Designing Configurable, Modifiable And Reusable Components For Simulation of Multicore Systems

Jun Wang*, Jesse Beu†, Sudhakar Yalamanchili* and Tom Conte†
*School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, Georgia 30332–0250
Email: {jun.wang, sudha}@ece.gatech.edu
†School of Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332–0765
Email: Jesse.Beu@gmail.com, Tom@conte.us

***Abstract*—A simulation system for modern multicore architectures is composed of various component models. For such a system to be useful for research purposes, modifiability is a key quality attribute. Users, when building a simulation model, need to have the capability to adjust various aspects of a component, or even replace a component with another of the same type. Software design considerations can determine whether or not a simulation system is successful in providing such capabilities. This paper presents a few design tactics that we adopt in creating configurable, modifiable, and reusable components for Manifold, our parallel simulation framework for multicore systems. The main example component is MCP-cache, a coherence cache model. The ideas behind the tactics are general enough and should be useful to designers of similar systems.**

I. Introduction

Simulation is an important tool in computer architecture research. For a simulation system to be useful for research purposes, it must be designed in a way that allows the components to be configurable and modifiable. This is because the very nature of research means researchers, in their quest to gain insights from simulations, need constantly experiment with system configurations: changing the value of a parameter, replacing one algorithm with another, mixing-and-matching components in the system model, and so on. A well designed system would allow users to make such configuration changes with relative ease, and may even allow users to modify configuration files only without re-compiling. A rigid design, on the other hand, is usually characterized by hard-coded features and tightly coupled components, making modification or extension impossible without changing the source code. This paper presents a few design tactics that have been proven successful for the Manifold project, at both the software architecture level and design level. Layered software architecture separates a system into layers, thus allowing changes to be localized and promoting reuse. Standardized interfaces make it possible for components of the same type to be interchangeable. Separation of concerns is an important design principle that divides the software system's functionalities into different modules. This again allows changes to be localized and facilitates reuse. Dependency injection decouples a component and its dependency such that the component only uses the dependency objects' services without creating the objects. Changes made to the dependency does not require its user to change at all, as long as the interface stays unchanged. With these software design tactics we have made Manifold a highly extensible, modifiable, and configurable system that allows users to easily adjust various aspects of the system for their research purposes. These tactics do not rely on any Manifold-specific features and should be useful to designers of similar systems.

II. Background

A. Manifold

Manifold [6] is a scalable parallel discrete event simulation (PDES) framework for research of modern multicore computer architectures. From a module point of view, Manifold is composed of a parallel simulation kernel and a set of computer architecture models. Users use these components to build system models for parallel simulations. Based on the needs of the user community, there are three important goals that Manifold is designed to achieve: (1) PDES services should be transparent to users. (2) the system must be open and allow the research community to make contributions either by porting exiting components or writing new components. (3) users should be able to mix-and-match components from the component repository to build their system models.

To achieve these goals, Manifold has adopted a layered software architecture that separates the parallel simulation kernel from computer architecture models (components), and, as a general rule, components are independent of each other. This is depicted in the software architecture view in Figure 1.

B. MCP-cache

This section gives an overview of Manager-Client Pairing (MCP) [2], and MCP-cache, a coherent cache model that implements MCP. MCP-cache is our main example component in presenting design tactics for creating modifiable and reusable components.

1) MCP: MCP is a methodology for creating hierarchical cache coherence in shared-memory multicore architectures. Its main purpose is to address the scalability issue of cache coherence.

MCP maintains cache coherence through a hierarchy of coherence realms forming a tree structure, as shown in Figure 2.

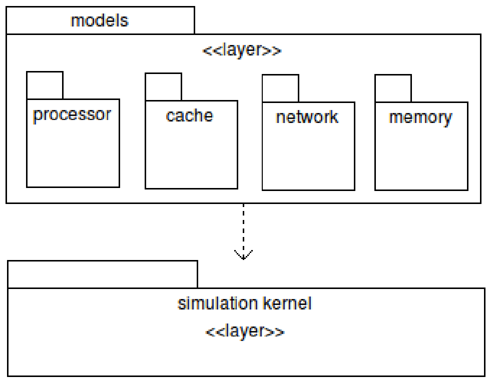

Fig. 1. Manifold software architecture.

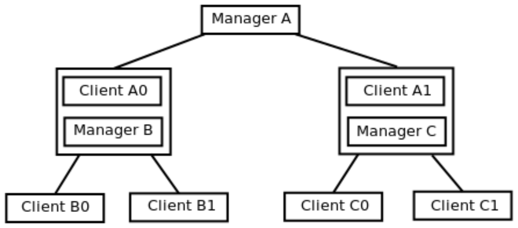

Fig. 2. MCP Coherence Hierarchy.

Within each realm there is a manager-client pair at the root of the sub-tree: the manager manages coherence within the realm, while the client acts as a representative of its realm in the parent realm. Using MCP each realm can achieve coherence using a different coherence protocol. This is made possible by defining two generic interfaces, one for clients and one for managers, as shown in Figure 3.

The client interface defines operations for the client to check for permissions such as read permission. If it does not have the permission, it sends a request to its manager. A manager, on the other hand, when receiving a request from a lower realm, checks with its pairing client to determine whether or not the request should propagate upward. Eventually, a response is sent to the lower realm client to complete its request.

2) MCP-cache: MCP-cache implements a cache system that consists of two levels of caches. The L1 cache is where coherence is enforced and the L2 cache serves as the directory. Taking the cache-centric approach [7], the state of a cache line in L2 is the aggregation of its states in all the L1s. The L1 and L2 caches both have a hash table. To implement MCP, the L1 cache has one client associated with each hash entry, and the L2 cache has one manager for each hash entry.

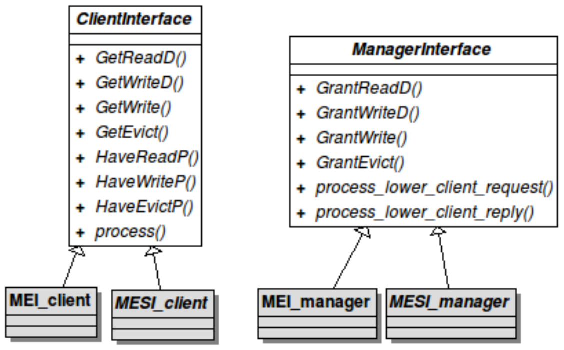

Fig. 3. Client and Manager Classes.

III. Design Tactics for Components

In this section we present a few design tactics that we adopted for Manifold to create modifiable and reusable components. We use MCP-cache as the main example.

A. Layered Architecture

Following a common practice [1], we start by identifying the key quality attributes of MCP-cache and then select a software architecture that can best achieve the qualities.

The most important design goal of MCP-cache is to provide the user the ability to experiment with different coherence protocols. This is stemmed from the basic purpose of MCP, which, as mentioned above, is to allow coherence hierarchy to be easily created where sub-realms of the hierarchy can use different protocols. Therefore, the most important quality attribute of MCP-cache is modifiability. The system should be designed in such a way that state machines for different coherence protocol are exchangeable, and, adding new state machines only involves localized changes.

Based on this, we adopted the layered architectural style [4], also known as the layers architectural pattern [3]. The most important characteristics of this architectural style, as pointed out in [3], include promoting modifiability, portability and reuse, and achieving separation of concerns.

We divide MCP-cache into two layers, a protocol-independent layer and a protocol layer, as shown in Figure 4. The protocol-independent layer implements basic cache functions and does not have or require any knowledge of the particular coherence protocol being used. The protocol layer implements coherence protocols such as MESI in the form of state machines. Obviously, for the protocol state machines to be interchangeable, they must share a common interface, through which the protocol-independent layer accesses the state machines to carry out the cache coherence operations.

This common interface, incidentally, is the generic interfaces that MCP defines for clients and managers, as shown in Figure 3. The two base classes `ClientInterface` and `ManagerInterface` define a set of operations that is considered common to all broadcast and directory-based coherence protocols. Figure 3 also shows concrete subclasses for coherence protocols such as MESI. The client and manager in general are implemented as state machines. The internals of the state machines such as the states and transitions are obviously determined by the coherence protocol and implemented in the subclasses. For example, the `MESI_client`

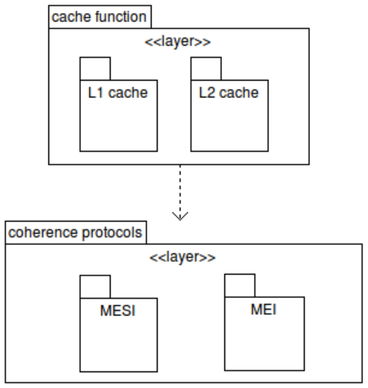

Fig. 4. Layered architecture for MCP-cache.

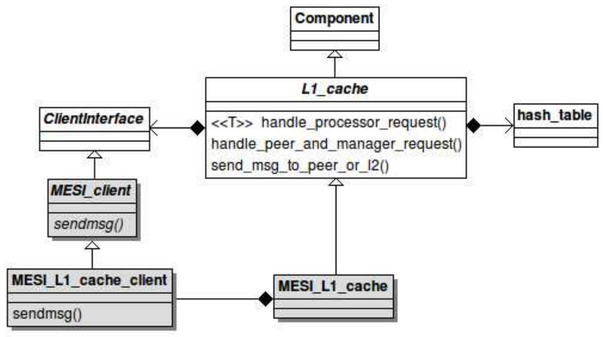

Fig. 5. L1 cache.

and `MESI_manager` classes together implement the MESI protocol.

The protocol-independent layer contains the L1 and L2 caches. Class diagram for L1 is shown in Figure 5, with the classes in the protocol layer in darker boxes. L2 cache is very similar.

From Figure 5 we can see, when adding a new coherence protocol, on the L1 side, it only requires creating a subclass of `L1_cache` and a subclass of `ClientInterface`. The major purpose of the former is to instantiate objects of the latter, while the latter implements the coherence protocol state machine. For example, to implement the MEI protocol for MCP-cache, all we need to do is to create, say, `MEI_L1_cache` as a subclass of `L1_cache`, an abstract subclass of `ClientInterface` called `MEI_client` and its subclass `MEI_L1_cache_client`. The reason why we need two subclasses of `ClientInterface` is because messaging is separated from the state machine, following the principle of separation of concerns, as discussed further in Section III-C. The L2 side is very similar.

With this design, the user would create an `MESI_L1_cache` object for L1 if MESI is the selected protocol, as shown in Figure 6(a). When the user wants to replace MESI with MEI, all that is required is to replace the line in Figure 6(a) with the line in Figure 6(b).

```
L1_id = Component::Create<MESI_L1_cache>(...);
```
(a)

```
L1_id = Component::Create<MEI_L1_cache>(...);
```
(b)

Fig. 6. Using different coherence protocols.

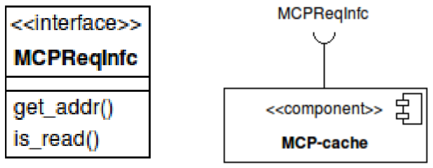

Fig. 7. MCP-cache required interface.

We believe this layered architecture can well achieve the major quality goal of the system, which is to allow new coherence protocols to be easily integrated with the existing infrastructure. From Figures 1 and 4 we can see the layered architecture supports clean separation of concerns, helps localize changes, and promotes reuse. It not only is applicable at the overall system level, but at the component level as well.

B. Standardized Interfaces

Figure 1 shows that the components in the models layer of Manifold are independent of each other. One of the most important advantages of this independence is that it provides users the ability to mix-and-match components when building system models. Take the processor-cache interface as an example. If a cache model C has a dependency on a processor model $P1$, then it's hard to build a system model with C and another processor model $P2$ without making changes to the source code, which is not always possible. On the other hand, if C is independent of either $P1$ or $P2$, it can be easily assembled with either $P1$ or $P2$.

In Manifold, it is the standardized interfaces that make the model independence possible. For example, to make MCP-cache work with more than one processor model, we define a required interface [4] for MCP-cache, as shown in Figure 7. The interface defines two functions: `get_addr()` returns the address of the cache request, and `is_read()` returns true if the request is a load request. Any data type can be sent to MCP-cache from the processor model as long as the two functions in the required interface are defined. With the standardized interface, MCP-cache can work with any compliant processor models, making the processor models interchangeable.

C. Separation of Concerns

Separation of concerns is "the process of separating a computer program into distinct features that overlap in functionality as little as possible" [8]. The layered architecture, as already mentioned, promotes separation of concerns. Here we describe two more design choices that follow the principle of separation of concerns to promote reuse.

The first is the separation of the hash table from the caches. The hash table manages address tags. Its responsibilities include storing address tags, allocating new entries, replacing entry when a set is full, and so on. By putting all these functionalities in a separate class `hash_table`, as shown in Figures 5, not only do we free `L1_cache` and `L2_cache` from having to manage the hash table, but we can also reuse the class and save development and testing time. In fact, it is used by both `L1_cache` and `L2_cache`.

Another case of separation of concerns is messaging. If the clients and managers of MCP-cache are tightly coupled with the underlying messaging system, then reuse is more difficult. We have chosen to completely separate messaging from the clients and managers by making the messaging related function abstract in the state machines. Taking the client as an example, in Figure 5, the `sendmsg` function in the `MESI_client` is made abstract. This allows `MESI_client` to focus on state transitions and leave out messaging. The main purpose of the subclass `MESI_L1_cache_client` is to implement messaging. Object of this class has a reference to an `MESI_L1_cache` object, which is a subclass of Manifold's `Component` class. Eventually, it is the messaging function of `Component` that is used by `MESI_L1_cache_client`. It can be easily seen that this separation of concerns makes the state machine class `MESI_client` completely independent of Manifold, thus facilitating reuse. If MCP-cache is to be ported to another system, the state machines need no modification because the system-dependent messaging is decoupled.

D. Dependency Injection

Dependency injection [5] is a software design pattern that decouples an object and its dependencies, and allows dependency to be bound at run-time rather than compile time. This pattern involves three elements: an object that uses another object (dependency), an interface that specifies the dependency, and an injector.

In MCP-cache, an L1 cache, when sending a request to an L2 cache, needs to know the node ID of the L2 cache, for the given memory address, as there are multiple L2 slices in a distributed shared cache. Similarly, when an L2 cache sends a request to a memory controller, it needs to map the memory address to the destination's node ID. Clearly, it is not desirable to hard-code the mapping from address to node ID. Therefore we created an abstract class called `DestMap`, whose abstract function `lookup` returns a node ID for a given address. A few concrete subclasses of `DestMap` were also created, as shown in Figure 8.

The next question is where to create the concrete mapping object. If, for example, `L1_cache` creates a `PageMap` object, then `L1_cache` is tightly bound to `PageMap`; it cannot use a different mapping object to do the address-to-node mapping. Using dependency injection, neither `L1_cache` nor `L2_cache` instantiates the mapping object. Instead, the objects are created externally and injected into them. Figure 9(a) shows a `PageMap` object is created and passed to the `Create` function, which in turn, passes the object

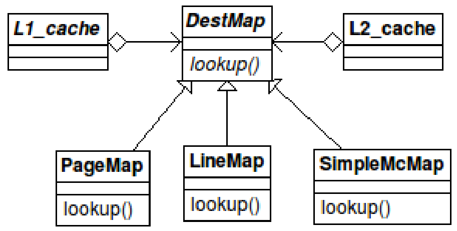

Fig. 8. L1_cache, L2_cache, and DestMap.

```
DestMap* mapping = new PageMap();
L1_id = Component::Create<MESI_L1_cache>(mapping);
```
(a)

```
DestMap* mapping = new LineMap();
L1_id = Component::Create<MESI_L1_cache>(mapping);
```
(b)

Fig. 9. Dependency injection when creating L1 cache.

to the L1 cache's constructor. If the user decides to use `LineMap`, only one line needs to be changed, as shown in Figure 9(b). Note that L1 cache does not need to know what concrete subclasses of `DestMap` exist. If a new subclass of `DestMap` is created and used in a system model, there is no change required for `L1_cache`. With dependency injection, `L1_cache` (`L2_cache` as well) only uses the service provided by the mapping object and does not concern itself with the instantiation of the object. This decouples the cache function and the mapping service and gives the client code of MCP-cache the freedom of selecting its own mapping objects.

In Manifold, we divide the concerns into two classes: those concerning individual components, and those concerning system models. Any dependency that should not be tightly coupled with a component is injected from the outside when the system model is being built. This, to a certain extent, makes the components independent and customizable, increases their reusability, and gives the users the possibility of customizing various aspects of the system.

IV. Conclusions and Future Work

Simulators for computer architectures must be configurable and modifiable in order to best serve research purposes. To produce such a system, software designers must make conscientious design choices that promote modifiability. In this paper we have presented a few design tactics, at both the software architecture level and design level, that have proven to help create flexible components and software systems. The tactics have been successfully used in the Manifold project, resulting in a highly modifiable simulation system where same-class components are interchangeable, different components are independent of each other, and features involving multiple components are injected.

In the future, we plan to move as much system configuration as possible to configuration files and greatly reduce the need to recompile, further improving Manifold's usability.

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[2] Jesse G. Beu, Michael C. Rosier, and Thomas M. Conte. Manager-client pairing: A framework for implementing coherence hierarchies. *The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, 2011.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. Wiley, 1996.

[4] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2nd edition, 2011.

[5] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

[6] manifold.gatech.edu. Manifold. http://manifold.gatech.edu.

[7] D.J. Sorin, M.D. Hill, and D.A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool Publishers, 2011.

[8] Wikipedia.org. Separation of concerns. http://en.wikipedia.org/wiki/Separation_of_concerns.