The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo,
Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

DASH is a scalable shared-memory multiprocessor currently being developed at Stanford's Computer Systems Laboratory. The architecture consists of powerful processing nodes, each with a portion of the shared-memory, connected to a scalable interconnection network. A key feature of DASH is its distributed directory-based cache coherence protocol. Unlike traditional snoopy coherence protocols, the DASH protocol does not rely on broadcast; instead it uses point-to-point messages sent between the processors and memories to keep caches consistent. Furthermore, the DASH system does not contain any single serialization or control point. While these features provide the basis for scalability, they also force a reevaluation of many fundamental issues involved in the design of a protocol. These include the issues of correctness, performance and protocol complexity. In this paper, we present the design of the DASH coherence protocol and discuss how it addresses the above issues. We also discuss our strategy for verifying the correctness of the protocol and briefly compare our protocol to the IEEE Scalable Coherent Interface protocol.

1 Introduction

The limitations of current uniprocessor speeds and the ability to replicate low cost, high-performance processors and VLSI components have provided the impetus for the design of multiprocessors which are capable of scaling to a large number of processors. Two major paradigms for these multiprocessor architectures have developed, *message-passing* and *shared-memory*. In a message-passing multiprocessor, each processor has a local memory, which is only accessible to that processor. Interprocessor communication occurs only through explicit message passing. In a shared-memory multiprocessor, all memory is accessible to each processor. The shared-memory paradigm has the advantage that the programmer is not burdened with the issues of data partitioning, and accessibility of data from all processors simplifies the task of dynamic load distribution. The primary advantage of the message passing systems is the ease with which they scale to support a large number of processors. For shared-memory machines providing such scalability has traditionally proved difficult to achieve.

We are currently building a prototype of a scalable shared-memory multiprocessor. The system provides high processor performance and scalability though the use of coherent caches and a directory-based coherence protocol. The high-level organization of the prototype, called DASH (Directory Architecture for SHared memory) [17], is shown in Figure 1. The architecture consists of a number of processing nodes connected through a high-bandwidth low-latency interconnection network. The physical memory in the machine is distributed among the nodes of the multiprocessor, with all memory accessible to each node. Each processing node, or *cluster*, consists of a small number of high-performance processors with their individual caches, a portion of the shared-memory, a common cache for pending remote accesses, and a directory controller interfacing the cluster to the network. A bus-based snoopy scheme is used to keep caches coherent within a cluster, while inter-node cache consistency is maintained using a distributed directory-based coherence protocol.

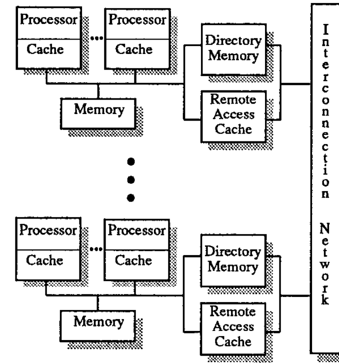

Figure 1: General architecture of DASH.

The concept of directory-based cache coherence was first proposed by Tang [20] and Censier and Feautrier [6]. Subsequently, it has been been investigated by others ([1],[2] and [23]). Building on this earlier work, we have developed a new directory-based cache-coherence protocol which works with distributed directories and the hierarchical cluster configuration of DASH. The protocol also integrates support for efficient synchronization operations using the directory. Furthermore, in designing the machine we have addressed many of the issues left unresolved by earlier work.

In DASH, each processing node has a directory memory corresponding to its portion of the shared physical memory. For each memory block, the directory memory stores the identities

CH2887-8/90/0000/0148$01.00 © 1990 IEEE

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

of all remote nodes caching that block. Using the directory memory, a node writing a location can send point-to-point invalidation or update messages to those processors that are actually caching that block. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of DASH depends on this ability to avoid broadcasts. Another important attribute of the directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, we can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were originally developed for message-passing machines [7].

While the design of bus-based snoopy coherence protocols is reasonably well understood, this is not true of distributed directory-based protocols. Unlike snoopy protocols, directory-based schemes do not have a single serialization point for all memory transactions. While this feature is responsible for their scalability, it also makes them more complex and forces one to rethink how the protocol should address the fundamental issues of correctness, system performance, and complexity.

The next section outlines the important issues in designing a cache coherence protocol. Section 3 gives an overview of the DASH hardware architecture. Section 4 describes the design of the DASH coherence protocol, relating it to the issues raised in section 2. Section 5 outlines some of the additional operations supported beyond the base protocol, while Section 6 discusses scaling the directory structure. Section 7 briefly describes our approach to verifying the correctness of the protocol. Section 8 compares the DASH protocol with the proposed IEEE-SCI (Scalable Coherent Interface) protocol for distributed directory-based cache coherence. Finally, section 9 presents conclusions and summarizes the current status of the design effort.

2 Design Issues for Distributed Coherence Protocols

The issues that arise in the design of any cache coherence protocol and, in particular, a distributed directory-based protocol, can be divided into three categories: those that deal with correctness, those that deal with the performance, and those related to the distributed control of the protocol.

2.1 Correctness

The foremost issue that any multiprocessor cache coherence protocol must address is correctness. This translates into requirements in three areas:

Memory Consistency Model: For a uniprocessor, the model of a correct memory system is well defined. Load operations return the last value written to a given memory location. Likewise, store operations bind the value returned by subsequent loads of the location until the next store. For multiprocessors, however, the issue is more complex because the definitions of "last value written", "subsequent loads" and "next store" become less clear as there may be multiple processors reading and writing a location. To resolve this difficulty a number of memory consistency models have been proposed in the literature, most notably, the sequential and weak consistency models [8]. Weaker consistency models attempt to loosen the constraints on the coherence protocol while still providing a reasonable programming model for the user. Although most existing systems utilize a relatively strong consistency model, the larger latencies found in a distributed system favor the less constrained models.

Deadlock: A protocol must also be deadlock free. Given the arbitrary communication patterns and finite buffering within the memory system there are numerous opportunities for deadlock. For example, a deadlock can occur if a set of transactions holds network and buffer resources in a circular manner, and the consumption of one request requires the generation of another request. Similarly, lack of flow control in nodes can cause requests to back up into the network, blocking the flow of other messages that may be able to release the congestion.

Error Handling: Another issue related to correctness is support for data integrity and fault tolerance. Any large system will exhibit failures, and it is generally unacceptable if these failures result in corrupted data or incorrect results without a failure indication. This is especially true for parallel applications where algorithms are more complex and may contain some non-determinism which limits repeatability. Unfortunately, support for data integrity and fault-tolerance within a complex protocol that attempts to minimize latency and is executed directly by hardware is difficult. The protocol must attempt to balance the level of data integrity with the increase in latency and hardware complexity. At a minimum, the protocol should be able to flag all detectable failures, and convey this information to the processors affected.

2.2 Performance

Given a protocol that is correct, performance becomes the next important design criterion. The two key metrics of memory system performance are latency and bandwidth.

Latency: Performance is primarily determined by the latency experienced by memory requests. In DASH, support for cachable shared data provides the major reduction in latency. The latency of write misses is reduced by using write buffers and by the support of the release consistency model. Hiding the latency for read misses is usually more critical since the processor is stalled until data is returned. To reduce the latency for read misses, the protocol must minimize the number of inter-cluster messages needed to service a miss and the delay associated with each such message.

Bandwidth: Providing high memory bandwidth that scales with the number of processors is key to any large system. Caches and distributed memory form the basis for a scalable, high-bandwidth memory system in DASH. Even with distributed memory, however, bandwidth is limited by the serialization of requests in the memory system and the amount of traffic generated by each memory request.

Servicing a memory request in a distributed system often requires several messages to be transmitted. For example, a message to access a remote location generates a reply message containing the data, and possibly other messages invalidating remote caches. The component with the largest serialization in this chain limits the maximum throughput of requests. Serialization affects performance by increasing the queuing delays, and thus the latency, of memory requests. Queueing delays can become critical for locations that exhibit a large degree of sharing. A protocol should attempt to minimize the service time at all queuing centers. In particular, in a distributed system no central resources within a node should be blocked while inter-node communication is taking place to service a request. In this way serialization is limited only by the time of local, intra-node operations.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

The amount of traffic generated per request also limits the effective throughput of the memory system. Traffic seen by the global interconnect and memory subsystem increases the queueing for these shared resources. DASH reduces traffic by providing coherent caches and by distributing memory among the processors. Caches filter many of the requests for shared data while grouping memory with processors removes private references if the corresponding memory is allocated within the local cluster. At the protocol level, the number of messages required to service different types of memory requests should be minimized, unless the extra messages directly contribute to reduced latency or serialization.

2.3 Distributed Control and Complexity

A coherence protocol designed to address the above issues must be partitioned among the distributed components of the multiprocessor. These components include the processors and their caches, the directory and main memory controllers, and the interconnection network. The lack of a single serialization point, such as a bus, complicates the control since transactions do not complete atomically. Furthermore, multiple paths within the memory system and lack of a single arbitration point within the system allow some operations to complete out of order. The result is that there is a rich set of interactions that can take place between different memory and coherence transactions. Partitioning the control of the protocol requires a delicate balance between the performance of the system and the complexity of the components. Too much complexity may effect the ability to implement the protocol or ensure that the protocol is correct.

3 Overview of DASH

Figure 2 shows a high-level picture of the DASH prototype we are building at Stanford. In order to manage the size of the prototype design effort, a commercial bus-based multiprocessor was chosen as the processing node. Each node (or *cluster*) is a Silicon Graphics POWER Station 4D/240 [4]. The 4D/240 system consists of four high-performance processors, each connected to a 64 Kbyte first-level instruction cache, and a 64 Kbyte write-through data cache. The 64 Kbyte data cache interfaces to a 256 Kbyte second-level write-back cache through a read buffer and a 4 word deep write-buffer. The main purpose of this second-level cache is to convert the write-through policy of the first-level to a write-back policy, and to provide the extra cache tags for bus snooping. Both the first and second-level caches are direct-mapped.

In the 4D/240, the second-level caches are responsible for bus snooping and maintaining consistency among the caches in the cluster. Consistency is maintained using the Illinois coherence protocol [19], which is an invalidation-based ownership protocol. Before a processor can write to a cache line, it must first acquire exclusive ownership of that line by requesting that all other caches invalidate their copy of that line. Once a processor has exclusive ownership of a cache line, it may write to that line without consuming further bus cycles.

The memory bus (MPBUS) of the 4D/240 is a pipelined synchronous bus, supporting memory-to-cache and cache-to-cache transfers of 16 bytes every 4 bus clocks with a latency of 6 bus clocks. While the MPBUS is pipelined, it is not a split transaction bus. Consequently, it is not possible to efficiently interleave long duration remote transactions with the short duration local

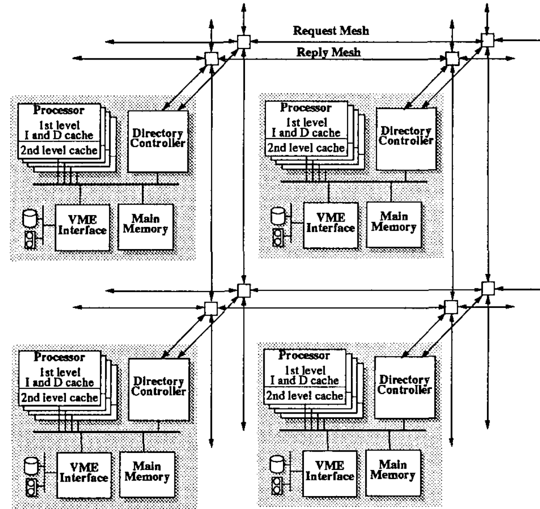

Figure 2: Block diagram of sample 2 x 2 DASH system.

transactions. Since this ability is critical to DASH, we have extended the MPBUS protocol to support a retry mechanism. Remote requests are signaled to retry while the inter-cluster messages are being processed. To avoid unnecessary retries the processor is masked from arbitration until the response from the remote request has been received. When the response arrives, the requesting processor is unmasked, retries the request on the bus, and is supplied the remote data.

A DASH system consists of a number of modified 4D/240 systems that have been supplemented with a directory controller board. This directory controller board is responsible for maintaining the cache coherence across the nodes and serving as the interface to the interconnection network.

The directory board is implemented on a single printed circuit board and consists of five major subsystems as shown in Figure 3. The *directory controller* (DC) contains the directory memory corresponding to the portion of main memory present within the cluster. It also initiates out-bound network requests and replies. The *pseudo-CPU* (PCPU) is responsible for buffering incoming requests and issuing such requests on the cluster bus. It mimics a CPU on this bus on behalf of remote processors except that responses from the bus are sent out by the directory controller. The *reply controller* (RC) tracks outstanding requests made by the local processors and receives and buffers the corresponding replies from remote clusters. It acts as memory when the local processors are allowed to retry their remote requests. The *network interface* and the local portion of the network itself reside on the directory card. The interconnection network consists of a pair of meshes. One mesh is dedicated to the request messages while the other handles replies. These meshes utilize *wormhole routing* [9] to minimize latency. Finally, the board contains *hardware monitoring logic* and miscellaneous control and status registers. The monitoring logic samples a variety of directory board and bus events from which usage and performance statistics can be derived.

The directory memory is organized as an array of directory

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

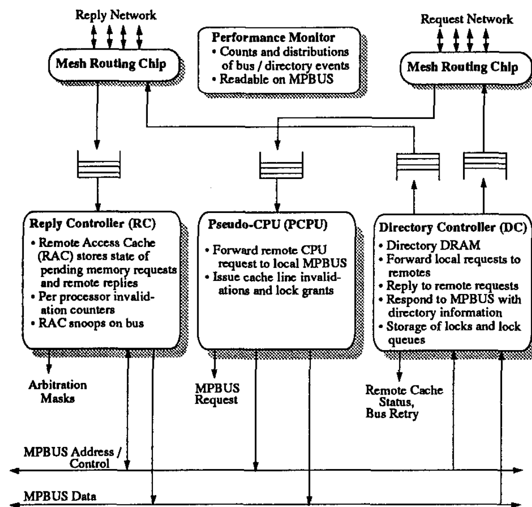

Figure 3: Directory board block diagram.

entries. There is one entry for each memory block. The directory entries used in the prototype are identical to that originally proposed in [6]. They are composed of a single state bit together with a bit vector of pointers to clusters. The state bit indicates whether the clusters have a read (shared) or read/write (dirty) copy of the data. The bit vector contains a bit for each of the sixteen clusters supported in the prototype. Associating the directory with main memory allows the directory to be built with the same DRAM technology as main memory. The DC accesses the directory memory on each MPBUS transaction along with the access to main memory. The directory information is combined with the type of bus operation, address, and result of the snooping within the cluster to determine what network messages and bus controls the DC will generate.

The RC maintains its state in the *remote access cache* (RAC). The functions of the RAC include maintaining the state of currently outstanding requests, buffering replies from the network and supplementing the functionality of the processors' caches. The RAC is organized as a snoopy cache with augmented state information. The RAC's state machines allow accesses from both the network and the cluster bus. Replies from the network are buffered in the RAC and cause the waiting processor to be released for bus arbitration. When the released processor retries the access the RAC supplies the data via a cache-to-cache transfer.

3.1 Memory Consistency in DASH

As stated in Section 2, the correctness of the coherence protocol is a function of the memory consistency model adopted by the architecture. There is a whole spectrum of choices for the level of consistency to support directly in hardware. At one end is the *sequential consistency* model [16] which requires the execution of the parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. As one moves towards weaker models of consistency, performance gains are made at the cost of a more complex programming model for the user.

The base model of consistency provided by the DASH hardware is called *release consistency*. Release consistency [10] is an extension of the weak consistency model first proposed by Dubois, Scheurich and Briggs [8]. The distinguishing characteristics of release consistency is that it allows memory operations issued by a given processor to be observed and complete out of order with respect to the other processors. The ordering of operations is only preserved before "releasing" synchronization operations or explicit ordering operations. Release consistency takes advantage of the fact that while in a critical region a programmer has already assured that no other processor is accessing the protected variables. Thus, updates to these variables can be observed by other processors in arbitrary order. Only before the lock release at the end of the region does the hardware need to guarantee that all operations have completed. While release consistency does complicate programming and the coherence protocol, it can hide much of the overhead of write operations.

Support for release consistency puts several requirements on the system. First, the hardware must support a primitive which guarantees the ordering of memory operations at specific points in a program. Such fence [5, 10] primitives can then be placed by software before releasing synchronization points in order to implement release consistency. DASH supports two explicit fence mechanisms. A *full-fence* operation stalls the processor until all of its pending operations have been completed, while a *write-fence* simply delays subsequent write-operations. A higher performance implementation of release consistency includes implicit fence operations within the releasing synchronization operations themselves. DASH supports such synchronization operations yielding release consistency as its base consistency model. The explicit fence operations in DASH then allow the user or compiler to synthesize stricter consistency models if needed.

The release consistency model also places constraints on the base coherence protocol. First, the system must respect the local dependencies generated by the memory operations of a single processor. Second, all coherence operations, especially operations related to writes, must be acknowledged so that the issuing processor can determine when a fence can proceed. Third, any cache line owned with pending invalidations against it can not be shared between processors. This prevents the new processor from improperly passing a fence. If sharing is allowed then the receiving processor must be informed when all of the pending invalidates have been acknowledged. Lastly, any operations that a processor issues after a fence operation may not become visible to any other processor until all operations preceding the fence have completed.

4 The DASH Cache Coherence Protocol

In our discussion of the coherence protocol, we use the following naming conventions for the various clusters and memories involved in any given transaction. A *local cluster* is a cluster that contains the processor originating a given request, while the *home cluster* is the cluster which contains the main memory and directory for a given physical memory address. A *remote cluster* is any other cluster. Likewise, *local memory* refers to the main memory associated with the local cluster while *remote memory* is any memory whose home is not the local.

The DASH coherence protocol is an invalidation-based own-

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

ership protocol. A memory block can be in one of three states as indicated by the associated directory entry: (i) *uncached-remote*, that is not cached by any remote cluster; (ii) *shared-remote*, that is cached in an unmodified state by one or more remote clusters; or (iii) *dirty-remote*, that is cached in a modified state by a single remote cluster. The directory does not maintain information concerning whether the home cluster itself is caching a memory block because all transactions that change the state of a memory block are issued on the bus of the home cluster, and the snoopy bus protocol keeps the home cluster coherent. While we could have chosen not to issue all transactions on the home cluster's bus this would had an insignificant performance improvement since most requests to the home also require an access to main memory to retrieve the actual data.

The protocol maintains the notion of an *owning cluster* for each memory block. The owning cluster is nominally the home cluster. However, in the case that a memory block is present in the dirty state in a remote cluster, that cluster is the owner. Only the owning cluster can complete a remote reference for a given block and update the directory state. While the directory entry is always maintained in the home cluster, a dirty cluster initiates all changes to the directory state of a block when it is the owner (such update messages also indicate that the dirty cluster is giving up ownership). The order that operations reach the owning cluster determines their global order.

As with memory blocks, a cache block in a processor's cache may also be in one of three states: invalid, shared, and dirty. The shared state implies that there may be other processors caching that location. The dirty state implies that this cache contains an exclusive copy of the memory block, and the block has been modified.

The following sections outline the three primitive operations supported by the base DASH coherence protocol: read, read-exclusive and write-back. We also discuss how the protocol responds to the issues that were brought up in Section 2 and some of the alternative design choices that were considered. We describe only the normal flow for the memory transactions in the following sections, exception cases are covered in section 4.6.

4.1 Read Requests

Memory read requests are initiated by processor load instructions. If the location is present in the processor's first-level cache, the cache simply supplies the data. If not present, then a cache fill operation must bring the required block into the first-level cache. A fill operation first attempts to find the cache line in the processor's second-level cache, and if unsuccessful, the processor issues a read request on the bus. This read request either completes locally or is signaled to retry while the directory board interacts with the other clusters to retrieve the required cache line. The detailed flow for a read request is given in Figure 7 in the appendix.

The protocol tries to minimize latency by using cache-to-cache transfers. The local bus can satisfy a remote read if the given line is held in another processor's cache or the remote access cache (RAC). The four processor caches together with the RAC form a five-way set associative (1.25 Mbyte) cluster cache. The effective size of this cache is smaller than a true set associative cache because the entries in the caches need not be distinct. The check for a local copy is initiated by the normal snooping when the read is issued on the bus. If the cache line is present in the shared state then the data is simply transferred over the bus to the requesting processor and no access to the remote home cluster is needed. If the cache line is held in a dirty state by a local processor, however, something must be done with the ownership of the cache line since the processor supplying the data goes to a shared state in the Illinois protocol used on the cluster bus. The two options considered were to: (i) have the directory do a sharing write-back to the home cluster; and (ii) have the RAC take ownership of the cache line. We chose the second option because it permits the processors within a cluster to read and write a shared location without causing traffic in the network or home cluster.

If a read request cannot be satisfied by the local cluster, the processor is forced to retry the bus operation, and a request message is sent to the home cluster. At the same time the processor is masked from arbitration so that it does not tie up the local bus. Whenever a remote request is sent by a cluster, a RAC entry is allocated to act as a placeholder for the reply to this request. The RAC entry also permits merging of requests made by the different processors within the same cluster. If another request to the same memory block is made, a new request will not be sent to the home cluster; this reduces both traffic and latency. On the other hand, an access to a different memory block, which happens to map to a RAC entry already in use, must be delayed until the pending operation is complete. Given that the number of active RAC entries is small the benefit of merging should outweigh the potential for contention.

When the read request reaches the home cluster, it is issued on that cluster's bus. This causes the directory to look up the status of that memory block. If the block is in an uncached-remote or shared-remote state the directory controller sends the data over the reply network to the requesting cluster. It also records the fact that the requesting cluster now has a copy of the memory block. If the block is in the dirty-remote state, however, the read request is forwarded to the owning, dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is sent directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory. The flow of messages for this case is shown in Figure 4.

As shown in Figure 4, any request not satisfied in the home cluster is forwarded to the remote cluster that has a dirty copy of the data. This reduces latency by permitting the dirty cluster to respond directly to the requesting cluster. In addition, this forwarding strategy allows the directory controller to simultaneously process many requests (i.e. to be multithreaded) without the added complexity of maintaining the state of outstanding requests. Serialization is reduced to the time of a single intra-cluster bus transaction. The only resource held while inter-cluster messages are being sent is a single entry in the originating cluster's RAC.

The downside of the forwarding strategy is that it can result in additional latency when simultaneous accesses are made to the same block. For example, if two read requests from different clusters are received close together for a line that is dirty remote, both will be forwarded to the dirty cluster. However, only the first one will be satisfied since this request will force the dirty cluster to lose ownership by doing a sharing writeback and changing its local state to read only. The second request will not find the dirty data and will be returned with a *negative acknowledge* (NAK) to its originating cluster. This NAK will force the cluster to retry its access. An alternative to the forwarding approach used by our protocol would have been to buffer the read request at the home cluster, have the home send

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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.