The Case for a Single-Chip Multiprocessor

Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
http://www-hydra.stanford.edu

Abstract

Advances in IC processing allow for more microprocessor design options. The increasing gate density and cost of wires in advanced integrated circuit technologies require that we look for new ways to use their capabilities effectively. This paper shows that in advanced technologies it is possible to implement a single-chip multiprocessor in the same area as a wide issue superscalar processor. We find that for applications with little parallelism the performance of the two microarchitectures is comparable. For applications with large amounts of parallelism at both the fine and coarse grained levels, the multiprocessor microarchitecture outperforms the superscalar architecture by a significant margin. Single-chip multiprocessor architectures have the advantage in that they offer localized implementation of a high-clock rate processor for inherently sequential applications and low latency interprocessor communication for parallel applications.

1 Introduction

Advances in integrated circuit technology have fueled microprocessor performance growth for the last fifteen years. Each increase in integration density allows for higher clock rates and offers new opportunities for microarchitectural innovation. Both of these are required to maintain microprocessor performance growth. Microarchitectural innovations employed by recent microprocessors include multiple instruction issue, dynamic scheduling, speculative execution and non-blocking caches. In the future, the trend seems to be towards CPUs with wider instruction issue and support for larger amounts of speculative execution. In this paper, we argue against this trend. We show that, due to fundamental circuit limitations and limited amounts of instruction level parallelism, the superscalar execution model will provide diminishing returns in performance for increasing issue width. Faced with this situation, building a complex wide issue superscalar CPU is not the most efficient use of silicon resources. We present the case that a better use of silicon area is a multiprocessor microarchitecture constructed from simpler processors.

ASPLOS VII 10/96 MA, USA
© 1996 ACM 0-89791-767-7/96/0010...$3.50

To understand the performance trade-offs between wide-issue processors and multiprocessors in a more quantitative way, we compare the performance of a six-issue dynamically scheduled superscalar processor with a 4 × two-issue multiprocessor. Our comparison has a number of unique features. First, we accurately account for and justify the latencies, especially the cache hit time, associated with the two microarchitectures. Second, we develop floor-plans and carefully allocate resources to the two microarchitectures so that they require an equal amount of die area. Third, we evaluate these architectures with a variety of integer, floating point and multiprogramming applications running in a realistic operating system environment.

The results show that on applications that cannot be parallelized, the superscalar microarchitecture performs 30% better than one processor of the multiprocessor architecture. On applications with fine grained thread-level parallelism the multiprocessor microarchitecture can exploit this parallelism so that the superscalar microarchitecture is at most 10% better. On applications with large grained thread-level parallelism and multiprogramming workloads the multiprocessor microarchitecture performs 50–100% better than the wide superscalar microarchitecture.

The remainder of this paper is organized as follows. In Section 2, we discuss the performance limits of superscalar design from a technology and implementation perspective. In Section 3, we make the case for a single chip multiprocessor from an applications perspective. In Section 4, we develop floor plans for a six-issue superscalar microarchitecture and a 4 × two-issue multiprocessor and examine their area requirements. We describe the simulation methodology used to compare these two microarchitectures in Section 5, and in Section 6 we present the results of our performance comparison. Finally, we conclude in Section 7.

2 The Limits of the Superscalar Approach

A recent trend in the microprocessor industry has been the design of CPUs with multiple instruction issue and the ability to execute instructions out of program order. This ability, called dynamic scheduling, first appeared in the CDC 6600 [21]. Dynamic scheduling uses hardware to track register dependencies between instructions; an instruction is executed, possibly out of program order, as soon as all of its dependencies are satisfied. In the CDC 6600 the register dependency checking was done with a hardware structure called the *scoreboard*. The IBM 360/91 used register renaming to improve the efficiency of dynamic scheduling using hardware struc-

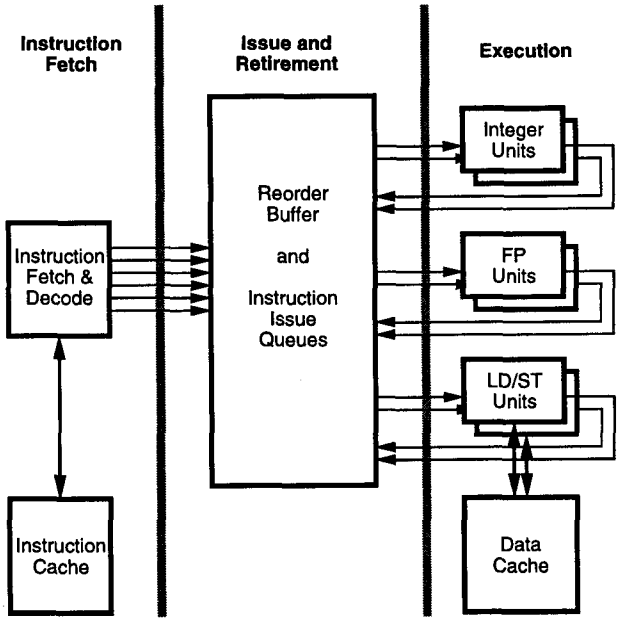

Figure 1. A dynamic superscalar CPU

tures called reservation stations [3]. It is possible to design a dynamically scheduled superscalar microprocessor using reservation stations; Johnson gives a thorough description of this approach [13]. However, the most recent implementations of dynamic superscalar processors have used a structure similar to the one shown in Figure 1. Here register renaming between architectural and physical registers is done explicitly, and instruction scheduling and register dependency tracking between instructions are performed in an instruction issue queue. Examples of microprocessors designed in this manner are the MIPS Technologies R10000 [24] and the HP PA-8000 [14]. In these processors the instruction queue is actually implemented as multiple instruction queues for different classes of instructions (e.g. integer, floating point, load/store). The three major phases of instruction execution in a dynamic superscalar machine are also shown in Figure 1. They are fetch, issue and execute. In the rest of this section we describe these phases and the limitations that will arise in the design of a very wide instruction issue CPU.

The goal of the fetch phase is to present the rest of the CPU with a large and accurate window of decoded instructions. Three factors constrain instruction fetch: mispredicted branches, instruction misalignment, and cache misses. The ability to predict branches correctly is crucial to establishing a large, accurate window of instructions. Fortunately, by using a moderate amount of memory (64Kbit), branch predictors such as the selective branch predictor proposed by McFarling are able to reduce misprediction rates to under 5% for most programs [15]. However, good branch prediction is not enough. As Conte pointed out, it is also necessary to align a *packet* of instructions for the decoder [7]. When the issue width is wider than four instructions there is a high probability that it will be necessary to fetch across a branch for a single packet of instructions since, in integer programs, one in every five instructions is a branch [12]. This will require fetching from two cache lines at once and merging the cache lines together to form a single packet of instructions. Conte describes a number of methods for achieving this. A technique that divides the instruction cache into banks and fetches from multiple banks at once is not too expensive to implement and provides performance that is within 3% of a perfect scheme on an 8-wide issue machine. Even with good branch prediction and alignment a significant cache miss rate will limit the ability of the fetcher to maintain an adequate window of instructions. There are still some applications such as large logic simulations, transactions processing and the OS kernel that have significant instruction cache miss rates even with fairly large 64 KB two way set-associative caches [19]. Fortunately, it is possible to hide some of the instruction cache miss latency in a dynamically scheduled processor by executing instructions that are already in the instruction window. Rosenblum *et. al.* have shown that over 60% of the instruction cache miss latency can be hidden on a database benchmark with a 64KB two way set associative instruction cache [19]. Given good branch prediction and instruction alignment it is likely that the fetch phase of a wide-issue dynamic superscalar processor will not limit performance.

In the issue phase, a packet of renamed instructions is inserted into the instruction issue queue. An instruction is issued for execution once all of its operands are ready. There are two ways to implement renaming. One could use an explicit table for mapping architectural registers to physical registers, this scheme is used in the R10000 [24], or one could use a combination reorder buffer/instruction queue as in the PA-8000 [14]. The advantage of the mapping table is that no comparisons are required for register renaming. The disadvantage of the mapping table is that the number of access ports required by the mapping table structure is $O \times W$, where O is the number of operands per instruction and W is the issue width of the machine. An eight-wide issue machine with three operands per instruction requires a 24 port mapping table. Implementing renaming with a reorder buffer has its own set of drawbacks. It requires $n \times Q \times O \times W$ 1-bit comparators to determine which physical registers should supply operands for a new packet of instructions, where n is the number of bits required to encode a register identifier and Q is the size of the instruction issue queue. Clearly, the number of comparators grows with the size of the instruction queue and issue width. Once an instruction is in the instruction queue, all instructions that issue must update their dependencies. This requires another set of $n \times Q \times O \times W$ comparators. For example, a machine with eight wide issue, three operand instructions, a 64-entry instruction queue, and 6-bit comparisons requires 9,216 1-bit comparators. The net effect of all the comparison logic and encoding associated with the instruction issue queue is that it takes a large amount of area to implement. On the PA-8000, which is a four-issue machine with 56 instruction issue queue entries, the instruction issue queue takes up 20% of the die area. In addition, as issue widths increase, larger windows of instructions are required to find independent instructions that can issue in parallel and maintain the full issue bandwidth. The result is a quadratic increase in the size of the instruction issue queue. Moving to the circuit level, the instruction issue queue uses a broadcast mechanism to communicate the tags of the instructions that are issued, which requires wires that span the length of the structure. In future advanced integrated circuit technologies these wires will have increasingly long delays relative to the gates that drive them [9]. Given this situation, ultimately, the instruction issue queue will limit the cycle time of the processor. For these reasons we believe that the instruction issue

queue will fundamentally limit the performance of wide issue superscalar machines.

In the execution phase, operand values are fetched from the register file or bypassed from earlier instructions to execute on the functional units. The wide superscalar execution model will encounter performance limits in the register file, in the bypass logic and in the functional units. Wider instruction issue requires a larger window of instructions, which implies more register renaming. Not only must the register file be larger to accommodate more renamed registers, but the number of ports required to satisfy the full instruction issue bandwidth also grows with issue width. Again, this causes a quadratic increase in the complexity of the register file with increases in issue width. Farkas *et. al.* have investigated the effect of register file complexity on performance [10]. They find that an eight-issue machine only performs 20% better than a four-issue machine when the effect of cycle-time is included in the performance estimates. The complexity of the bypass logic also grows quadratically with number of execution units; however, a more limiting factor is the delay of the wires that interconnect the execution units. As far as the execution units themselves are concerned, the arithmetic functional units can be duplicated to support the issue width, but more ports must be added to the primary data cache to provide the necessary load/store bandwidth. The cheapest way to add ports to the data cache is by building a banked cache [20], but the added multiplexing and control required to implement a banked cache increases the access time of the cache. We investigate this issue in more detail in Section 4.2.

3 The Case for a Single-Chip Multiprocessor

The motivation for building a single chip multiprocessor comes from two sources; there is a technology push and an application pull. We have already argued that technology issues, especially the delay of the complex issue queue and multi-port register files, will limit the performance returns from a wide superscalar execution model. This motivates the need for a decentralized microarchitecture to maintain the performance growth of microprocessors. From the applications perspective, the microarchitecture that works best depends on the amount and characteristics of the parallelism in the applications.

Wall has performed one of the most comprehensive studies of application parallelism [22]. The results of his study indicate that applications fall in two classes. The first class consists of applications with low to moderate amounts of parallelism; under ten instructions per cycle with aggressive branch prediction and large, but not infinite window sizes. Most of these applications are integer applications. The second class consists of applications with large amounts of parallelism, greater than forty instructions per cycle with aggressive branch prediction and large window sizes. The majority of these applications are floating point applications and most of the parallelism is in the form of loop-level parallelism.

The application pull towards a single-chip multiprocessor arises because these two classes of applications require different execution models. Applications in the first class work best on processors that are moderately superscalar (2 issue) with very high clock rates because there is little parallelism to exploit. To make this more concrete we note that a 200 MHz MIPS R5000, which is a single issue machine when running integer programs, achieves a SPEC95 integer rating which is 70% of the rating of a 200 MHz MIPS R10000, which is a four-issue machine [6]. Both machines have the same size data and instruction caches, but the R5000 has a blocking data cache, while the R10000 has a non-blocking data cache. Applications in the second class have large amounts of parallelism and see performance benefits from a variety of methods designed to exploit parallelism such as superscalar, VLIW or vector processing. However, the recent advances in parallel compilers make a multiprocessor an efficient and flexible way to exploit the parallelism in these programs [1]. Single-chip multiprocessors, designed so that the individual processors are simple and achieve very high clock rates, will work well on integer programs in the first class. The addition of low latency communication between processors on the same chip also allows the multiprocessor to exploit the parallelism of the floating point programs in the second class. In Section 6 we evaluate the performance of a single-chip multiprocessor for these two application classes.

There are a number of ways to use a multiprocessor. Today, the most common use is to execute multiple processes in parallel to increase throughput in a multiprogramming environment under the control of a multiprocessor aware operating system. We note that there are a number of commercially available operating systems that have this capability (e.g. Silicon Graphics IRIX, Sun Solaris, Microsoft Windows NT). Furthermore, the increasingly widespread use of visualization and multimedia applications tends to increase the number of active processes or independent threads on a desktop machine or server at a particular point in time.

Another way to use a multiprocessor is to execute multiple threads in parallel that come from a single application. Two examples are transaction processing and hand parallelized floating point scientific applications [23]. In this case the threads communicate using shared memory, and these applications are designed to run on parallel machines with communication latencies in the hundreds of CPU clock cycles; therefore, the threads do not communicate in a very fine grained manner. Another example of manually parallelized applications are fine-grained thread-level integer applications. Using the results from Wall's study, these applications exhibit moderate amounts of parallelism when the instruction window size is very large and the branch prediction is perfect because the parallelism that exists is widely distributed. Due to the large window size and the perfect branch prediction it will be very difficult for this parallelism could be extracted with a superscalar execution model. However, it is possible for a programmer that understands the nature of the parallelism in the application to parallelize the application into multiple threads. The parallelism exposed in this manner is fine-grained and cannot be exploited by a conventional multiprocessor architecture. The only way to exploit this type of parallelism is with a single-chip multiprocessor architecture.

A third way to use a multiprocessor is to accelerate the execution of sequential applications without manual intervention; this requires automatic parallelization technology. Recently, this automatic parallelization technology was shown to be effective on scientific applications [2], but it is not yet ready for general purpose integer applications. Like the manually parallelized integer applications, these applications could derive significant performance benefits from the low-latency interprocessor communication provided by a single-chip multiprocessor.

	6-way SS	4x2-way MP
# of CPUs	1	4
Degree superscalar	6	4 x 2
# of architectural registers	32int / 32fp	4 x 32int / 32fp
# of physical registers	160int / 160fp	4 x 40int / 40fp
# of integer functional units	3	4 x 1
# of floating pt. functional units	3	4 x 1
# of load/store ports	8 (one per bank)	4 x 1
BTB size	2048 entries	4 x 512 entries
Return stack size	32 entries	4 x 8 entries
Instruction issue queue size	128 entries	4 x 8 entries
I cache	32 KB, 2-way S. A.	4 x 8 KB, 2-way S. A.
D cache	32 KB, 2-way S. A.	4 x 8 KB, 2-way S. A.
L1 hit time	2 cycles (4 ns)	1 cycle (2 ns)
L1 cache interleaving	8 banks	N/A
Unified L2 cache	256 KB, 2-way S. A.	256 KB, 2-way S. A.
L2 hit time / L1 penalty	4 cycles (8 ns)	5 cycles (10 ns)
Memory latency / L2 penalty	50 cycles (100 ns)	50 cycles (100 ns)

Table 1. Key characteristics of the two microarchitectures

4 Two Microarchitectures

To compare the wide superscalar and multiprocessor design approaches, we have developed the microarchitectures for two machines that will represent the state of the art in processor design a few years from now. The superscalar microarchitecture (SS) is a logical extension of the current R10000 superscalar design, widened from the current four-way issue to a six-way issue implementation. The multiprocessor microarchitecture (MP), is a four-way single-chip multiprocessor composed of four identical 2-way superscalar processors. In order to fit four identical processors on a die of the same size, each individual processor is comparable to the Alpha 21064, which became available in 1992 [8].

These two extremely different microarchitectures have nearly identical die sizes when built in identical process technologies. The processor size we select is based upon the kinds of processor chips that advances in silicon processing technology will allow in the next few years. When manufactured in a 0.25 µm process, which should be possible by the end of 1997, each of the chips will have an area of 430 mm^2 — about 30% larger than leading-edge microprocessors being shipped today. This represents typical die size growth over the course of a few years among the largest, fastest microprocessors [11].

We have argued that the simpler two-issue CPU used in the multiprocessor microarchitecture will have a higher clock rate than the six issue CPU; however, for the purposes of this comparison we have assumed that the two processors have the same clock rate. To achieve the same clock rate the wide superscalar architecture would require deeper pipelining due to the large amount of instruction issue logic in the critical path. For simplicity, we ignore latency variations between the architectures due to the degree of pipelining. We assume the clock frequency of both machines is 500 MHz. At 500 MHz the main memory latencies experienced by the processor are large. We have modeled the main memory as a 50-cycle, 100 ns delay for both architectures, typical values in a workstation today with 60 ns DRAMs and 40 ns of delays due to buffering in the DRAM controller chips [25].

Table 1 shows the key characteristics of the two architectures. We explain and justify these characteristics in the following sections. The integer and floating point functional unit result and repeat latencies are the same as the R10000 [24]

4.1 6-Way Superscalar Architecture

The 6-way superscalar architecture is a logical extension of the current R10000 design. As the floorplan in Figure 2 and the area breakdown in Table 2 indicate, the logic necessary for out-of-order instruction issue and scheduling dominates the area of the chip, due to the quadratic area impact of supporting 6-way instruction issue. First, we increased the number of ports in the instruction buffers by 50% to support 6-way issue instead of 4-way, increasing the area of each buffer by about 30-40%. Second, we increased the number of instruction buffers from 48 to 128 entries so that the processor examines a larger window of instructions for ILP to keep the execution units busy. This large instruction window also compensates for the fact that the simulations do not execute code that is optimized for a 6-way superscalar machine. The larger instruction window size and wider issue width causes a quadratic area increase of the instruction sequencing logic to 3-4 times its original size. Altogether, the logic necessary to handle out-of-order instruction issue occupies about 120 mm^2 — about 30% of the die. In comparison, the actual execution units only occupy about 70 mm^2 — just 18% of the die is required to build triple R10000 execution units in a 0.25 µm process.

Due to the increased rate at which instructions are issued, we also enhanced the fetch logic by increasing the size of the branch target buffer to 2048 entries and the call-return stack to 32 entries. This increases the branch prediction accuracy of the processor and pre-

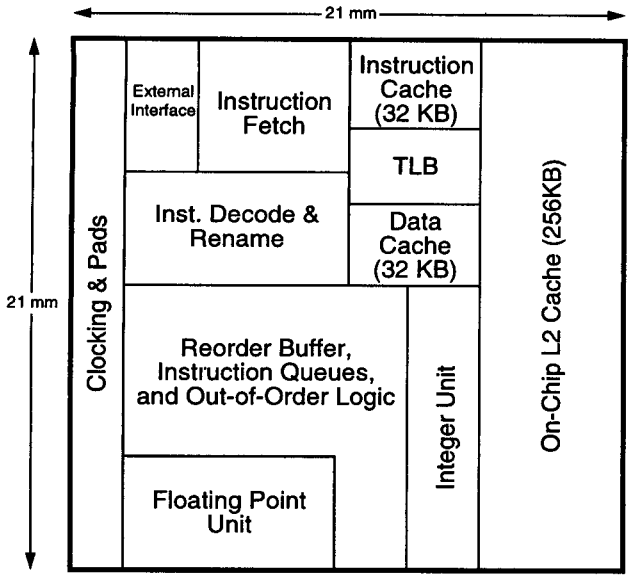

Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.

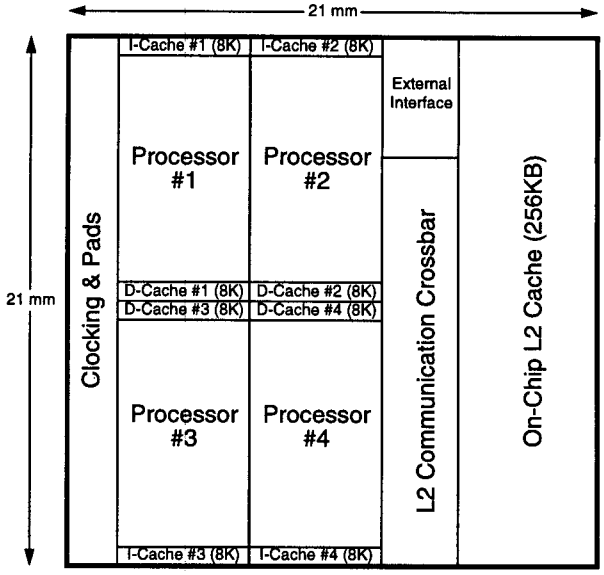

Figure 3. Floorplan for the four-way single-chip multiprocessor.

vents the instruction fetch mechanism from becoming a bottleneck since the 6-way execution engine requires a much higher instruction fetch bandwidth than the 2-way processors used in the MP architecture.

The on-chip memory hierarchy is similar to the Alpha 21164 — a small, fast level one (L1) cache backed up by a large on-chip level two (L2) cache. The wide issue width requires the L1 cache to support wide instruction fetches from the instruction cache and multiple loads from the data cache during each cycle. The two-way set associative 32 KB L1 data cache is banked eight ways into eight small, single-ported, independent 4 KB cache banks each of which handling one access every 2 ns processor cycle. However, the additional overhead of the bank control logic and crossbar required to arbitrate between the multiple requests sharing the 8 data cache banks adds another cycle to the latency of the L1 cache, and increases the area by 25%. Therefore, our modeled L1 cache has a hit time of 2 cycles. Backing up the 32 KB L1 caches is a large, unified, 256 KB L2 cache that takes 4 cycles to access. These latencies are simple extensions of the times obtained for the L1 caches of current Alpha microprocessors [4], using a 0.25 µm process technology

4.2 4 x 2-way Superscalar Multiprocessor Architecture

The MP architecture is made up of four 2-way superscalar processors interconnected by a crossbar that allows the processors to share the L2 cache. On the die, the four processors are arranged in a grid with the L2 cache at one end, as shown in Figure 3. Internally, each of the processors has a register renaming buffer that is much more limited than the one in the 6-way architecture, since each CPU only has an 8-entry instruction buffer. We also quartered the size of the branch prediction mechanisms in the fetch units, to 512 BTB entries and 8 call-return stack entries. After the area adjustments caused by these factors are accounted for, each of the four processors is less than one-fourth the size of the 6-way SS processor, as shown in Table 3. The number of execution units actually increases in the MP because the 6-way processor had three units of each type, while the 4-way MP must have four — one for each CPU. On the other hand, the issue logic becomes dramatically smaller, due to the decrease in instruction buffer ports and the smaller number of entries in each instruction buffer. The scaling factors of these two units balance each other out, leaving the entire processor very close to one-fourth of the size of the 6-way processor.

The on-chip cache hierarchy of the multiprocessor is significantly different from the cache hierarchy of the 6-way superscalar processor. Each of the 4 processors has its own single-banked and single-ported 8 KB instruction and data caches that can both be accessed in a single 2 ns cycle. Since each cache can only be accessed by a single processor with a single load/store unit, no additional overhead is incurred to handle arbitration among independent memory-access units. However, since the four processors now share a single L2 cache, that cache requires an extra cycle of latency during every access to allow time for interprocessor arbitration and crossbar delay. We model this additional L2 delay by penalizing the MP an additional cycle on every L2 cache access, resulting in a 5 cycle L2 hit time.

5 Simulation Methodology

Accurately evaluating the performance of the two microarchitectures requires a way of simulating the environment in which we would expect these architectures to be used in real systems. In this section we describe the simulation environment and the applications used in this study.

5.1 Simulation Environment

We execute the applications in the SimOS simulation environment [18]. SimOS models the CPUs, memory hierarchy and I/O devices

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.