SACache

SACache is an N-way Set-Associative cache. It is designed to provide a hybrid caching system that balances the high lookup performance of direct-mapped caches with the flexibility and higher hit ratios of fully associative caches.

General Theory of Operation

Direct-Mapped Caches

In a direct-mapped cache, cache "lines" (aka "slots") are directly mapped to memory blocks. The memory address for each value is hashed to map it to a cache line, and a simple lookup is performed to see if the desired value is in the cache. Although hardware caches typically do this by bit-masking physical memory addresses into block-number / block-offset pairs, high-level-language software caches can use basically the same technique by generating integer hash codes for objects, then Mod()'ing those values against the cache size.

Lookup performance is high for these caches because there is no iteration: both storage and lookup operations are O(1). However, hit/miss ratios are poor because there is only one cache line per physical memory block (or hash code range). Adjacent objects cannot be cached simultaneously, so iterating across collections of objects (a very common use-case) can actually produce worst-case performance, with each new retrieval evicting the previous value.

Associative Caches

Associative caches provide better hit/miss-ratios by allowing an object to be stored anywhere in the cache. An associative lookup is later used to retrieve it. This means large collections of related objects may be stored together, with collection sizes up to the cache size itself.

Unfortunately, associative caches have extremely poor lookup performance because they must evaluate so many stored values to find a match. On average, this is O(CacheSize/2), which is not terrible for small caches but makes increasing the cache size not an effective strategy for increasing performance. Storage performance can be as bad as O(CacheSize) once the cache is full for a naïve LRU eviction strategy (although this can be improved).

It is worth noting that associative caches have the same basic behaviors as the Dictionary/HashMap collection types provided by nearly all high-level languages. They are thus rarely worth implementing on their own.

N-way Set-Associative Caches

N-way S/A caches are a hybrid between the two approaches. They retain (most of) the calculated-location lookup performance of direct-mapped caches, but allocate more than one cache line per physical memory block. The number of lines allocated per block ("N") becomes a tuning factor for the cache - N objects may be cached for each physical memory block, allowing

much better performance when handling a series of nearby objects.

Within each cache set, the N objects that are stored are handled associatively, allowing mixes of different physical object access patterns (random, contiguous series, etc.) to all attain good lookup performance. N is typically small (values from 2 to 16 are common), and this upper bound minimizes the negative effects of fully associative caches.

This style of cache is tuned by adjusting the value of N. Lookup performance is O(N), which is a bit slower than a direct-mapped cache but nowhere near a fully associative cache. Higher values of N reduce lookup performance but also provide higher hit ratios. Programmers can thus tune this type of cache to maximize the performance for each use-case.

Usage

Using SACache in a project is easy. First add a Reference to it in your project. Then create a cache with appropriate options and retain a reference to it for use in your code:

```
SACache<int, string> cache = new SACache<int, string>();
cache.put(0, "Test");
Assert.AreEqual(cache.get(0), "Test");
```

SACache allows both keys and values to be strongly typed, and of any type that inherits from System.Object:

```
class MyKeyType { int id; }
class MyValueType { string value; }
SACache<MyKeyType, MyValueType> cache = new SACache<MyKeyType, MyValueType>();
```

When creating a cache with default options, the cache will default to a size of 32M objects configured for a set size of N=4. These values may be overridden to tune the cache for a specific application:

```
// Create a 1024-object cache with N=8. This might be good for small collections of
// frequently-iterated items.
SACache<int, string> cache = new SACache<int, string>(1024, 8);
```

SACache by default uses LRU as its eviction technique when a cache set is full, and `Object.GetHashCode()` to generate hash codes. These may be adequate for many use-cases but may be overridden when the programmer has a better strategy:

```
class UselessEvictor : IEvictor<KeyType, ValType> {
    public int evict(CacheEntry<KeyType, ValType>[] cacheEntries, int startIndex,
int endIndex) {
        return startIndex; // Always evict the first entry in the range
    }
```

```
}

class EmployeeSSNHasher : IHashGenerator<Employee> {
    public int getHashCode(Employee obj) {
        return employee.ssn; // Let's say we're storing the ssn as an int already
    }
}

SACache<Employee, Employee> cache = new SACache<Employee, Employee>(4, 2, new
UselessEvictor(), new EmployeeSSNHasher());
```

If null is specified for either behavior, SACache will use LRU for the eviction strategy and `Object.GetHashCode()` for the hash generator. Note that this example also illustrates using the same type for both key and value. This is a convenience pattern that allows you to use shorthand when storing and retrieving complex object types in the cache, while allowing you to later change how the key's hash code is generated as the application evolves.

For your convenience, SACache ships with both LRU and MRU eviction techniques as options, which should cover nearly all but the most specialized use-cases.

Tuning the cache

Three metrics are provided to help you tune the cache parameters:

- `cache.cacheHits` The number of objects found during GET operations (HITS)
- `cache.cacheMisses` The number of objects NOT found during GET operations (MISSES)
- `cache.cacheEvictions` The number of objects evicted during PUT operations

It is important to use these values together, rather than individually. For example, a high rate of evictions is not necessarily bad if an application tends to re-use objects only once or twice. In general, a high hit/miss ratio should be the primary goal for most caches.

Cache tuning may require extensive benchmarking to identify ideal values, and what is provided here is only general guidance. Typically, increasing cache size is a good first step if memory is available, but at some point there will be diminishing returns.

After cache size the set cardinality (N) should be considered, but this should be done at the same time as reviewing the hashing and eviction techniques. Ideal hashing routines should provide a good distribution of stored values across the cache. An early warning sign of a bad hashing technique is a high miss ratio and eviction count that are not improved by increasing the size of the cache.

Ideal eviction techniques should be based on the application's object access pattern. The standard LRU approach should be good for most applications. MRU may be better in cases where data sets greatly exceed possible cache sizes or object accesses are mostly random.

Additional tuning guidance may be found in UCSB's detailed presentation.

TODO

- Allow a backing-store connection to be specified via a lookup/retrieval Interface. This would streamline application code in many cases by eliminating `if (obj == null)` branches after each retrieval.
- Add more metrics, such as cache usage and set usage distribution, to help developers tune their applications.
- Add utility functions like `clear()`, `resize()`, and preload/warming helpers.

Contributing

Please follow all C# coding standards as outlined in Microsoft's C# Coding Standards guide.

The SACache Solution includes a test suite called SACacheTest, and a code-coverage tool that can be manually executed via testcover.bat (the solution does not run it every post-build for performance reasons). All contributions should include sufficient tests to maintain or increase the current code-coverage ratio (87%).

Ideally, the test case for any new features should be descriptive enough to illustrate appropriate usage for the feature, and/or the pull-request should include an update to this README to document the feature.