Concurrent Caching

Jay Nelson
DuoMark International, Inc.
http://www.duomark.com/

Abstract

A concurrent cache design is presented which allows cached data to be spread across a cluster of computers. The implementation separates persistent storage from cache storage and abstracts the cache behaviour so that the user can experiment with cache size and replacement policy to optimize performance for a given system, even if the production data store is not available. Using processes to implement cached objects allows for runtime configurability and adaptive use policies as well as parallelization to optimize resource access efficiency.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming

General Terms Design, Languages, Performance

Keywords concurrent cache, erlang

1. Introduction

Cache implementations date to the earliest days of computer programming. Their importance continues due to the inherent speed difference which has remained between permanent storage and transient storage. In its typical form, a cache consists of an index to locate a specific datum and a storage area containing a currently active subset of the collection of data whose access is being speeded, which together are controlled by a set of functions for accessing the compound data structure. Requests for a datum are checked against the cache index and returned if present, otherwise the desired datum is loaded from its original slow location, stored in the cache and returned to the caller. As the cache approaches capacity the new datum may take the place of an existing datum based on a replacement policy which heuristically selects an item that is likely to not be needed soon [1].

Traditionally the data structure and code were entangled on a single processing node with one or more threads accessing the cache behavior. While this construction yielded a great leap forward in performance, it left the entire cache vulnerable to a single failure and restricted the size to the capacity of the compute node.

This paper describes an alternate design couched in a network of processes which spread the cache to leverage multiple processing nodes if available and isolate faults to reduce their impact on the cache as a whole. An implementation of the design in `erlang` is discussed showing the advantages that Concurrency-Oriented Programming Languages (COPLs) [2] provide for this class of problem. Finally, a framework is provided for experimenting with capacity sizing constraints, cache element replacement policies and other aspects of performance. Future areas of research are identified in the implementation of concurrent caches.

2. Design of a Concurrent Cache

Using `erlang` or another COPL affords an implementation of caching that relies on processes rather than data structures to organize the cached data. By placing each cached datum in a separate process, several benefits accrue:

- Each datum can respond actively, independently and in parallel to requests
- The total size of cached data may exceed the memory of a single compute node
- An idle datum may be easily removed by terminating its process
- Stale data may be replaced in parallel when multiple changes to the data store occur
- Replacement policies may be implemented independently from the cache itself
- A cached datum may dynamically migrate to the CPU that best responds to requests

Several assumptions were made for the implementation presented here. Removing any of them may have significant consequences on the results presented:

- Each datum can be represented by a single process in the available memory on every node in the cache cluster
- The speed of the cluster network is much faster than the original store, and is likely much faster than the user's access to the cluster
- All nodes in the cluster may obtain access to the permanent data

The example problem domain envisioned is serving dynamic web pages or database objects to users requesting the data across the Internet. The services are provided using a local area network of possibly disparate computers. The approach presented here should adapt well to other situations where a pool of resources may be spread across a cluster of computers and must be allocated efficiently to provide a responsive, coherent system.

2.1 The Canonical Problem

Given a webserver and a large set of documents which reside on disk, provide an interface that efficiently serves the documents to requesters without knowing *a priori* the sequence of access. Here we are concerned only with providing the data to another application, not the mechanism of implementing the webserver.

Figure 1 shows the relationship among elements of a system solving the canonical problem. Data flows from storage to cache processes when requested by external code. The requester may

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang '06 September 16, 2006, Portland, Oregon, USA
Copyright © 2006 ACM 1-59593-490-1/06/0009...$5.00

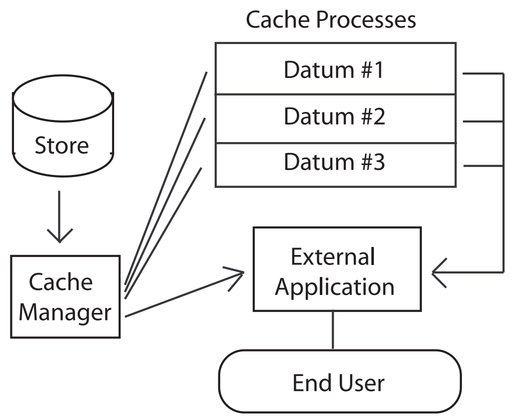

Figure 1. Relationship of elements of the canonical system.

then reformat the data for display to a user or to be forwarded to another software component. This approach strives to achieve a *separation of concerns* [3] whereby the storage format, cache format and output format are all independent of one another.

The separate cache processes are monitored by a cache manager which maintains the location of the actual cache object. To avoid funneling all the cached data through the manager, it relays the process id to the requester so that a connection may be established directly. When a cache process terminates, the manager is notified and the location index is updated. If a request is made for a datum that is not currently cached, the manager initiates the cache process and then replies to the external request.

2.2 Active Cache Datum

Each cached datum operates as a service which can field requests. The most commonly fielded request is to return the datum. The cache manager may request that the datum be replaced or that the datum should use a different expiration time period.

When the datum process is instantiated, it is provided a Time To Live (TTL) parameter which defines how long it may remain in the cache as an idle process. If it has not fielded any requests in that time period it may safely expire. Whenever a request is handled, the expiration clock is reset so that a frequently accessed datum will expire when it is no longer popular. The TTL parameter should be short enough to keep the cache small, but at least several times longer than the amount of time it takes to reconstruct a new datum process including retrieving and formatting the datum for use.

By allowing each datum to expire independently, a simple mechanism for pruning the cache is built in without extra code logic. The mechanism can become dynamic so that each datum can have an expiration behavior tailored to its characteristics. This is a major advance that is independent of the replacement policy used. Since each datum can also expire when requested by the cache manager by changing the TTL to zero, the replacement policy mainly consists of deciding which datum processes should expire and notifying them to do so.

Active datum processes reduce the likelihood of a full cache and simplify the implementation of a replacement policy to handle the cases when the cache is consuming too many resources.

3. Skeleton Implementation

A sketch of the code needed to implement the design is presented here along with the reasoning for its structure. The discussion uses `erlang` examples but is intended to be described in a way that would allow easy implementation in any COPL system. The full implementation is available as an Internet download [4].

3.1 Minimal Datum Server

Each datum process acts as a server replying to client requests. It stores a single datum which may be any data element supported by the language. In `erlang` it may be represented as an atom, number, string, binary, function, list, tuple, or record (the latter three provide the means to form compound data structures containing elements of any of type, potentially recursively), essentially any arbitrary data. The granularity and format of the datum is decided by the application writer and implemented by the data access module. The module is passed as an argument to the newly constructed datum process to remove all knowledge of the source data format from the implementation of the cache or its elements.

The body of the server function consists of a `receive` statement that handles incoming requests. A simple version of the datum server `receive` loop is shown below in Listing 1.

```
-record(datum_state, {id, mgr, data, ttl}).

datum_loop(#datum_state{id=Id,mgr=Mgr,data=Data,ttl=TTL}=State)
              when is_pid(Mgr) ->
  receive

    %% Cache manager requests data to be replaced...
    {new_data, Mgr, Replacement} ->
      Mgr ! {new_data, self(), Data},
      datum_loop(State#datum_state{data=Replacement});

    %% Cache manager indicates new Time To Live (TTL)...
    {new_ttl, Mgr, Replacement} ->
      Mgr ! {new_ttl, self(), TTL},
      datum_loop(State#datum_state{ttl=Replacement});

    %% Invalid cache manager request, causes code load...
    {InvalidMgrRequest, Mgr, _OtherData} ->
      Mgr ! {InvalidMgrRequest,self(),invalid_creator_request},
      cache:datum_loop(State);

    %% Request for cached data...
    {get, From} when is_pid(From) ->
      From ! {get, self(), Data},
      datum_loop(State);

    %% Invalid request, causes code load...
    {InvalidRequest, From} when is_pid(From) ->
      From ! {InvalidRequest, self(), invalid_request},
      cache:datum_loop(State);

    %% Any other request is ignored, but causes code load.
     _ -> cache:datum_loop(State)

  after
    %% Time To Live or TTL (integer or 'infinity').
    TTL ->
       TerminateReason = timeout,
       io:format(
           "Cache object ~p owned by ~p freed due to a ~w~n",
           [Id, self(), TerminateReason])
  end.
```

Listing 1. Receive loop of a cache datum process.

To ensure proper management of the message queue, requests from the cache manager are 3-element tuples while requests from clients are 2-element tuples. This approach helps guarantee that the code handles all messages received properly, clears the queue of improper requests and responds with specific relevant errors if necessary. It should also be efficient as the matcher will check the size of the tuple before attempting to unify with any of the pattern elements as it traverses the message queue. The wildcard case in each half of the `receive` statement is used to prevent the mailbox queue from getting clogged with unhandled messages of either type.

When the datum process is spawned, the `datum_state` record instance created holds the process id of the cache manager in the `mgr` field. The first three statements in the `receive` listing above match only when the requester is the cache manager. A request for `new_data` delivers a replacement for the internal datum element causing the old one to be returned and discarded. The second pattern handles `new_ttl` which causes replacement of the TTL parameter in the same manner. The third clause is the wildcard that consumes all other triplets delivered by the cache manager, returning an error to the caller.

The next two clauses cover the case of any client, including the cache manager. The `get` request returns a copy of the datum element. The second clause is a wildcard that catches all other doublets delivered by clients, returning an error to the caller.

The last clause is a catch all pattern for any other request (*i.e.*, not triplets requested by the cache manager nor doublets requested by any process). Since the code cannot determine which argument signifies the requester, it cannot return an error message to the client.

One other point to note is that in each of the error clauses, the `datum_loop` function is called with the module name specified so that a new version of the function will be loaded from disk if present. This approach allows new clauses to be added to the loop which may be introduced to the executing environment by sending any unrecognized message.

The `after` clause of the `receive` statement is used for the timed expiration of the process. It comes for free as an element of the language, automatically reducing the memory consumed by the cache process when it has been idle for too long.

3.2 Cache Manager

The cache manager is implemented as a separate process that can spawn cache datum processes as shown in Listing 2 below. It contains the index of currently active processes, a monitor on each so that it can remove processes from the index when they terminate and the ability to spawn a new entry which is added to the index after initialization. It uses a passed in module name and function name to access the data store and retrieve the desired datum to be cached. The mechanism of the cache manager is completely independent of the format of the data store, its access methods or the structure of a single datum element.

The index of processes are stored in an `ets` table using entries consisting of `{Datum_Id, Datum_Process_Id}`. The datum id is a unique key across all objects in the data store that may be cached, however, it may consist of any `erlang` term including compound ones.

The first clause of the `receive` handles datum processes that have signaled that they are no longer executing. The corresponding index entry is simply removed.

The second clause handles all requests to locate cached data. Either the process already exists, a new one is launched and added to the index or there is some error that prevents a new process from being launched. In the first two cases, the client is informed of the process id and in the error case the client is told that the request failed.

A wildcard clause catches all other requests formulated as a three-tuple. The client is notified of the error and the function is reloaded from disk.

The next clause handles all requests for statistics about the cache usage. The caller is informed of the `ets` table name, the size of the cache manager process and the number of currently active datum processes. A corresponding wildcard clause for erroneous two-tuple requests follows the `stats` request.

The last clause handles malformed requests. When a malformed request is received, the caller is not known and can not therefore be notified. The function is reloaded for this error case just as in the others.

3.3 Analysis

The code defined above is sufficient to have a minimal caching capability. Provided that the frequency of unique requests within one TTL time period does not exceed the available resources, the system should work with improved performance without any further code effort. As individual datum processes become idle too long, they will automatically die, triggering notification of the cache manager to remove the corresponding location reference. The memory resources consumed by the cache will grow and shrink in response to the dynamic behavior of client requests even though no replacement policy in the traditional sense has been provided.

Comparing this to a typical implementation, one finds many of the same elements. There is a hashed cache index and a cache manager that serializes access to the index. Elements of the index identify the location of memory data structures, a copy of which is returned when the corresponding index id is requested. This implementation explicitly disallows caching two data objects on the same key, although an alternate implementation could easily allow it.

```
mgr_loop(DatumIndex, DataModule, DataAccessor) ->

  receive

    %% A monitored datum process just went down...
    {'DOWN', _Ref, process, Pid, _Reason} ->
      ets:match_delete(DatumIndex, {'_', Pid}),
      mgr_loop(DatumIndex, DataModule, DataAccessor);

    %% Return a process id from the cache index...
    {locate, From, DatumId} when is_pid(From) ->
      case find_or_launch_datum(DatumId, DatumIndex,
                   DataModule, DataAccessor) of
        {found, Pid} ->
          From ! {locate, self(), DatumId, Pid};
        {launched, Pid} ->
          From ! {locate, self(), DatumId, Pid};
        Failed ->
          From ! {locate, self(), DatumId, failed, Failed}
      end,
      mgr_loop(DatumIndex, DataModule, DataAccessor);

    %% Invalid requests signal client and reload code...
    {InvalidRequest, From, DatumId} when is_pid(From) ->
      From ! {InvalidRequest,self(),DatumId,invalid_request},
      cache:mgr_loop(DatumIndex, DataModule, DataAccessor);

    %% Diagnostic queries of the cache manager...
    {stats, From} ->
      EtsInfo = tuple_to_list(ets:info(DatumIndex)),
      CacheName = proplists:get_value(name, EtsInfo),
      DatumCount = proplists:get_value(size, EtsInfo),
      From ! {stats, [{cache_name, CacheName},
             {cache_size,
               element(2, process_info(self(), memory))},
             {datum_count, DatumCount}]},
      mgr_loop(DatumIndex, DataModule, DataAccessor);

    %% Invalid requests signal client and reload code...
    {InvalidRequest, From} when is_pid(From) ->
      From ! {InvalidRequest, self(), invalid_request},
      cache:mgr_loop(DatumIndex, DataModule, DataAccessor);

    %% Other requests cannot be handled, but reload code.
    MalformedRequest ->
      cache:mgr_loop(DatumIndex, DataModule, DataAccessor)
  end.
```

Listing 2. Receive loop of the cache manager process.

3.3.1 Code simplicity

On inspection, half of the code presented thus far is for diagnostic purposes and error handling. Including access functions and wrappers for the message sending and receiving, plus error handling of the detail presented here, the bare bones implementation has about 200 lines of code. In a small, simple package one has a fully dynamic cache that can be used with any arbitrary data store, merely by providing a data access module with functions for obtaining and replacing a datum by unique key. The cache integrates naturally with existing `erlang` applications, even if the data store is not `erlang` based.

3.3.2 Distributed scalability

The code as presented does not differentiate between processes on the same `erlang` node and processes on other nodes, because a pid may refer to a local or remote process. Multiple nodes translate to multiple memory banks and multiple CPUs, so the approach can scale the size of the data cache larger than the entire memory of a single node and it can retrieve multiple datum values in parallel.

3.3.3 Multicore parallelism

Symmetric Multi-Processing (SMP) is the latest feature chip manufacturers use to increase the speed of their designs. The advent of SMP chips means that inherently parallel actions, such as retrieving multiple non-overlapping datum values, can be executed simultaneously on a single node. While the amount of memory available to the cache is not necessarily increased, the performance of the cache may increase, depending on the inherent parallelism of the data access pattern, merely by moving to a SMP platform and specifying multiple schedulers to run on the same node. No code change is necessary to gain this advantage.

3.3.4 Behavior independence

The completely orthogonal separation of cache management, data representation and replacement policy provides the power to experiment with performance knowing that the base architecture still behaves correctly. Thus far, we have only seen the cache management code and the fact that the integrator provides the data representation module. Replacement policy implementation will be discussed in the next section. What we can see is that the cache manager monitors and directs access to the active cache datum elements. This code can be tested, verified, stressed and modified without knowing the data representation model or the replacement policy.

Likewise, the performance of the data store and its elements can be simulated to observe the real performance characteristics of the cache. In a scenario where the data is not yet available, or is too complex to provide in entirety to the development environment, modules can be built to emulate the data size and handling delays. Empty buffers, forced idle time and large memory allocations can all be implemented in the data access module without using actual data values. The performance on realistically sized processing nodes with realistically sized data can be measured and tuned prior to building the hard parts of the application. Real benchmark numbers can be identified before deciding if the project can meet the performance requirements.

In the code presented earlier, an `ets` table was used to implement the cache manager to make the example concrete. The `mgr_loop` should be extended to take an *IndexModule* as its first parameter and the calls to the `ets` module should be replaced with appropriate calls to the functions `remove_datum`, `find_datum`, `add_datum` and `gen_stats`. This would separate the cache index implementation from the control flow of the cache manager loop, providing another component of the scaffolding that could be tuned independently.

Finally, the isolation of replacement policy means that various approaches can be tested with real data or with realistic simulations. Also, dynamically adaptable policies can be used so that the size of particular data elements may determine the policy which applies, or the complexity of construction or destruction can vary the replacement policy. If nodes of disparate performance are linked, the policies may take this into account and cause the migration of datum processes to those nodes on which they are best served, even caching some objects to multiple nodes provided the cache index is extended to recommend the best node given the location of the requester.

4. Replacement Policy Implementation

Extensive research has gone into the discovery of optimal cache replacement algorithms. Much of it has been driven by the use of caches for memory pages in Operating System (OS) implementations, which have different access characteristics than the canonical problem here yet share many of the same principles. In the code presented above, no replacement scheme was used, merely an expiration of idle objects.

As the datum elements are distributed, it seems natural that the replacement policy can be freed from its traditionally centralized location and distributed as well. First a single process approach is described and then a more sophisticated distributed approach. In either case, the role of the replacement policy is to recognize when it is needed, identify candidate processes to eliminate and terminate the one(s) determined to be least necessary.

4.1 Centralized Approach

Traditionally, the index manager recognizes that the index is full and selects an item to be replaced. The implementation used here is consistent with a single assignment language, so replacement is effected by eliminating the stale value's process and creating a new process to take its place. Since the code already creates a new process, elimination of the stale process is the only thing to be added. This can be accomplished simply by sending a message to the datum process replacing its TTL with zero. The datum will expire as soon as its message queue is empty.

To directly simulate traditional behavior, the cache manager would synchronously send a message to expire an existing process before spawning the new one. Listing 3 shows the old implementation of `launch_datum/4` which relied on datum self-expiration to free memory. It assumed that the memory size and access characteristics are such that the number of cache processes would not grow without bounds and could fit in the available resources, albeit possibly causing memory swapping on that node. It is enhanced by the wrapper `launch_datum/5` with an extra boolean argument to indicate when the cache is full. Using two separate functions allows the cache manager to choose to implement a replacement policy or not.

```
launch_datum(DatumId, DatumIndex, DataModule,
             DataAccessor, false) ->
  launch_datum(DatumId,DatumIndex,DataModule,DataAccessor);

launch_datum(DatumId, DatumIndex, DataModule,
             DataAccessor, true) ->
  reap_datum(DatumIndex),
  launch_datum(DatumId,DatumIndex,DataModule,DataAccessor).

launch_datum(DatumId, DatumIndex,DataModule,DataAccessor) ->
  DataToCache = DataModule:DataAccessor(DatumId),
  Pid = datum_start(DatumId, DataToCache),
  erlang:monitor(process, Pid),
  ets:insert(DatumIndex, {DatumId, Pid}),
  Pid.
```

Listing 3. Functions to launch a new datum process.

A simple method for managing the cache size is to set a maximum number of allowed processes, which is checked in the `mgr_loop` and to call `launch_datum/5` to signal whether the cache is currently full. The function `reap_datum` as shown in Listing 4 asks all processes for their current age and selects the oldest for elimination, only when the cache is determined to have too many processes. A time limit is placed on the responses by each cached process since the cache manager is blocked during this request.

The `receive` loop of Listing 1 for a cache datum process needs to be enhanced to hold the time of last activity in the datum record. It also requires that the TTL parameter is not reset by asking the process for its age, a refinement contained in the download source code but not in these code examples. These are minor changes which would be desirable for most any replacement policy.

```
reap_datum(DatumIndex) ->
  AllProcs = ets:tab2list(DatumIndex),
  Msg = {last_active, self()},
  Pids = [Pid || {_DatumId,Pid} <- AllProcs,(Pid ! Msg)==Msg],
  NumPids = length(Pids),
  reap_oldest(NumPids, 200, none).

reap_oldest(0, _Timeout, none) ->  ok;
reap_oldest(0, _Timeout, {OldestPid, _WhenActive}) ->
  OldestPid ! {new_ttl, self(), 0},
  ok;
reap_oldest(NumPids, Timeout, OldestPidPair) ->
  receive
    {last_active, Pid, WhenLastActive} ->
      NewOldest = choose_oldest(OldestPidPair,
                                {Pid, WhenLastActive}),
      reap_oldest(NumPids-1, Timeout, NewOldest)
  after
    Timeout -> reap_oldest(0, Timeout, OldestPidPair)
  end.

choose_oldest(none, PidPair) -> PidPair;
choose_oldest({_Pid1, When1}=Pair1, {_Pid2, When2}=Pair2) ->
  case When1 < When2 of
    true -> Pair1;
    false -> Pair2
  end.
```

Listing 4. Eliminating the oldest datum process.

This approach avoids implementing a data structure to manage the dynamic age of processes. It instead relies on each process to manage its own information and then uses collective messaging to identify the candidate for elimination. The code described so far operates equivalently to a Least Recently Used (LRU) replacement policy implemented in the traditional manner. As written, any other property of the datum processes could be used to determine the candidate for elimination by sending a different message type and calling an alternative function to `choose_oldest`.

The main drawback of this approach is the overhead involved in determining the oldest datum. In traditional systems, a data structure maintains an ordered collection of objects which must be updated with every access. In this approach, a simpler update is used on every access since only one value is maintained inside each datum process, but finding the oldest at any given time requires message transmission. The tradeoff is frequency of access versus frequency of cache replacement events. The reduced overhead here should win out in non-pathological situations. Recognizing the overhead, however, might drive consideration of an alternative to LRU that did not require round trip messaging. For example, tell all datum processes to expire if the process has been idle for 80% of the TTL. No acknowledgement or collection of replies is necessary and from 0 to N processes will be removed – an advantage in situations with new data accessed in bursts but a disadvantage if only an occasional single new datum is accessed or where previously referenced values are frequently re-referenced.

The benefit of having a constellation of datum processes is that alternate methods for determining elimination can be used:

- The datum process could dynamically manage TTL.
- The cache manager could eliminate processes proactively based on usage patterns.
- The replacement logic could act independently from the cache manager.

These three alternatives for determining the processes to replace, respectively correspond to local heuristics, predictive access pattern heuristics and partitioned or global optimization heuristics. Depending on the problem space and data access methods, some or all of these three techniques may prove useful for a particular situation. In any event, the code here encourages using a separate module for data access, cache index internals and replacement policy implementation. This behaviour-based approach gives more flexibility and power to the implementor.

4.2 Distributed Approach

The design space for the problem increases dramatically if multiple nodes or multiple processes are used to implement the replacement policy. The new considerations range from cache manager independence to network and node topology.

A supervisor process per compute node would be a natural extension of the architecture in a distributed network. Instead of a central process asking all datum processes for their age, the question would be asked by each supervisor. Each node would then return only a single answer and the controlling process would decide which datum should expire. If the nodes are partitioned on data type or other characteristics, it is possible only a subset of the supervisors need to take action.

A more sophisticated network of policies would monitor the access patterns on each node. Nodes that are seeing light traffic could advertise to take over work for other nodes. Datum processes could migrate from heavily loaded to lightly loaded nodes. The policy manager on each node would track load, frequency of access, rate of new datum process creation and expiration and other factors that could characterize the data access pattern. Each manager could decide whether to respond to resource requests by refusing without consulting the datum processes, or by broadcasting requests to a subset or all datum processes on the managed node and selecting from the responses.

Ultimately, the goal is to have a cache that is spread across the network like an amoeba with thick fingers on high capacity nodes and thinner fingers on low capacity nodes. The software should adapt dynamically to the actual ability of the nodes and network as currently configured and connected.

5. Eliminating Bottlenecks

The cache manager represents a nexus of requests and is likely the slowest aspect of the approach. The first step to improving it would be to abstract the index mechanism to an external module as described earlier. The implementor would then have control over the data structure and could optimize its performance based on the observed key frequencies and access patterns.

A second step is to make the cache manager distributed rather than a single central process. The benefit is that a single point of entry for clients is replicated to provide more simultaneous access channels to the cache, and the possibility of a a single point failure is eliminated. The drawback is that the index would have to be replicated in real time across multiple nodes, the synchronization of which would lead to a constant overhead factor. In a heavily used application, however, parallel access could easily overcome the replication costs especially if on balance the access profile favors reads over writes. The enhancement could be introduced naturally by switching to memory-based `mnesia` tables for the cache index. Replication is a proven feature of `mnesia`, even in the face of network partitioning.

A deeper problem is the speed of the data store. This article has mainly avoided modifying the data store or using write-thru caches. The naive approach is to let the cache manager replace the value in the datum process only after committing a change to the data store. Using a single process cache manager results in access delays for other values while waiting for the data update to occur. It is more efficient to spawn a new process to update the data store and signal the `new_data` call before terminating so that the cache will refer to the latest value. Doing so, requires marking the current datum as dirty and refusing requests until it is clean.

6. Further Research

The implementation described here is intended to be a complete but simple system to demonstrate the concepts. Further work is necessary to optimize the code in significant ways that would increase the performance on large operational systems. The key benefit is that it should be possible to introduce caching to existing systems with minimal code changes, either by introducing the code onto the existing hardware or by connecting additional hardware to the local network.

Adaptive process migration is a new area of research suggested by this implementation. It has the potential to be as important as a replacement policy in affecting the performance of a concurrent caching system. Specialized hardware, competing applications, asymmetric transmission bandwidth or variable data element sizes all could create non-uniform advantages to particular nodes in a caching cluster. By allowing the cache to dynamically detect and adapt to the topology performance, the system can tune itself to optimal performance on a continual basis. If even partially effective, the approach could only improve over a static configuration of resources. Work on this area would benefit from research in the areas of self-organizing systems and emergent behavior.

Another area ripe for consideration involves the simulation and benchmarking of cache performance. Ideally, a set of data challenges similar to the Transaction Processing Performance Council's on-line transaction processing (OLTP) benchmark [5] would be helpful in measuring and comparing the efficiency of different approaches to caching for a given data store's characteristic behavior. A standard library distributed with the framework could model typical problems in data store access, such as slow access, complicated initial object construction, variable data object size, asymmetric node performance or other access issues.

The analysis presented here does not do justice to the complexity of the messaging system. While *selective receive* is very handy in this problem domain, there are difficult complications in any system with multiple communicating processes. The main source of problems occurs when particular message patterns build up in the message queue, slowing down all message handling progress or when the order messages are sent differs from the order of receipt desired.

Deeper analysis through simulation is necessary to harden the message response methods. For example, both clients and datum processes send messages to the cache manager, but in the code above the cache manager uses timeouts to prevent slow responses to age requests of datum processes from blocking the manager itself. As implemented, the timeout allows slow messages to collect in the message queue over time, slowing all selective receives. A better approach is to spawn a new process to collect results and have it terminate when the data is no longer needed, thereby eliminating the queue of messages. Other more subtle issues may still remain. A concentrated effort to improve message handling in this code implementation is warranted.

7. Conclusion

Leveraging the principal benefit of COPLs, it is possible to define an approach to caching data using multiple compute nodes and SMP cores by organizing the cache around a federation of processes. A small amount of code can deliver a bare bones implementation that may be integrated with arbitrary data stores and network topologies. The methods for dynamically managing memory consumption of the cache and optimizing response times are also open and flexible, supporting performance evaluation with or without the production data store.

The potential of COPLs can only be realized if the historical solutions in Computer Science are revisited with an eye towards design using processes. A re-examination of the field's most fundamental assumptions will soon be essential if software is to take advantage of the multi-core architectures intended to speed computation.

Acknowledgments

Special thanks to Richard Cameron for querying the `erlang` email list to obtain ideas on how to implement a LRU queuing scheme.

References

[1] A. Aho, P. Denning, J. Ullman. *Principles of Optimal Page Replacement*. J. ACM, vol. 18, pp 80-93, 1971.

[2] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. Ph.D. Thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[3] E. Dijkstra. *EWD 447: On the role of scientific thought*, in *Selected Writings on Computing: A Personal Perspective*. ISBN 0-387-90652-5, Springer-Verlag, 1982. Available at http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[4] DuoMark International, Inc. *ICFP 2006: Concurrent Caching* Code modules. Internet download. Available at http://www.duomark.com/erlang/.

[5] Transaction Processing Performance Council. Internet Website. Available at http://www.tpc.org/tpcc/.