4.4(b) shows the data dependences for the code sequence. In this example, instructions *A*, *B*, and *C* are in the ready queue of the processor's issue logic, and will issue in the current cycle; instructions *D* and *E* are decoded but are not part of the ready queue since they must wait for instructions *A* through *C* to complete before they can be issued.

This example assumes each instruction has an operational latency of one cycle and that three functional units are available. Figure 4.5(a) shows a possible

1	A	B	C
2			D
3			E

(a) Standard Schedule

1	A_1	B	C
2	A_2		D
3			E

(b) Schedule Exploiting Slack

Figure 4.5. Possible Execution Schedules

execution schedule for this code sequence; instruction A can be executed either at cycle 1 or 2 and still produce a useful result by cycle 3, when the result of A is needed by instruction D.

In this example, instruction A has one cycle of slack; on a conventional architecture, it could be issued one cycle later (*i.e.* either at cycle 1 or 2) and not impact the performance of the processor. If it is possible to design hardware that employs multiple clock domains, this extra degree of slack also implies that instruction A could be run at $1/2$ speed (or, in general, at $1/(slack+1)$ speed) and achieve the same performance. For example, instruction A could take two cycles to actually execute, which we denote in Figure 4.5(b) by the functional unit being filled with A_1 and A_2. Alternatively, the instruction could be scheduled at cycle 2 to reduce the total number of concurrently executing instructions, possibly reducing peak power demands.

4.1 Theoretical Underpinnings of Slack Scheduling

The principles behind "slack scheduling" can be precisely defined using a conventional graphic-theoretic model. However, this module is impractical to implement in hardware. We first describe the formal model and then the realizable implementation.

We use conventional dependence graph notation. To compute scheduling slack for a set of instructions (for either statically or dynamically scheduled processors) each cycle, we construct a dependence graph $G = (N, E)$ of all available instructions. We assume we know the operational latency of each instruction (Lat_j). The "start time" for an instruction (S_j) is the earliest that a given instruction can issue; the "finish time" (F_j) is when that instruction will finish execution. We can recursively define these terms as:

$$\begin{aligned}\mathrm{F}_j &= \mathrm{S}_j + \mathrm{Lat}_j \\ \mathrm{S}_j &= \max_{k \in \mathrm{pred}_j} \mathrm{F}_k\end{aligned}$$

with $\mathrm{S}_j = 0$ when node j has no predecessors ($\mathrm{pred}_j = \emptyset$). We will also need to know the maximum delay for the critical path, $\mathrm{MaxDelay} = \max_{k \in N} \mathrm{F}_k$.

The "slack" (Sl_j) in an instruction is the difference between when that instruction can finish and the earliest time that the results are needed. To compute this, we need to introduce the notion of a "latest time" (L_j) for an instruction - this is the latest time by which an instruction must finish to meet the critical path delay for the instructions. For a single dependence graph, the latest time and finish time are identical; in practice, the dependence relations form a multi-graph and portions of the graph may not be on the critical path and thus may not need to be executed quickly. This leads to the following:

$$\begin{aligned} \mathrm{L}_j &= \begin{cases} \mathrm{MaxDelay} & \text{if } \mathrm{succ}_j = \emptyset \\ \max_{k \in \mathrm{succ}_j} (\mathrm{L}_k - \mathrm{Lat}_k) & \text{if } \mathrm{succ}_j \neq \emptyset \end{cases} \\ \mathrm{Sl}_j &= \mathrm{L}_j - \min_{k \in \mathrm{succ}_j} \mathrm{S}_k \end{aligned}$$

It is easiest to explain these equations by extending our earlier example to include multiple dependence graphs, as shown in Figure 4.6; again, we assume all operational latencies are a single cycle. Each node in Figure 4.6 is labeled with the corresponding start, finish and late times as well as the computed slack. The critical path ($B \rightarrow D \rightarrow E$) delay is 3 cycles. Instructions X and Y are not on the critical path. Since no instructions depend on Y, it could execute in either cycle 2 or 3 with no impact on the total execution time, *assuming that preceding initiate at their start time*. Likewise, instruction X can execute at either cycle 1 or 2 without affecting the critical path.

Up to now, we have only addressed the *innate scheduling slack* that depends purely on dependence relations. Actual programs have considerable innate slack available even when considering just the issuable instructions [2]; however, processors can also exploit *runtime slack*. This additional slack arises either due to mandatory delays such as cache misses or due to advisable delays such as those caused by pipeline gating. For example, a load instruction normally has an operational latency of a single cycle, unless the load misses in the cache, at which time the load may have a latency of 10's or 100's of cycles. Likewise, a branch instruction normally has a single cycle of operational latency, but pipeline gating may delay instruction fetch following that branch for speculation control and thus increase the effective operational latency beyond that measured in this paper.

To underscore this point, refer to Figure 4.7, which shows two load instructions (`ld A` and `ld D`). Even if we know the expected load delay of the `ld A` instruction, we can not use it to any advantage. But, if we know that `ld D` is likely to miss, the start time for `Y = X + D` is lengthened, providing more slack for the `X = A + 10` instruction and `ld A`. We would expect this would introduce slack in most programs.

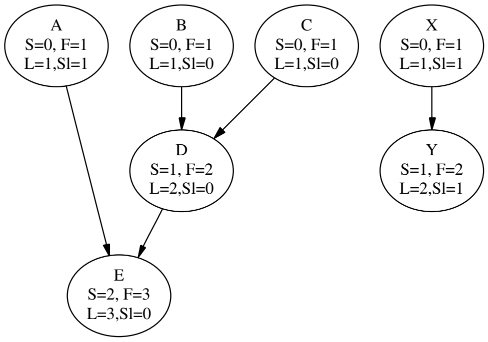


Figure 4.6. More Complex Example Containing a Multi-Graph to Explain Flow Equations

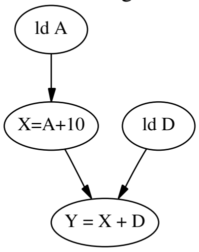


Figure 4.7. Simple Example Showing the Potential Influence of Cache Misses on Schedule Slack

In any given cycle, multiple instructions may have innate slack, but it may be that only a subset of the instructions can exercise that slack. This may affect the efficacy of on-line methods to determine instruction slack.

Scheduling slack can be used for a number of purposes – it may be possible to issue instructions to different clusters based on slack or prioritize dependence resolution based on slack. We have initially used slack to schedule instructions on "low power" functional units. This means we are mainly interested in the scheduling slack available for instructions that are *issuable* (*e.g.* instructions A, B, C and X in Figure 4.6).

4.2 Implementing Slack Detection

We've conducted limit studies to examine the potential scheduling slack in applications [2]; our preliminary findings were encouraging because we found that most execution cycles contain at least one instruction that can be delayed by at least a single cycle. The remaining challenge is to find a practical method to identify slack. The graph-theoretic model we described is impractical, because it examines all instructions every processor cycle; this would lead to excessive overhead.

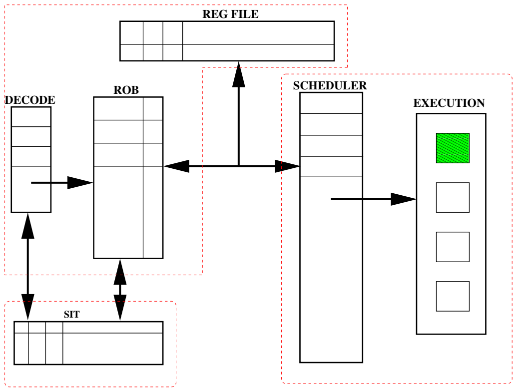


Figure 4.8. Major Architecture Components Involving SIT

We've focused on a technique that "learns" which instructions have scheduling slack using some simple hardware, and have conducted in-depth power and performance analysis for this implementation. The proposed hardware support for the dynamic power reduction system uses three components to detect and exploit instruction dependency slack. The elements perform detection, state tracking, and execution of slackful instructions. First, the instruction must have a minimal amount of *slack*, allowing slow execution without impacting performance. Second, the slack state must be maintained across execution iterations so that the slack may be utilized. Finally, lower power integer execution units exploit the detected slack, saving power at the expense of increased latency. The following components function as a system to accurately detect slackful operations within dynamic instruction sequences and reduce integer execution core power without significantly impacting performance:

Slack Indicator Table (SIT) is a simple structure that is the center of this approach. This tracks instruction slack behavior across execution iterations. In this way, it serves as a history-based prediction mechanism for the scheduling of tolerant operations on slow execution units.

Re-Order Buffer, and Register File extensions are necessary for communicating slack state between both in-flight and completed operations. Further, they identify the *first* consumer of an operations output; consumer ordering is imperative to the computation and back-propagation of slack from a consumer (child) to its producers (parents).

Scheduler Modifications and Low Power Execution Units complete the system with the units which actually save power and the means to utilize them appropriately.

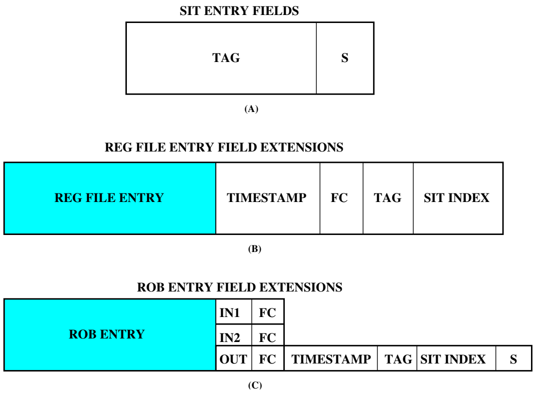


Figure 4.9. Additional Fields Required for SIT

As an instruction is decoded, SIT table is probed to determine if it has had slack in the past. If the operation has had slack, it is preferably dispatched to a slow unit if available; if a fast unit is ready, it may execute there or lose its slack and wait. As instructions complete, they normally announce their completion to all dependent instructions; in our implementation, dependent instructions determine if an input dependence would have scheduling slack. For example, assume an instruction is waiting for two registers, R_s and R_t; if the producer of R_s finishes before the producer for R_t, than the producer of R_s *may* have slack. In fact, the producer of R_s must complete before *any* dependent instruction needs the value of R_s. However, we only need to record when the *first* instruction requests the value of R_s.

4.3 Slack Indicator Table

Figure 4.9(a) is one entry of the Slack Indicator Table. The table is simply an array of such entries, each comprised merely of a *Tag* and *Slack Bit* (S). The tag and index of the SIT are derived and used in the same manner as in a cache. The slack bit indicates if the operation at that index was found to be slackful in its previous execution. The hardware cost of the SIT is proportional the number of entries and the corresponding tag length. The power cost of the SIT can be estimated as an equivalent size cache tag array.

Figure 4.9(b) shows the fields added to the register file entries. The *TIMESTAMP* field is the cycle count value when the result of the operation was written into that register. The timestamp field has to be large enough to capture the largest latency between when a consumer is mapped to a producer and when that producer completes execution (e.g., memory access latency). The *FC* (First Consumer) is a single bit field that is set only by the first operation to read the value after it has been written. Subsequent operations will see that

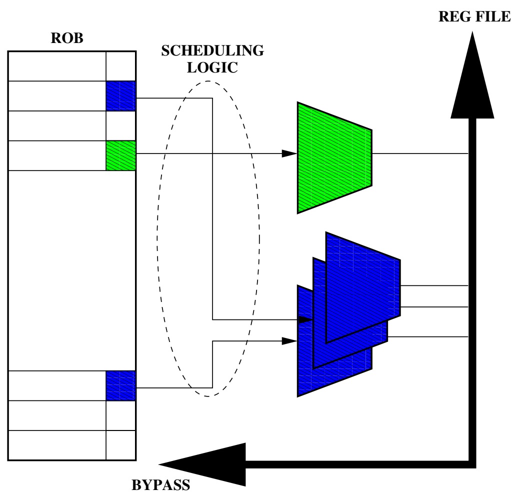


Figure 4.10. Scheduling of Slackful Operations to Slow and Fast Execution Units

this value has been consumed by a previous operation in execution order. The instruction writing the register copies its tag into the *TAG* field. Likewise, the *SIT INDEX* is from the writing instruction's PC, and points to its entry in the SIT.

The ROB entry extensions are shown in Figure 4.9(c). There are two sets of extensions: the inputs set tracks the inputs for which this operation is the first consumer; the output set contains the state for the operation in that ROB entry (and the output it will produce). The FC flags are associated with the input dependence pointers. The *FC*, *TIMESTAMP*, *TAG*, and *SIT INDEX* fields for the ROB output are set and used the same as those in the register file. The output of an operation in the ROB, however, may or may not be written into the register file. Hence, the state must be maintained in that ROB entry until it is either squashed (the state is also squashed) or written into the register file (the state is transferred into the matching fields of the register). The slack bit *S* is read from the SIT and written into the ROB as the instruction is being decoded.

4.3.1 Using the Indicated Slack. The scheduler treats a ROB entry slack bit as a priority bit for selecting an execution resource, as shown in Figure 4.10. If the slack bit is set, the operation will be scheduled to the slow unit, if possible. If the bit is clear, the scheduler attempts to send the operation to a fast unit.

There are several avenues to creating a low power resource. The specific path chosen depends upon the cost-benefit tradeoffs considered. Power can be traded for area or complexity – the circuit design techniques used to achieve a single-cycle functional unit becomes increasingly complex as processors enter multi-Ghz frequency domains. Scaling back the aggressiveness of a functional unit design can significantly reduce the energy needed by that functional unit. Still another option is to design a resource that operates at a lower frequency and/or voltage. The type of tradeoff in part determines the amount of power the device can conserve, as well as the nature and degree of the performance impacts it will incur. The selection of solution is largely implementation specific.

The low power resources used in this work are restricted to integer execution units (IEU). However, the principles for use can be extended to any resource type without loss of generality. The integer units are partitioned into two clusters, one which runs at the standard voltage (V_{dd}) and speed (F), and the other running at one half the standard voltage and one half the standard frequency. This approach leverages the technology of multiple voltage and frequency circuit design. From the Power relation of $P = V_{dd}^2 * F * C$, there is an approximate $3/4$ power savings by halving both the voltage and frequency. However, due to the 2X increase in operation latency (albeit at $1/4$ power) will result in a net energy savings of $1/2$. In this work, the standard IEU latency is a single cycle, with the slow units having a latency of two cycles. Hence, the slack threshold is one cycle.

4.3.2 Costs and Benefits of the SIT Slack Detection.

The power model for this study is derived from actual Pentium III processor power values [1]. Where there is no direct correlation between the hardware which the power is based upon and the architecture modeled by the simulator, the power model was modified to fit. The base power values did not employ any form of clock gating. Inactive and static power values were derived from [5], representing an aggressive clock gating infrastructure which is captured by the TEMPEST power model [6]. The inactive power of each component was taken as 10% of the active power.

Figure 4.11 shows the IPC degradation (higher is worse) for an infinite SIT across the benchmarks. We examined a number of mechanisms to maintain the SIT state, including decay (*Decay*), self-identified slack (*Self-ID*), slack identification stage (*Stage*), and the default (*Default*), which employs none of these state update techniques. Decay allows the in-flight-instruction to clear its slack bit (but leave the SIT state untouched) if it misses a dispatch opportunity. Self-ID marks completing instructions with no dependents as slackful, rather than waiting for a child to propagate slack to it. Stage performs the producer slack determination by operations as they are about to issue, rather than when they are awakened. This variation attempts to capture more of the resource constraints of the machine by making the timestamp comparison closer reflect the actual completion to use time delta. There is no single technique that is

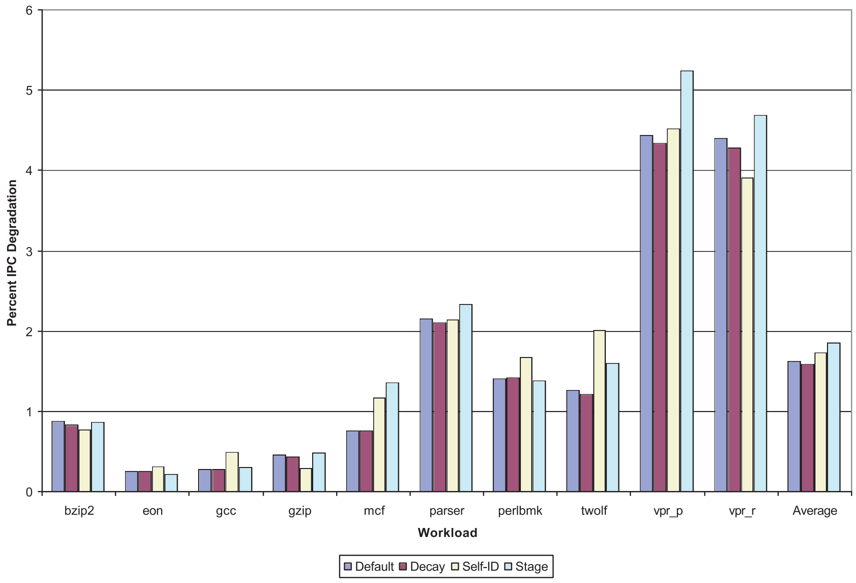


Figure 4.11. IPC Degradation when Using Various State Update Techniques and an Infinite SIT

preferable across all workloads, and the IPC impact varies by workload. It appears that using the default mechanism performs as well as the other techniques, on average. Note that most workloads suffer an IPC degradation of under 2%. The *vpr_p* application suffers the most, using Stage, at 5.3% IPC decrease. There is an inherent lowering of IPC using this scheme, even under ideal conditions, due to the effective integer instruction throughput being reduced to 3.5 per cycle (maximum) from the default of four.

The integer execution unit power reduction is shown in Figure 4.12 (higher is better). The average IEU power is reduced by 11%, with slightly better results from the Self-ID and Stage techniques. These two techniques are more aggressive towards marking instructions as slackful. This accounts for both their resultant decrease in IPC and increase in IEU power savings versus the default mechanism.

Our work on detecting and exploiting slack is only beginning, but it's clear from these results that there are practical mechanisms that can identify scheduling slack and that scheduling slack can be profitably employed. Although our first application of scheduling slack targeted instruction execution, we think it can be applied to processor components that consume more energy, such as the instruction issue and retirement logic. Moreover, we are examining mechanisms other than the SIT tables that may be more accurate in identifying slack.

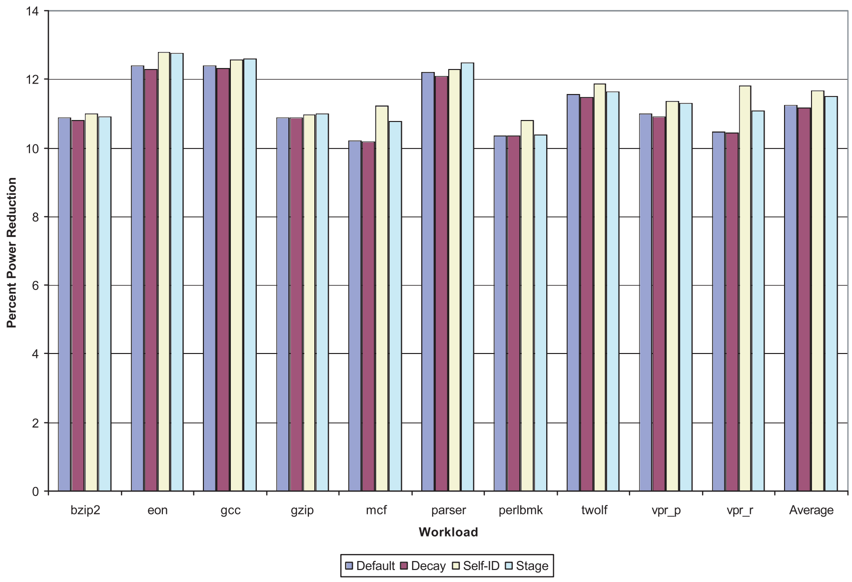


Figure 4.12. IEU Power Reduction when using Various State Update Techniques and an Infinite SIT

5. Summary

The thesis of our research group when we started examining micro-architectural techniques to reduce processor power was that there were mechanisms that could reduce power but that wouldn't normally be considered because they didn't increase performance. Our efforts have focused on speculation control, or attempting to reduce the waste caused by the speculative execution needed to yield high performance processors. Pipeline gating controls speculation by reducing instruction fetch when it is unlikely to be productive. Slack scheduling delays instruction execution to occur no earlier then necessary to save power and energy. We think that the techniques described here and other techniques we've examined show that speculation control can reduce power; in turn, this allows processors to consume less energy and also to run at faster clock speeds.

Acknowledgments

I would like to thank the doctoral students with whom I've had the good fortune to work on issues of power-aware microarchitecture: Jason Casmira Soraya Ghiasi, Artur Klauser and Srilatha ('bobbie') Manne. In particular, the work on confidence estimation was part of the thesis work of Artur Klauser and Bobbie Manne; Bobbie also explored issues in pipeline gating. Jason Casmira has been exploring how "slack scheduling" can be used to reduce processor power. I would also like to thank Steve Gunther of Intel for power data from Pentium processors.

References

[1] D. Carmean. Power, a perspective from the desktop. In *Kool Chips Workshop, invited talk*. ACM, Dec 2000.

[2] J. Casmira and D. Grunwald. Dynamic instruction scheduling slack. In *Kool Chips Workshop*. ACM, Dec 2000.

[3] A. Chandrakasan and R. Brodersen (Eds.). *Low-Power CMOS Design*. IEEE Press, 1998.

[4] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[5] Intel Corporation. Pentium 4 Processor Datasheet. http://developer.intel.com/design/Pentium4/datashts/.

[6] A. Dhodapkar, C. Lim, G. Cai, and W. Daasch. Tempest: A thermal enabled multi-model power/performance estimator. In *Workshop on Power Aware Computer Sytstems*. ACM, Nov 2000.

[7] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proc. of 28th. Int. Symposium on Computer Architecture*, June 2001.

[8] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, and David Patterson. The Energy Efficiency of IRAM Architectures. Technical report, May 1997.

[9] Soraya Ghiasi and Dirk Grunwald. A comparison of two architectural power models. In *Workshop on Power Aware Computer Sytstems*. ACM, Nov 2000.

[10] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[11] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence estimation for speculation control. In *Proceedings 25th Annual Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998. ACM.

[12] Steve Gunther and Intel Corp. Suresh Rajgopal. Personal communication.

[13] Mutsunori Igarashi, Kimiyoshi Usami, Kazutaka Nogami, Fumihiro Minami, Yukio Kawasaki, Takahiro Aoki, Midori Takano, Chiharu Mizuno adn Takashi Ishikara, Masahiro Kanazawa, Shinji Sonoda, Makoto Ichida, and Naoyuki Hatanaka. Clustered voltage scaling techniques for low-power design. In *IEEE Symposium on Low Power Electronics*, 1997.

[14] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual Inter-*

national Symposium on Microarchitecture, pages 142–152, Paris, France, December 2–4, 1996.

[15] M.C. Johnson and K. Roy. Optimal selection of supply voltages and level conversions during data path scheduling under resource constraints. In *Proceedings of the International Conference on Computer Design*, 1996.

[16] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *IEEE Micro*, December 1997.

[17] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald. Selective eager execution on the polypath architecture. In *Proceedings 25th Annual Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998. ACM.

[18] U. Ko, P. T. Balsara, and A. K. Nanda. Energy optimization of multilevel processor cache architectures. In *Proceedings of the International Symposium on Low Power Design*, 1995.

[19] Mikko Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

[20] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings 25th Annual Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998. ACM.

[21] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Branch prediction using selective branch inversion. In *Proceedings of Parallel Architectures and Compilation Techniques PACT'99*, Oct 1999.

[22] J. Montanaro and *et. al*. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *Digital Technical Journal*, volume 9. Digital Equipment Corporation, 1997.

[23] M. Srivastava, A. Chandrakasan, and R. Broderen. Predictive system shutdown and other architectural techniques or energy efficient programmable computation. *IEEE Transactions on Very Large Scale Integration Systems*, 4(1):42–55, March 1996.

[24] C. Su and A. Despain. Cache design trade-offs for power and performance optimization: A case study. In *Proceedings of the International Symposium on Low Power Design*, 1995.

[25] Kimiyoshi Usami and Mark Horowitz. Clustered voltage scaling techniques for low-power design. In *IEEE Symposium on Low Power Electronics*, 1995.