Deterministic Clock Gating to Eliminate Wasteful Activity in Out-of-Order Superscalar Processors due to Wrong-path Instructions[1]

Nasir Mohyuddin, Kimish Patel and Massoud Pedram
Department of Electrical Engineering (Systems)
University of Southern California, Los Angeles, CA, USA
E-mail: {mohyuddi,kimishpa,pedram}@usc.edu

***Abstract* - In this paper we present deterministic clock gating schemes for various micro architectural blocks of a modern out-of-order superscalar processor. We propose to make use of 1) idle stages of the pipelined function units (FUs) and 2) wrong-path instruction execution during branch mis-prediction, in order to clock gate various stages of FUs. The baseline Pipelined Functional unit Clock Gating (PFCG), presented for evaluation purpose only, disables the clock on idle stages and thus results in 13.93% chip-wide energy saving. Wrong-path instruction Clock Gating (WPCG) detects wrong-path instructions in the event of branch mis-prediction and prevents them from being issued to the FUs, and subsequently, disables the clock of these FUs along with reducing the stress on register file and cache. Simulations demonstrate that more than 92% of all wrong-path instructions can be detected and stopped from being executed. The WPCG architecture results in 16.26% chip-wide energy savings which is 2.33% more than that of the baseline PFCG scheme.**

I. INTRODUCTION

Power dissipation and the resulting temperature rise have become the dominant limiting factors to processor performance and constitute a significant component of its cost. Expensive packaging and heat removal techniques are required to achieve acceptable substrate and interconnect temperatures in high-performance microprocessors. The total amount of power required to distribute the clock signal across a microprocessor chip is as large as 20-40% of the total power consumption [1].

Clock gating is a well known technique used to reduce power dissipation in clock associated circuitry. The idea of clock gating is to shut down the clock of any component whenever it is not being used (accessed). It involves inserting combinational logic along the clock path to prevent the unnecessary switching of sequential elements. The conditions under which the transition of a register may be safely blocked should automatically be detected. This problem is the target of our paper.

In out-of-order superscalar processors, branch miss-predictions cause wrong-path instructions to be executed since there is a lag between the branch prediction, actual branch resolution, and subsequent commit of the branch. The wrong-path instructions are of course never committed to the actual state of the processor; however, because they are issued and executed, they can give rise to two negative effects: performance degradation and power waste.

Many researchers have worked on eliminating or reducing the power consumed by wrong-path instructions. These schemes are primarily probabilistic in nature. They rely on some kind of branch history as explained next. The pipeline gating technique of [2] assigns confidence levels about their prediction accuracy to branches. When the number of low confidence branches exceeds a preset threshold, the instruction fetch and decode are stopped. This method suffers from both performance overhead and lost energy saving opportunities since some low confidence branches may be predicted correctly while some high confidence branches are in fact predicted wrongly. Reference [3] improves on the all-or-nothing throttling mechanism of [2] by having different types and degrees of throttling.

In [4] the authors propose a deterministic clock gating approach which takes advantage of the resource utilization information available in advance. When it is known ahead of time that some of the processor resources will not be used, clock gating signals are generated, at the issue stage, to clock-gate these resources during their idle times. Another approach, called transparent clock gating [5], enhances the existing clock gating in latch-based pipelines by keeping the latches transparent by default i.e., by not clocking them. Latches are clocked only when there is a need to avoid a data race condition. Register level clock gating of [6] introduces the concept of clock gating parts of stage registers i.e., when there are not enough instructions to be issued, parts of stage register associated with the issue stage are clock gated.

Most of the previous work on clock gating either ignores the fact that a noticeable fraction of the total power is dissipated in executing wrong-path instructions during branch misprediction or use a probabilistic approach to avoid the resulting power waste. In this paper we take branch misprediction as an opportunity for clock gating the unnecessarily-used processor resources by *deterministically* detecting the wrong-path instructions.

[1] This research was sponsored in part by a grant from the National Science Foundation.

II. MOTIVATION

Many of the currently available state-of-the-art microprocessors employ aggressive branch prediction in order to boost performance. Although branch predictors help increase the processor performance, when a branch is mispredicted, many of the wrong-path instructions (i.e., instructions that are on the predicted path of the mispredicted branch) are still executed. Due to the out-of-order execution in modern processors, at the time when a branch is resolved and found to be mispredicted, there can be a mix of correct path and wrong-path instructions in the execution pipelines and the instruction queue. Because of the prohibitive complexity of selective squashing mechanism, many processor architectures do not flush the pipeline until the mispredicted branch reaches the head of the ReOrder Buffer (ROB) so that one is assured that all the instructions on the correct path have retired (Note that instruction fetch and decode are stopped upon detecting a branch misprediction). As a result many of the wrong-path instructions are still executed only to be thrown away when the pipeline is flushed. Figure 1 on the primary Y-axis (left) shows the fraction of instructions that are executed but never committed (retired), due to mispredicted branches with respect to the total number of instructions executed. This estimate is obtained from simplescalar simulation, using the processor configuration that is described in detail in the experimental results section, which shows that on average around 8.29% of the executed instructions are due to mispredicted branches. These instructions not only consume power in functional units during their execution, but also consume power in (i) register file (RF) by reading their input operands; and (ii) caches by executing wrong-path loads. The impact of these wrong-path instructions on power dissipation is even more severe with deeper pipelines on account of increased branch misprediction penalty.

As stated earlier, many of the wrong-path instructions are executed even after the branch is resolved. More precisely, when a branch is resolved to be mispredicted, there may exist wrong-path instructions which a) have already been issued and thus they either are in the pipeline or have been completed (type (i)), or b) have not been issued yet, i.e., they are still in the issue queue (IQ) (type (ii)). By the time the mispredicted branch reaches the head of the ROB, many of the instructions which are still in IQ (type (ii)) could be issued to execution units. It is quite expensive (from a hardware cost and control point of view) to identify and prune type (i) instructions. Fortunately, it is easy to stop the second set of instructions from being issued, which in turn can result in considerable power saving.

In Figure 1 on the secondary Y-axis (right), the bars on the left within each set show the average number of type (i) + type (ii) instructions when the mispredicted branch retires. This number tells us the average number of wrong-path instructions that could be prevented from being issued if we had a perfect oracle that would tell us which instruction is or will be in the wrong-path. The bars on the right within each set show the average number of type (ii) instructions when the mispredicted branch retires, i.e., the wrong instructions issued after the branch is resolved to be mispredicted and before it retires. These are the wrong-path instructions which can actually be prevented from being issued and executed. These results show that 92.63% of the wrong-path instructions are issued after the branch is resolved, which provides a great opportunity for power saving via clock gating.

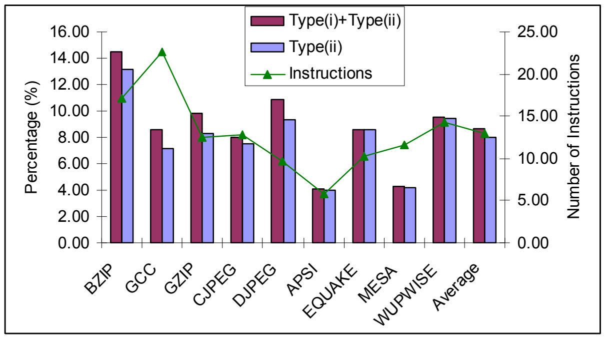

Figure 1. Percentage of wrong-path instruction over total instructions executed and average number of wrong-path instructions per mispredicted branch.

III. PROPOSED CLOCK GATING ARCHITECTURE

Based on the aforesaid observations, we present two clock gating techniques that 1) make use of idle cycles in pipelined functional units when some stage of the functional unit is idle, and 2) prevent wrong-path instructions of type (ii) from being issued.

The first clock gating technique, called Pipeline Functional unit Clock Gating (PFCG), is straightforward and is presented and implemented here only to serve as a baseline against which the power efficiency of a second technique i.e., WPCG, is compared.

A. Pipelined Functional Unit Clock Gating

Figure 2 depicts the PFCG technique at the architectural level. The proposed architecture utilizes the idleness of various stages of structurally-pipelined functional units in a processor pipeline.

Note that different stages of a pipelined FU can be idle due to any of a number of reasons:

- Typically the total number of FUs, including integer and floating point functional units, is larger than the processor's issue width. Hence not all the FUs are used in every cycle of the program's execution.
- Different applications exhibit different degrees of instruction level parallelism (ILP) and therefore the FU's usage varies across different programs.
- Different application programs exercise different sets of FUs. For example, integer programs will be using completely a different set of FUs (integer ones) compared to the floating point programs.

- Because of structurally pipelined FU with multi clock cycle latencies (but throughput of 1 operation per cycle), depending on the number of operations that are concurrently being executed on the same functional unit, one or more stages of the pipelined FU may be idle at any given clock cycle.

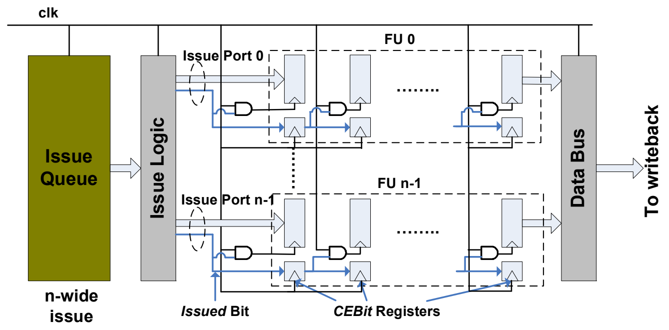

Figure 2. PFCG Architecture.

In the modern processors, the decoded instructions, after renaming, are stored in an issue queue (IQ), where they wait for their input operands to become available (if these operands are being produced by some instruction in the pipeline). The issue logic examines all instructions that have both of their operands ready and issues n instructions (for an issue width of n) to appropriate FUs assuming that the corresponding FUs are available. We define a pipeline stage of an FU as an input register set plus the combinational logic that succeeds it. In the presented clock gating (CG) architecture, each stage register set of the FU is appended with a one-bit register called Clock Enable Bit register (*CEBit*). The *CEBit* of stage i of FU j controls the clock of stage i+1 of that FU. (Note that since the last stage of the FU will not be used to gate any clock signal, it is not appended with the *CEBit*).

The clock fed to each stage register set, except for the *CEBit* register which is never clock gated, goes through an AND gate. The AND gate essentially takes the clock and the *CEBit* of the previous stage and performs logical AND on them to produce the clock that will be fed to the current stage. Hence, during a particular clock cycle, if the *CEBit* of the previous stage is '0', the clock for the current stage is masked for that cycle. As shown in the figure, the *CEBit* propagates through subsequent stages at each clock cycle thanks to the *CEBit* shift register structure.

The *CEBit* register of the first stage of each FU is set either to '0' or to '1' by the issue logic via the *issued bit* (cf. Figure 2). If, during a particular cycle *m*, no instruction is issued to the FU, then the *issued bit* will be set to '0', indicating that no instruction is issued to this particular FU during cycle *m*. The *issued bit* is also used to gate the clock of the first stage. In the subsequent clock cycles as the *CEBit* travels through the subsequent stages of an FU, it appropriately gates the clock of those stages.

B. *Wrong-Path instruction Clock Gating*

We saw in section II that on average 8.29% of the total executed instructions are never committed due to wrong-path instructions on mispredicted branches. Figure 1 showed, on average, how many wrong-path instructions can be prevented from being issued when the branch is resolved and is known to be mispredicted. As seen, when the branch is mispredicted, majority of the issued wrong-path instructions can be blocked since the majority of these wrong-path instructions are still in IQ. Therefore, we propose a clock gating technique that eliminates the switching activity in the logic and the stage registers due to wrong-path instructions.

Figure 3 shows the architecture of Wrong-Path instructions Clock Gating (WPCG). Note that when a branch is resolved to be mispredicted, the instructions in the IQ may be correct path instructions (i.e., instructions that were fetched before the mispredicted branch instruction) or wrong-path instructions (i.e., instructions that have been fetched after the mispredicted branch instruction). Therefore, in the WPCG architecture, the IQ is augmented with some logic to determine whether the instruction selected by the issue logic is a wrong-path instruction or not.

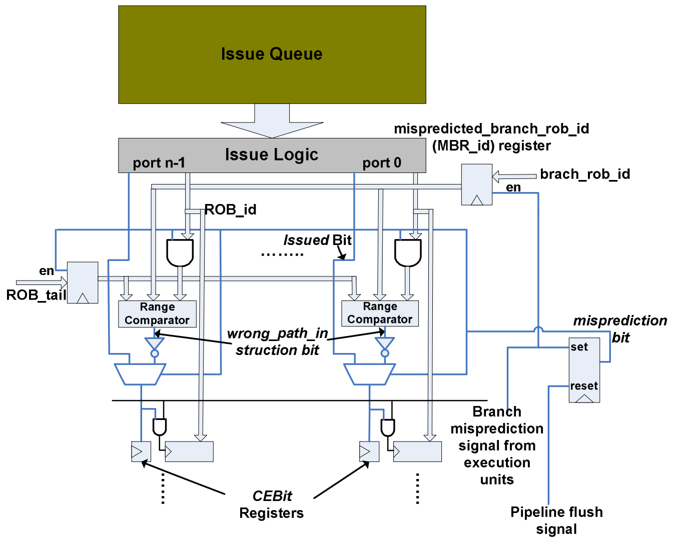

Figure 3. The WPCG architecture.

As depicted in Figure 3, the misprediction bit is set to '0' initially when the correct path instructions are being executed and no branch misprediction has taken place. When a branch is resolved to be mispredicted, the mispredicted_branch_rob_id (MBR_id) register is updated with the ROB ID of the branch (branch_rob_id) in the next clock cycle. At the same time, the misprediction bit will be set to '1'. This will enable the range comparator in front of each issue port of the IQ, which will subsequently determine whether the instruction being issued is a wrong-path instruction or not.

The AND gate in front of each issue port essentially takes the ROB ID of the selected instruction and ANDs it with the misprediction bit. This is necessary since we do not want unnecessary switching activity in the comparator circuit when the branch is predicted correctly. Hence, in the event of misprediction, the ROB ID of the selected instruction is available to the comparator. Furthermore the comparator also receives the tail of the ROB as input to determine if the selected instruction is between the mispredicted branch and the tail of the ROB. If it is, then the comparator will output a '1', indicating that the selected instruction is in the wrong-path and thus it should not be executed. The inverted output of the comparator goes to a 2-to-1 MUX controlled by the misprediction bit.

In the event of a misprediction, the inverted output of the comparator is chosen to set the value in the *CEBit* register of the first stage of the FU. This output is also used to clock

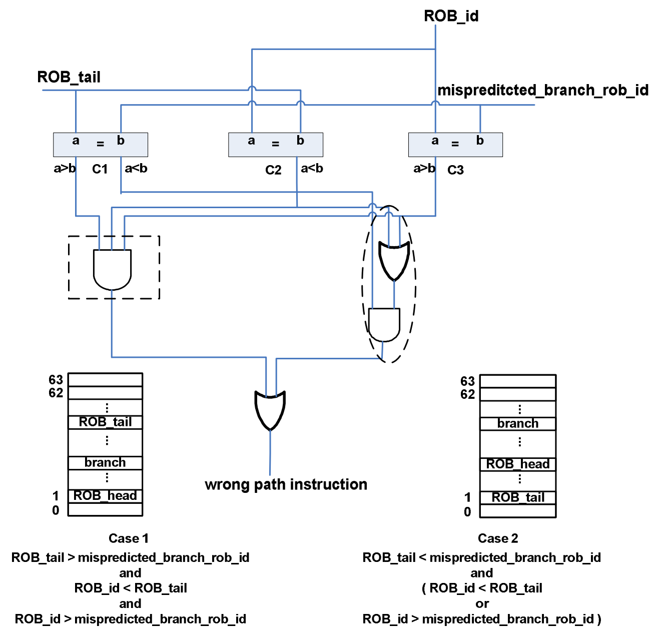

Figure 4. Circuitry used to detect wrong-path instructions.

gate the first stage register set of the FU. Note that when the branch is not mispredicted, the added circuitry is functionally equivalent to the PFCG architecture (cf. Figure 2) and consumes minimal power since there will be no switching activity in the comparators.

When the head of the ROB reaches the mispredicted branch, we will flush the ROB and the pipeline. At that time, the misprediction bit will be reset so that starting with the next clock cycle, the WPCG is disabled.

It is important to emphasize the fact that, in out-of-order processors all types of instructions can be potentially executed out of order, and therefore, branches can also be executed out of order. Hence, once we detect a branch misprediction and update the MBR_id register and set the misprediction bit to '1', it is possible that an older branch gets executed and gets resolved to be mispredicted. An older branch can still be issued and executed since it falls into the correct path with respect to the mispredicted branch whose ROB ID is stored in the MBR_id register. Therefore, if an older branch is resolved to be mispredicted, we should update the MBR_id register with the ROB ID of the just-resolved older branch since updating the MBR_id register with this new branch will cover more wrong-path instructions. For the sake of completeness we mention that if a younger branch gets resolved to be mispredicted, then we do not alter the content of MBR_id register. Note however that this scenario is not possible since if a branch is younger than the branch whose ROB ID is in the MBR_id register, then the younger branch will fall into the category of wrong-path instructions with respect to the branch whose ROB ID is in MBR_id register. Thus if a branch is resolved to be mispredicted while the misprediction bit is set to '1', then this newly mispredicted branch must be older and we update the MBR_id register. Since we update the MBR_id register any time a branch is mispredicted, we are already taking care of this scenario.

Furthermore, it is possible that more than one branch gets resolved to be mispredicted in the same cycle. In this case, ideally, we would like to select the branch that is the oldest and update MBR_id register with the ROB ID of that branch. But this would require comparison between the ROB IDs of all the branches that are resolved to be mispredicted in the same cycle. Our simulation results show that, on average, only 6.25% of the total mispredicted branches are resolved in the same cycle. Therefore, in order to avoid the overhead of multiple range comparators, we select only one of the mispredicted branches from one of the Branch Execution Units with a predefined priority.

C. *Hardware Overhead*

Figure 4 shows the design of the range comparator block used in the WPCG architecture. As shown in the figure we actually need 3 comparators. This is because the ROB is a circular queue where the head of the ROB points to the earliest (oldest) instruction whereas the tail of the ROB points to the latest (youngest) instruction.

Due to this circular queue structure, we must deal with two different scenarios in order to determine whether the instruction being issued is a wrong-path instruction or not. For this purpose, we use three comparators. Comparator C1 compares the tail of the ROB with the ROB ID of the mispredicted branch. Comparator C2 compares the ROB ID of the instruction being issued (ROB_id) with the tail of the ROB whereas comparator C3 compares the ROB ID of the instruction being issued with the ROB ID of the mispredicted branch. Essentially we want to determine if the ROB ID of the instruction being issued is in between the mispredicted branch and ROB_tail. If so, the ROB ID belongs to the wrong-path instruction since the instructions following the branch are from the mispredicted path. As shown in the Figure 4 there are two possible scenarios:

- Case 1: ROB_tail is larger than the mispredicted branch's ROB ID (mispredicted_branch_rob_id in Figure 4). In this case the instruction being issued is on the wrong-path exactly if its ROB ID is larger than the mispredicted_branch_rob_id ***and*** smaller than the ROB_tail. This task is accomplished by the AND gate in the dotted rectangle.
- Case 2: ROB_tail is smaller than the mispredicted_branch_rob_id. In this case the instruction being issued is on the wrong-path exactly if its ROB ID is larger than the mispredicted_branch_rob_id ***or*** it is smaller than the ROB_tail. This task is accomplished by the gates in dotted oval.

Notice that the inputs of the comparators do not switch when the branch is not mispredicted. This is due to the fact that the ROB_tail and mispredicted_branch_rob_id registers (cf. Figure 3) are updated only in the event of misprediction. Therefore, they do not consume any power during the correct path execution. We implemented this circuit in Hspice and carried out the energy overhead analysis. The results presented in experimental section account for this overhead.

D. *Timing Overhead*

Potentially there can be a timing penalty for routing the misprediction bit and the mispredicted_branch_rob_id from the Execution stage back to the Issue stage. In the conventional processor implementations the branch misprediction information is sent to the Fetch and the Commit stages and the additional routing cost to get it to the Issue stage could be quite low. Hence we expect that this additional reverse signal path to have little or no impact on the clock cycle time. If, however, this becomes a concern, then we can also pipeline the reverse routing path for the misprediction bit signal from the Execution Unit to the Issue Logic; this will allow some wrong-path instructions to be issued into the pipeline, which reduces the energy savings of the WPCG technique, but will have no other performance or functional effects.

More generally, the WPCG architecture adds some logic to determine if the instruction is a wrong-path instruction, and thus, it adds some delay although the impact of this delay on the clock cycle time depends on which pipeline stage is the most timing critical one. In the worst case scenario, we must pipeline the issue logic, resulting in an extra clock cycle penalty for detecting wrong-path instructions. This additional stage will be bypassed when the branches are predicted correctly and therefore the penalty reduces to the Mux delay without any extra clock cycle penalty. In our simulations we pipelined this logic to account for the worst case scenario when the delay of the logic is too high to be accommodated within the same cycle of the issue. Therefore simulation results account for the associated performance penalty and are presented in experimental section.

IV. EXPERIMENTAL RESULTS

To carry out the evaluation of the proposed clock gating scheme, we used a simplescalar-based simulation platform. The PFCG and WPCG methods were implemented in *simplescalar* [7] with appropriate modifications to simplescalar to implement realistic branch execution. The processor model used for the evaluations is described in Table 1 . The benchmarks used for the evaluation included a few integer SPEC 2000 benchmarks (bzip, gzip, gcc) and a few floating point SPEC 2000 benchmarks (wupwise, apsi, mesa, equake) [8] along with a couple of multimedia benchmarks (djpeg, cjpeg) [9] . A subset of benchmarks was chosen which exhibits the same average branch prediction rate as that of the full suite it is representing. All benchmarks were run by fast forwarding 300M instructions followed by cycle accurate out of order simulation of 1B instructions. From simplescalar simulations, we obtained the access counts for various structures such as the integer functional units, RF, and caches.

Table 1 : Processor Model used for Evaluations.

Processor	Fetch, Decode, Issue and Commit: 4
ROB	128/64
LSQ	64/32
Caches	L1 I/D Cache 64KB 2-way, Hit Latency : 1-cycle, Unified L2 Cache of 2MB, 8-way, Hit Latency : 12-cycles
Memory Latency	100 cycles
Branch Predictor	Gshare predictor with table size: 4096 BTB 1024 2
Functional Units	Integer ALUs:4 Integer Multiplier/Dividers:2

To report the energy savings of the proposed clock gating scheme (while accounting for the overhead of the added circuitry), we used Hspice-based simulations using a 45nm CMOS technology obtained from the predictive technology models (PTM) [10]. Input registers of different stages of an FU were modeled as master-slave Flip Flops, implemented at the transistor-level, and simulated with Hspice to obtain the energy consumption when the clock is not gated as well as when the clock is gated. Furthermore to model a typical integer ALU, we designed and implemented a 32-bit adder, assuming for simplicity that an integer ALU consists of an adder, at transistor level and simulated it with Hspice. In order to obtain the energy consumption in the adder circuit, we divided the average switching activity per bit of the adder input operands into four ranges: [0, 25%), [25%, 50%), [50%, 75%) and [75%, 100%]. The corresponding energy consumptions were obtained by Hspice by performing Monte Carlo simulation of the adder circuit under appropriate bit-level switching activities taken from Simplescalar simulations. More precisely, we obtained the average bit-level switching activities for inputs of various integer ALUs in the target processor from simplescalar

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.