Externalizing Java Server Concurrency with CAL*

Charles Zhang[1] and Hans-Arno Jacobsen[2]

[1] Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
charlesz@cse.ust.hk
[2] Department of Electrical and Computer Engineering
and Department of Computer Science
University of Toronto
jacobsen@eecg.toronto.edu

Abstract. One of the most important decisions about the architecture of a server program is its concurrency mechanisms. However, a good concurrency model for general-purpose server programs is increasingly difficult to conceive as the runtime conditions are hard to predict. In this work, we advocate that the concurrency code is to be decoupled from server programs. To enable such separation, we propose and evaluate CAL, — the Concurrency Aspect Library. CAL provides uniform concurrency programming abstractions and mediates the intrinsic differences among concurrency models. Through CAL, a server program is not tied to any particular concurrency model and framework. CAL can be configured without modifications to use concurrency frameworks of fundamentally different natures. The concurrency code based on CAL is simpler and looks closer to the design. Leveraging the customizability of CAL, we show that a commercial middleware server, refactored to use CAL, outperforms its original version by as much as 10 times.

1 Introduction

A common definition of concurrency is the perceived simultaneous executions of multiple sets of program instructions within the same address space. Concurrency mechanisms, particularly in relation to I/O, are vital to the functionality of today's general-purpose server programs, such as databases, web servers, application servers, and middleware systems. Since the trend of multi-core architectures no longer focuses on the clock speed, server programs increasingly rely on concurrency for performance improvements. The current research on the design of concurrency models is characterized by the pattern-based concurrency designs [2, 18, 20, 21]. They primarily focus on achieving high, scalable and fair server throughput, assuming specific runtime conditions such as hardware concurrency capabilities and characteristics of incoming requests. As the nature of today's network-based applications continues to diversify, such concurrency

* In ECOOP 2008, Paphos, Cyprus

models will become increasingly hard, if not completely impossible, to design due to the difficulties in predicting the runtime conditions for general purpose server programs. Let us first exemplify this problem through a simple design exercise.

The goal of our design exercise is to allow a simple server program, presented in Figure 1(A), to provide a generic upload service to simultaneously connected clients. Despite its simplicity, the server performs some of the typical operations of Java server programs: binding to a server-side socket and waiting for incoming connections (Line 9), decoding the application frame from the incoming socket (Line 4), and processing thee received frame such as storing it in a database (Line 5). This server, as shown, can only serve one client for the duration of request processing.

```
1  class Server{
2      public boolean dispatch(Socket s){
3          InputStream in = s.getInputStream();
4          Frame buf = decodeData(in);
5          database.store(buf);
6      }
7
8      public void start(){
9          socket = serverSocket.accept();
10         dispatch(socket);
11     }
12 }
```

```
1  class Server{
2      ... ...
3      public void start(){
4          socket = serverSocket.accept();
5          new Thread(new Runnable(){
6              public void run(){
7                  dispatch(socket);
8              }
9          };
10     }
11 }
```

Fig. 1. (A) Upload server (B) Thread-per-connection

First solution. Our first attempt is to implement the "thread-per-connection" concurrency model (Figure 1(B)), common in tutorials, textbooks, and industrial practices. We evaluate our improved design through quantifying the server throughput measured as number of processed requests per unit time[3]. In Figure 3, we plot the number of frames received by the server within a fixed duration against the number of concurrent clients. We measure two types of client/server connections: *long*, i.e., the clients keep their connections alive for the entire duration (`upload(L)`); and *short*, i.e., the clients repeatedly connect to the server, send a piece of data, and disconnect (`upload(S)`). Figure 3(A) shows that our solution works well as the server throughput only degrades around 20% to 30% for both types of connections even for a high number of clients. We now introduce an evolutionary change to the example server by adding a new service: factorizing big integers, as illustrated in Figure 2(A). The measurements for this new service are plotted in Figure 3(A) with labels `factor(L)` and `factor(S)`. We immediately observe that, when the number of concurrent clients gets large (> 1000), the throughput of the constant connections (`factor(L)`) degrades as

[3] We also measure the fairness of the services. However, for motivation purposes here we omit the relevant discussions. We come back to the fairness issue in Section 4.

much as 90%, and the periodic connections also suffers from significant throughput oscillations. Seasoned concurrency programmers can quickly point out that the use of Java threads in our factor server does not scale to the large number of concurrent clients due to contention of the CPU resources between the thread-level context switches and the factorization work itself.

```
class Server{
  public boolean dispatch(Socket s){
    InputStream in = s.getInputStream();
    Frame buf = decodeData(in);
    switch(buf.jobtype){
      case CPUJOB:
          factor.doFactorization(buf);
          break;
      case IOJOB:
          database.store(buf);
          break;
      }
    return true;
  }
}
```

```
public void start(){
    final SocketChannel channel = ssc_.accept();
    channel.configureBlocking(false);
    /* notify me if data are ready on channel*/
    final SelectionKey key = channel.register(
          reactor.getSelector(), SelectionKey.OP_READ);
    class Handler implements IAsyncWorker{
          public void doAsyncWork() {
                dispatch(channel);
          }
    }
    /* reactor executes the handler if the READ
     * event is fired */
    reactor.registerWorker(key, new Handler());
}
```

Fig. 2. (A) Evolved server (B) Reactor-based event multiplexing

Modified solution. The availability of asynchronous I/O in the Java platform allows concurrency to be supported using the event multiplexing model, hence, avoiding the thread-level context switches. Figure 2(B) illustrates a modified implementation of the server using the *Reactor* [18] design pattern, in which each incoming connection is registered with a key (Line 5). The key is used by the reactor (Line 16) to invoke the corresponding handler when data from the network is ready to be processed. The stream-based sockets are also replaced by the channel-based counterparts. Figure 3(B) plots the measurements for the reactor-based server regarding both the factorization and the upload services. The factorization service scales very well for both connection types. However, the upload service suffers from around 63% degradation when admitting 5000 clients. This is because, when the dispatch table used by the reactor becomes large, frequent I/O event triggering and dispatching become costly for both OS and the VM when a large amount of network data arrive.

We now run the same server program on a dual-core CPU machine as quantified in Figure 4. For the number of connections lower than 2000, the event-dispatch model is once again significantly costly to use even for the CPU-bound requests that have low I/O dispatch overheads, due to the performance boost to the multi-threaded concurrency model by the multi-core CPU. Our modified solution, in spite of significant design and code-level alterations, is still not general enough for both types of connections.

Based on this example, we argue that, if designing a general concurrency model for our simplistic example server is not straightforward, it would be even more difficult to do so for servers of much more sophisticated semantics. The difficulty lies in the fact that concurrency designs are dependent upon the runtime

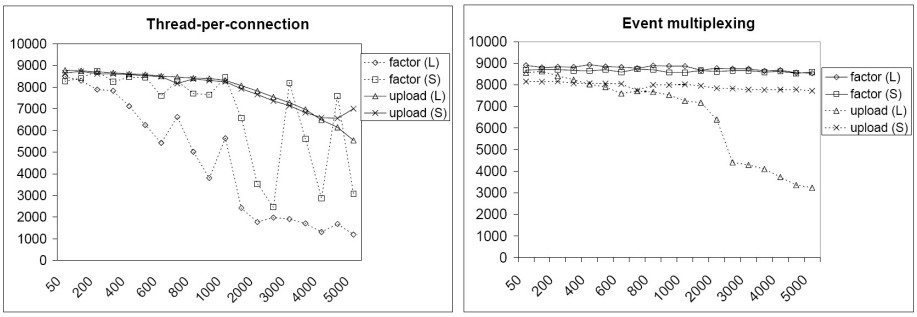

Fig. 3. (A) Thread-per-connection (B) Reactor-based event multiplexing. Data is collected on JRockit JVM version R27.3.1-jre1.6.0_01. Programs are hosted on two Intel Single CPU Xeon 2.5GHz machines with 512KB cache and 2GB physical memory connected by a 100MB switched network. Both machines run Fedora 2 with the 2.6.10 kernel. Uploads are in 80KB chunks. The integer factored is 22794993954179237141472. Each data point is measured three times for a duration of 60 seconds. The median value is chosen for the plot.

conditions of the server programs such as the load characteristics and the hardware capabilities. More specifically, these deployment and runtime conditions are subject to the following design uncertainties:

I: *Unforeseeable platform capabilities.* The computing resources of the hardware are not known until deployment time. Java programs are separated further from both the hardware and the operating system due to the virtual machine model. However, concurrency mechanisms are sensitive to hardware profiles such as the number of processors, the size of the physical memory, and execution privileges regarding the number of allowed open files or active OS-level threads.

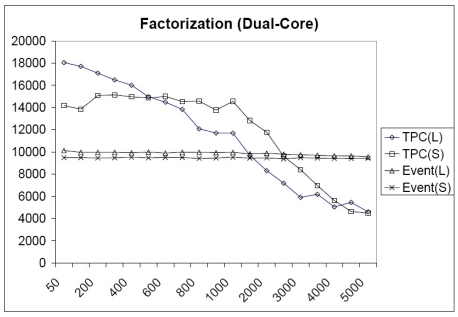

Fig. 4. Event-based factorization server on dual-core CPU

II: *Diversified load characteristics.* General-purpose server programs exert little control over how they are actually used, hence, they are often subject to diversified and yet specialized load characteristics. Browsing requests seen by Web servers are usually I/O-bound and short in duration, while providing services like YouTube needs to support longer I/O-bound requests of both outbound (viewing) and inbound (uploading) traffic. Client-server interactions in services such as online interactive games are typically CPU-bound and long in duration, while services for computing driv-

ing routes usually serve shorter CPU-bound requests.

III: *Unanticipated evolution.* Our server example demonstrates a case of unanticipated evolution of the server semantics. The recent push towards multi-core architectures, for example, makes it attractive for many legacy server architecture to parallelize their operations. Another kind of unanticipated evolution concerns with concurrency frameworks and libraries. The I/O and concurrency facilities of the Java platform is a very good example. Java servers on early JVMs only had synchronous I/O and the "thread" primitive at their disposal. More powerful concurrency and event-driven I/O support later came from third party libraries such as the `util.concurrent` library[4], the aio4j library[5], and the NBIO package[6]. The latest Java platforms introduce both more sophisticated concurrency primitives and the support for the asynchronous I/O. During this period of fast evolution, leveraging platform advances would require repeated and "deep" modifications to the server code even if the server semantics do not change.

IV: *Different correctness requirements.* It is important and often very difficult to maintain the safety and liveness [15] properties in complex concurrent systems. Most concurrency primitives, as those in JDK, rely on the experience of the programmers to guard against concurrency hazards such as race conditions, deadlocks, or livelocks. For some domains, the transient faults produced by concurrency bugs can be tolerated through sophisticated schemes such as replication [3]. But it is often important for the concurrency implementation to be provably correct. Verifiable concurrency models such as CSP [9] or the Actor model [1] have been proposed for decades. However, as far as Java programs are concerned, these models use very different abstractions and operating semantics, which are constitute barriers to popular adoption.

Conventional wisdom tells us that the effective way to treat the afore-mentioned uncertainties, or unanticipated changes in general, is through modularity and proven software decomposition principles such as information hiding [17] or design by contract [16]. However, due to the strong coupling of concurrency mechanism and application semantic, concurrency implementation and synchronization primitives are usually tightly integrated and entangled with the application code. For example, locks can appear as class variables of any class that need to be re-entrant. Inheritance is usually the only choice for a class type to use the primitives of concurrency libraries. The conventional concurrency code is typically invasive and not aligned with respect to the modular boundaries of the server logic. For this reason, concurrency is often referred to as a type of crosscutting concerns [13, 4, 14, 22] which is best treated by the aspect-oriented programming (AOP) [13] paradigm.

---

[4] The `util.concurrent` package. URL: `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`

[5] IBM NIO package. URL: `http://www.alphaworks.ibm.com/tech/aio4j`

[6] NBIO Package. URL: `http://www.eecs.harvard.edu/~mdw/proj/seda/`

We are not the first to examine the suitability of AOP to the support of concurrency. Earlier studies [4, 14] drew opposite conclusions regarding the possibilities of both the syntactic and the semantic separation of concurrency design from the application logic. We take the mid-ground. We believe that we should only abolish both the design-time and the code-level coupling between server code and *specific* concurrency models. However, the server code should stay amenable to the common characteristics of different concurrency models. We term this property "concurrency awareness". Its purpose is to make the coding structures of concurrent parts of the server logic explicit for allowing external manipulations. We formulate "concurrency awareness" as a set of programming invariants, encoding commonly observed design practices. In combination with these invariants, we create high-level abstractions for programmers to work with the differences of concurrency models through a uniform API and the associated Concurrency Aspect Library, CAL. CAL functions as a mediator between the "concurrency-aware" server logic and the diversified abstractions of concurrency libraries. We show that the use of CAL not only simplifies the coding effort but also makes concurrency implementations more explicit in terms of design. Due to the effective mediation capabilities of CAL, we can compose the same server code with a variety of different concurrency models purely by changing the compilation configurations. CAL not only incurs no observable performance overhead, but also significantly improves the performance of a commercial middleware implementation by as much as ten times through changing concurrency models according to the runtime conditions.

The contributions of our work are as follows:

1. We present the concept of "concurrency awareness" as the foundation of decoupling concurrency models from the server logic. We describe a set of programming invariants as the guiding principles for creating "concurrency awareness" in server programs.
2. We describe the design of CAL, the Concurrency Aspect Library, which allows programmers to work with concurrent frameworks of very different genders through a uniform API. CAL also effectively decouples these frameworks from the server logic at the code level. We make CAL publicly available at: `http://www.cse.ust.hk/~charlesz/concurrency` for inspections and experiments.
3. We present the quantification of our approach in terms of both the coding effort and performance measurements. We show that programming concurrency with CAL, especially against different concurrency models, can be simplified and be structurally explicit. CAL can support a complex commercial middleware system with no runtime penalty and, through adaptations, dramatically improve its performance.

The rest of the paper is organized as follows: Section 3 describes the "concurrency-aware" architecture principles and the implementation of CAL and Section 4 presents the evaluation of our approaches.

2 Related work

In this section, we present the related research on enabling the architecture-level customization of concurrency mechanisms. Please refer to [20] for a good summary of the different types of concurrency mechanisms themselves. We first present research in the code-level separation of concurrency mechanisms using AOP-like approaches, i.e., those based on aspect-oriented programming and other meta-programming approaches such as the use of annotations. We then present approaches for enabling the flexible compositions of concurrency models in server applications, not limited to Java applications. We last discuss the difference of our work compared to the AOP treatment of design patterns in general.

Java concurrency externalization. The work closest related to our approach is the assessment of concurrency and failure in distributed systems conducted by Kienzle and Guerraoui [14]. They have focused on investigating the semantic separation of concurrency and failure through the use of AspectJ in the context of transaction processing. The conclusion was that a separation is not possible. We agree that the server logic cannot be made entirely semantically oblivious to concurrency semantics. However, we demonstrate that, through making the server code concurrency-aware, it can be made semantically and syntactically oblivious to lower-level details of specific concurrency models. We delay the study of our approach in treating transaction-based concurrency to future work. Douence et al [6] introduced a generative approach to synthesize and to coordinate concurrency mechanisms defined in aspect modules. Their approach is complementary to our effort in verifying the correctness of concurrency model compositions.

D [4] is a language system for separating the distribution code from Java programs. D consists of a simplified Java language, the Cool language for composing the synchronization of threads, and the Ridl language for composing the communication between threads. The D aspect weaver is responsible for merging three language systems to produce the transformed Java sources. The main objective of D is to provide one of the first evaluations of the benefit of using AOP-like languages to compose distribution.

JAC [8] uses annotation-based hints in Java programs and the accompanied Java pre-compiler to separate the concurrency code from the operational logic of the server. The pre-compiler modifies the Java source by inserting both synchronization and concurrency code based on the annotations. We think that annotation-based approaches, despite sharing many similarities to the aspect-oriented approach, do not achieve the source-level detachment of concurrency models compared to our approach. The server implementation is hardwired to JAC-based concurrency support. With respect to our work, it is not clear how different concurrency models and the composition of these models can be supported by JAC annotations. The evaluation of the JAC approach on complex distributed systems is not reported.

Java concurrency can be entirely externalized for Java programs hosted by application servers. For example, Java server programs written as Enterprise

Java Beans (EJB) can be free of concurrency and synchronization concerns and, instead, have them configured as runtime policies understood by the EJB containers. Our work is concerned with the concurrency models used by application containers themselves. It is possible to build server programs on top of containers such as Spring[7] and have the container control the concurrency mechanisms. Due to the fact that containers typically utilize reflection to enable object invocations, we choose not to evaluate such approaches because of their significant performance overhead compared to the bytecode transformation of the AspectJ compiler.

Customizable concurrency. Many conventional approaches give server applications the flexibility of choosing the best concurrency models according to the specific server needs. SEDA [20] proposed and evaluated an architecture for Java servers utilizing asynchronous events and thread pools to partition server data flows into multi-staged pipelines. From the software engineering point of view, SEDA enables the server application to compose the most appropriate concurrency models by changing the topology and the depth of the pipeline as well as the control parameters of pipeline stages. Similarly, the ACE framework allows C++ servers to choose concurrency models adaptively through the use of C++ templates. The components of the ACE network library are in the form of parameterized templates so that the internal implementation mechanisms can be changed without affecting the user code. This is an instance of the open implementation principles [12]. The architecture adaptation of concurrency in these approaches is confined within the provided frameworks themselves. The applications are hardwired and subject to the framework capabilities, which is the exact problem we address in this work.

Aspect-oriented treatment of patterns. There have been numerous recent approaches on externalizing the implementation of design patterns with aspect-oriented programming by Kendall [10], Hannemann and Kiczales [7], and Cunha et al [5]. The externalization of patterns are realized by reusable pattern libraries implementing the roles of patterns as mix-in types and role interactions as re-targetable abstract pointcuts. Our work, inspired by this line of research, reasons about the common characteristics of concurrency patterns in general and takes the application-aspect co-design approach. As shown by our examples later in the paper, it is possible to support complex design patterns through composition of basic modules using the CAL library APIs.

3 Concurrency externalization

For the virtue of reuse and customizability, the afore-presented design uncertainties mandate the dismantling of both the design-time and the code-level coupling of server code to particular concurrency models or libraries. We achieve this goal first by making the observation that there exist common interaction assumptions which the different concurrency models make towards the server

[7] The Spring framework. URL: `www.springframework.org`

logic. The server code needs to be compatible with these assumptions and become "concurrency-aware". The main utility of an aspect concurrency library is essentially to facilitate programmers in capturing these assumptions in the server code through a uniform API. In this section, we introduce these concepts in detail.

3.1 Concurrency-aware servers

We loosely define the *concurrency-aware* server programs as programs not concurrent themselves but having salient properties about their structures and execution flows that are compatible to the common interaction assumptions of concurrency models. Finding a comprehensive list of these assumptions for all concurrency models is not an easy task. We present our initial findings which we have found to be effective as follows:

I.Captivity assumption. The primary interaction assumption of concurrency models is that certain parts of the server logic can be captured as units of concurrent activities and submitted to a concurrent executor. Popular concurrency libraries identify such parts of the server logic as instances of classes. For Java programs, a unit of concurrent activity is typically cast as an instance of `Thread`, `Runnable`[8], `Handler` [20], or `Task` in the `util.concurrent` package.

II. Execution context assumption. Each concurrent activity has an execution context that has control over the life-cycle of the activity: creation, modification, and termination. The context can be exclusive to each activity or shared among all activities. For example, creating a thread in Java through extending the `Thread` class type causes each thread to have independent object states. Creating threads through inner classes allows all threads to share the same object state.

III. Data flow assumption. Concurrent activities might have an immediate incoming data flow dependency upon their execution contexts. The context, however, typically does not have the same dependency upon the activities. For example, activities supported by thread-pools might rely on the context to perpetually supply data that are to be processed. These activities are usually continuously active and do not return control to the context until they terminate, hence, have no immediate outgoing data flow, such as passing a return value to the context.

IV. Execution mode assumption. The mode of the execution flow of the server logic can be active, with instructions executed in loops, or passive, completely subject to external activations. The execution mode is assumed to be consistent with the currency models of use. For example, for the *reactive* concurrency model [18], the concurrent activity is typically passive since it only reacts to events. Concurrent activities in models based on the abstraction of "thread" are in general active, i.e., executing in a proactive manner.

V. Synchronization assumption. The usages of synchronization primitives also need to be kept consistent with the concurrency models of use. For example, concurrent activities in the reactive model are usually unsynchronized

[8] Both `Thread` and `Runnable` are documented in the Java 5 Documentation. URL:`http://java.sun.com/j2se/1.5.0/docs/api/`

because they are always executed in a serialized manner. However, they need to be carefully synchronized for thread-based concurrency schemes. Inconsistent synchronization policies incur either runtime overhead or incorrect program behaviours.

One of the essential goals of *concurrency awareness* is to preserve these interaction assumptions in the server code. We therefore formulate concurrency awareness as a set of programming invariances as follows:

Rule 1: Group concurrent activities within concurrency-aware procedures. A concurrency-aware procedure usually satisfies three minimum requirements: (1) It has well defined termination conditions that are known to the caller; (2) It does not contain active execution controls such as persistent loops or regularly scheduled executions; (3) It does not return a value that is to be used later by the caller. In other words, we avoid the use of data that only live on the stack.

Rule 2: Localize data inflow at either invocations or instantiations. One of the major functions of concurrent activities in server programs is to process a continuous inflow of data or requests. We advocate that the data in-flow is in form of parameter passing at the time of initializing a concurrent activity or of invoking its procedures.

Rule 3: Make the concurrent activities of the server logic "synchronizable". The choices of synchronization mechanisms should be considered in conjunction with the chosen concurrency models. To protect the shared program state, we advocate making the relevant server logic synchronizable (not synchronized) by making critical regions structurally explicit, e.g., having procedural boundaries.

These structural rules are syntactic with no semantic connotations, hence, generally applicable. The first two rules are also common practices in the eyes of a veteran concurrency programmer. The last rule is to avoid any critical regions within undistinguishable code structures, which are problematic to have synchronization policies applied externally. Server code following these structural rules generally satisfies the common assumptions of many concurrency models. The physical composition between the server code and the concurrency libraries is facilitated by CAL, the Concurrency Aspect Library, presented in the next section.

3.2 Concurrency Aspect Library

The core of our externalization approach is the Currency Aspect Library CAL. CAL aims at providing high level abstractions to hide the details of concurrency models and to enable a closer correspondence between the concurrency design and the code. We design CAL with the following specific goals in mind:

1. **Oblivious.** The library should allow concurrency implementors to focus on expressing the concurrency in terms of application semantics while remaining oblivious to the details of the concurrency frameworks, as long as the server code remains concurrency-aware. This is a crucial requirement for achieving the separation of server semantics from concurrency mechanisms.

2. **Versatile.** The library should be capable of supporting concurrency frameworks of very different mechanisms and type abstractions. Neither design alteration nor coding changes are required for the server code if we choose to switch from one framework, such as a reactive model, to another, such as one based on threads.
3. **Uniform.** The library should provide simple and uniform programming interfaces to facilitate the implementation of concurrency. The programming effort required to use Framework A should not differ significantly from the use of Framework B. Otherwise the library is not effective in capturing common interaction assumptions.
4. **Efficient.** The library should only incur acceptable runtime overhead as a trade-off for the structural flexibility. For server programs, performance, more specifically, throughput and fairness, is the vital quality metric not to be significantly compromised.

We now describe our library from two perspectives: the static perspective of dealing with the diversification of types in concurrency libraries through "type mediation", and the dynamic perspective of integrating concurrency mechanisms with the server execution flow through "activity capture". We believe the first two design goals can be validated after the design of the library is presented in detail. The quantitative evaluations of these design goals are deferred until Section 4.

Type mediation: We have previously argued that the common interaction assumptions of concurrency models center around "activities" and "contexts of activities". In CAL, we use the entities `Activity` and `WorkingContext` to represent these two concepts. These two concepts mediate between the server-specific concepts and concurrency models through a two-step type-space adaptation process accomplished by the CAL user. The first step adapts towards the abstract data types in the concurrency frameworks, and the second to the server class types. The automatic adaptation is performed by the library if the activities are mapped to the call-sites of methods. We term this type of activity an *auto activity*. Auto activities might share the same working context if the mapped method invocations are made by the same caller. This simple scheme decouples the direct type-space wiring between the server code and concurrency frameworks. The liability lies with the generality of the library concepts in representing the interaction assumption of concurrency models. As we will show in our evaluation, we have found that our existing concepts are quite adequate with respect to a broad range of different concurrency schemes.

We illustrate the type mediation process in Figure 5, which depicts three sets of domain models. The concurrency models are exemplified by three popular schemes on the top of the drawing: *reactive*, *thread-per-task*, and *thread pooling*. The outer box with dotted borders represents the state of the program with respect to the concurrent executions. The inner light-shaded boxes represent concurrent activities (the "wired" box denotes the "thread" abstraction). The consistent shading and arrows across the concurrency and the library models signify the "representation" relationship. Arrows with bold lines represent the

"mapping" relationship between library models and server domain models are represented as UML diagrams on the bottom of the graph.

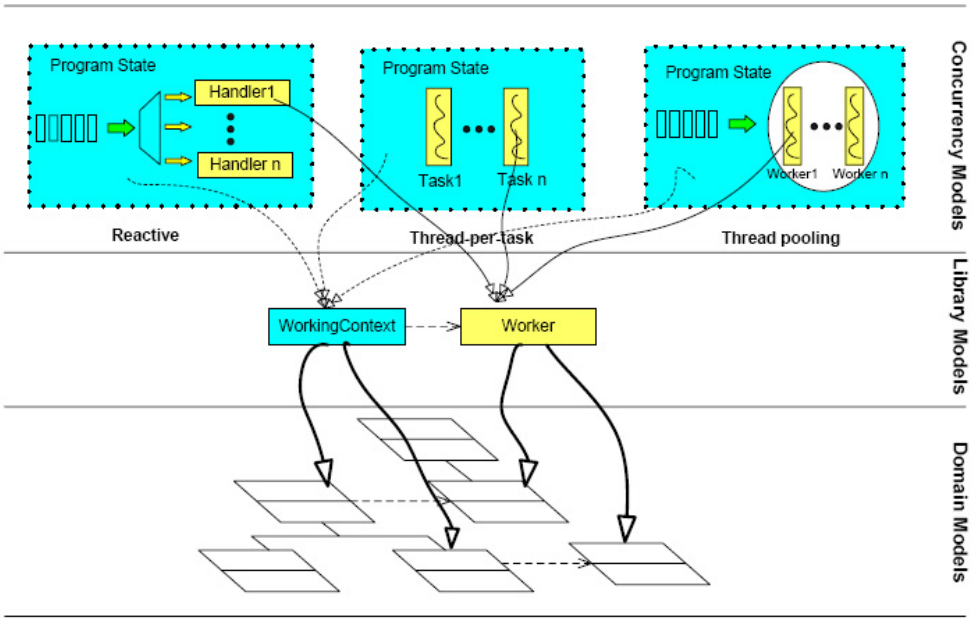

Fig. 5. Concept mediation.

Activity capture: The primary purpose of the type mediation is to flexibly integrate multiple concurrency libraries into a unified type space. To integrate the dynamic execution flow of the server logic into these libraries, we provide programming APIs for the users of the library to identify, based on domain knowledge, the appropriate dynamic execution points in the server program where the concurrent activities can be "captured" by the library. These APIs are in the forms of AspectJ [11] pointcuts:

`captureOnInstantiation` is a `call`-based pointcut used to identify the *instantiation* relationship between the `WorkingContext` and the `Activity`. The pointcut is typically mapped to constructor calls or factory methods. This entails that one class type is associated with one particular concurrency model with its concurrency-aware method adapted by the generic library interface. If the adapted concurrency-aware method of the class is invoked somewhere in the server code, the invocation needs to be canceled. The invocation can be canceled because, by our definition of concurrency-awareness, there is no data dependencies between the concurrency-aware procedures and their calling stacks. We

provide the `cancelCall` pointcut to automate this action.

`captureOnInvocation` is a `call`-based pointcut used to identify the *invocation* relationship instead. In this case, method invocations are created as the new concurrent activities. Compared to `captureOnInstantiation`, one advantage of this finer granularity of activity capturing is that it allows one class to have a different concurrency scheme per method, if the method is invoked by different callers. Our implementation of the method-level activity captivity essentially creates a Java inner class per method invocation, which, in spite of incurring no runtime overhead in our experiments, can be an expensive operation.[9]

In Figure 6, we show a simplified version of an AspectJ module in CAL, which supports the activity capture on method invocations for the Java 5 executor framework. Line 5 is the abstract pointcut, part of the library API, to be mapped to the invocation of a concurrency-aware procedure in the server code. Lines 10-16 execute in place of the procedure invocation by the `around` advice. The inner class (Lines 10-13) performs the type-space adaptation of the library native type, `IExecutorActivity`, to the Java executor interface type, `Runnable`. Line 15 submits method executions as inner classes of type `Runnable` to the executor framework.

```
   public abstract aspect DynaConcurrency<T extends IExecutorActivity> {
2        private IWorkingContext.executor_ = Executors.newCachedThreadPool();

4        /* This is the method(s) to be detached as a thread.*/
         public abstract pointcut captureOnInvocation();
6
         /* The generic activity creation on a method call.*/
8        boolean around(final IWorkingContext context):captureOnInvocation()&&this(context){
                 /* * Here we use inner class to wrap the method call captured by the joinpoint. */
10               class T implements Runnable{
                         public void run(){
12                               while(proceed(context));
                         }
14               }
                 context.executor_.execute(new T());
16               return true;
         }
18 }
```

Fig. 6. Implementation of Concurrency Aspect Library

3.3 The CAL Implementation

To verify the fundamental concepts of our aspect oriented approach, we have created an aspect library consisting of the support for four types of representa-

[9] Java duplicates the runtime state of the parent class for each inner class created.

tive and dramatically different concurrency models. All implementations assume that the concurrency-aware procedure uses a Boolean return value to signal the termination condition.

Reactive. Central to our reactive concurrency library is a simpler version of the Reactor [18] event multiplexer using the Java `nio` package. The server processes client requests in a single thread of execution, demultiplexing I/O events to a collection of `IAsyncWorkers`. In addition to the *type mediation* and *activity capture* functionalities, our library enables the automatic and seamless socket replacement for creating the asynchronous counterparts of the synchronous Java sockets and I/O stream classes. The replacement is realized by intercepting the creation process for synchronous sockets and streams using AspectJ advices. Our library implementation is capable of supporting 10K simultaneous connections on a 2GHZ commodity PC[10].

Executor framework. We have implemented the mediation and the capture capabilities leveraging the new Java 5 Executor concurrency framework[11]. Among many capable concurrency models provided in the executor framework, we chose to implement support for the thread-per-activity and the pooled-thread models.

JCSP framework. JCSP [19][12] is a Java framework implementing the concepts of the Communication Sequential Process [9]. JCSP facilitates the creation of "verifiable" concurrent programs for which the model-checking techniques can be used to check for concurrency problems such as race conditions, deadlocks, or livelocks. Our JCSP aspect library executes concurrent activities as JCSP processes. The pooling model is implemented leveraging the inherent synchronizing capabilities of JCSP `Channels`. Special support is needed for common synchronization mechanisms such as *locks* and *synchronized regions* due to the lack of these primitives in the CSP vocabulary.

Native Java thread. We have also implemented support for the native Java `thread` class as a representation of the conventional approach to multithreading. The *thread pool* model is implemented as a fixed number of threads feeding on an *activity* queue.

3.4 Example

We now go back to our simple server, presented in Section 1, to illustrate how users of CAL can compose different concurrency models without code modifications. We present two examples. The first example shows how CAL APIs are used to support the Java 5 executor framework. We then showcase the composition capabilities of CAL by building a multi-staged hybrid concurrency model from two basic ones: the executor framework and the event-driven model, again requiring no changes to the server code.

[10] The C10K problem. URL: `http://www.kegel.com/c10k.html`

[11] Java Executor. `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executor.html`

[12] Communicating Sequential Processes for Java URL:`http://www.cs.kent.ac.uk/projects/ofa/jcsp/`

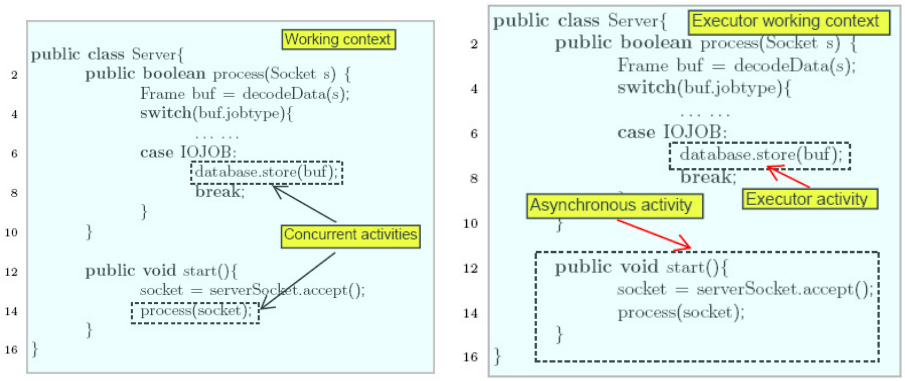

Fig. 7. (A) Analysis for Executor (B) Analysis for Hybrid

Java executor framework. The Java executor framework is a new addition to the Java platform that offers improved support for concurrency. We first present an abbreviated version of the server code in Figure 7(A). We identify, with dotted rectangles, two activities that can be executed concurrently: the establishment of a new connection (Line 14) and the persisting of uploaded data (Line 7). The concurrency implementation is given in Figure 8. Given some degree of familiarity with the AspectJ syntax, one can see that this implementation looks very close to an actual design blueprint. The user first determines that the type `Server` encapsulates some concurrent activities, represented by dotted rectangles (declare `Server` as the *WorkingContext* at Line 4). She then explicitly specifies two method invocations to be executed concurrently ("concretize" the abstract pointcut at Lines 7-9). Lines 11-14 are not part of the library usage, however, they are necessary to switch the server into the active execution mode. The example shows that the most important difference of our library approach, compared to conventional ways of concurrency programming, is that the concurrency perspective of the server program is not only *modularized* but also more *explicit* and *descriptive*.

Multi-staged concurrency model. In reality, complex server programs often use a combination of concurrency models to maximize the processing efficiency. For our simple server example, it could be more efficient to provide the data uploading service by using the event-based model to accept new incoming connections and the executor framework to dispatch the database operations in separate threads. In this way, we avoid the threading overhead for a higher number of clients and pipeline the incoming data towards the database service. Figure 7(B) depicts our design: The entire `start` method definition is identified as a unit of asynchronous activity and the call to the `Database.store` method

```
public privileged aspect EXConcurrency extends DynaConcurrency<IExecutorActivity> {
2       /* Type adaptation */
        declare parents: Server implements IWorkingContext;
4
        /* Capture two method calls as concurrent activities*/
6       public pointcut captureOnInvocation():
                call(* Server.process(..)) || call(* Database.store(..));
8
        Object around():call(* Server.start()){
10              /* Switch into the active execution mode*/
                while(true)
12                      proceed();
        }
14  }
```

Fig. 8. Implementing the executor framework through CAL

is to be executed concurrently by Java executors. The `Server`, therefore, is both an executor working context and an asynchronous activity[13].

We realize this implementation with two aspect modules in less than 20 lines of code as presented in Figure 9. In the reactive stage implementation, Line 3 maps a special pointcut defined in the reactive library to signify when the reactor starts to gain control of the program execution. Line 4 captures the instance of `Server` at creation as an *async activity*. Lines 6-8 perform the type adaption, and Lines 10-12 are to associate the `Server` class with the correct dispatching key. This example demonstrates the modular "composability" of CAL in building complex and multi-staged server programs as those described in the SEDA project [20]. Benefiting from the externalized approach, the server code remains oblivious to specific concurrency implementations. We have the option of using an entirely different concurrency model that is possibly better in a different service context.

3.5 Synchronization.

The externalization of synchronization is a challenging topic. In the context of this paper, we present the synchronization externalization as an implementation issue and delay further discussion to future work. We address synchronization in Java programs via two simple rewriting rules: (1) For methods to be declared with the `synchronized` keywords, we use `around` advices with `execution` pointcuts to enclose the method body within the `synchronized` blocks. (2) For block-level, i.e., intra-method, critical regions, we factor the block into a method, if necessary, and enclose the call-site of the method with either `synchronized` blocks or other synchronization primitives, such as `wait/notify` pairs, through the `call`-based `around` advices.

[13] Note that in this case `Server` is not a working context because we identify its method definition as the activity

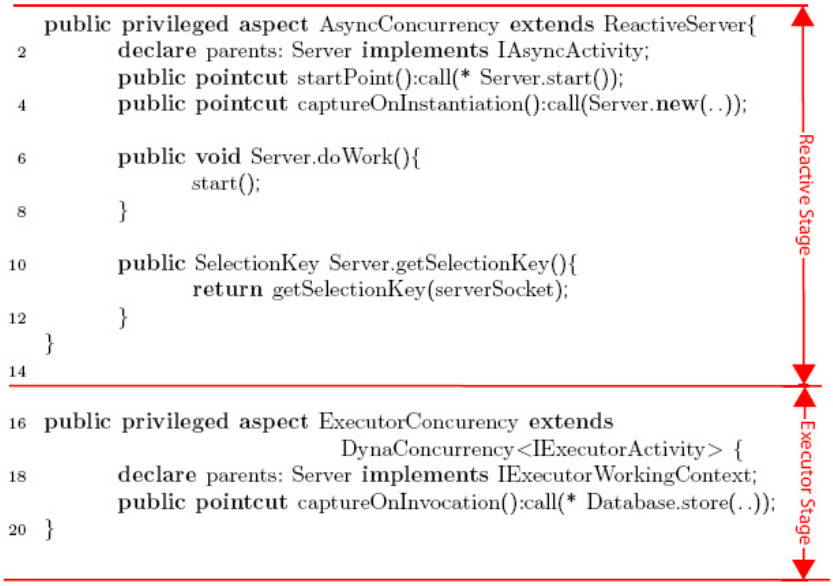

Fig. 9. Implementing the hybrid model through CAL

Special implementation concern is given to the JCSP library for its lack of common Java synchronization operations such as *synchronized regions* and *wait/notify/notifyAll* operations. These Java primitives are supported, with the exception of the *notifyAll* semantics, by the JCSP concept of `Channel` as we emulate mutual exclusion as the exclusive communication between the executing thread and an oracle. The *notifyAll* semantics is supported by the JCSP concept of `Bucket` for its capability of releasing multiple threads simultaneously. Our implementations support the correct operations of the systems that we have evaluated with negligible runtime overhead. The functional evaluation is presented in Section 4.

4 Evaluation

We intend to achieve four design objectives with our aspect-oriented library approach to concurrency externalization: *obliviousness*, *versatility*, *uniformity*, and *efficiency*. To evaluate these design objectives, we have used the CAL library on two server programs: our simple server presented earlier as a micro-benchmark, and ORBacus[14], a commercial middleware implementation. With respect to these two server programs, our evaluation aims at answering two questions:

[14] ORBacus URL:`http://www.iona.com/orbacus`

1. How effective is the concurrency awareness concept and the CAL API in simplifying concurrency implementations when working with diversified concurrency models?
2. What is the runtime cost of the CAL-based server implementations as a trade-off for the configuration flexibility compared to conventional approaches?

We answer the first question through the static quantification regarding the structures of the CAL user code. We provide insights to the second question by extensive runtime simulations. The rest of the section proceeds as follows. We first describe the relevant physical attributes of the CAL implementation and the applications of CAL to both the micro benchmark and ORBacus. We then present the quantification of the coding quality of the CAL-based implementations, followed by the metric-based runtime evaluations.

4.1 The CAL implementations

As aforementioned, the CAL library consists of support for four different types of concurrency models: Java thread, event-driven, Java executor framework, and JCSP framework. Each model is supported by CAL types extended from the generic `Activity` and `WorkingContext` interfaces. For instance, the `Runnable` interface of the *thread* and the *executor* models are adapted by `IThreadActivity` and `IExecutorActivity` interfaces in the library, respectively. For each concurrency model, we implemented both the bounded and unbounded versions with respect to the number of concurrently executed tasks. The unbounded version admits as many concurrency activities as possible and the bounded version uses a "thread" pool that feeds on a queue of CAL *activities.*[15] The library is fairly light weight, consisting of 84KB in total bytecode size. We prove by implementation that the concepts such as `Context` and `Activity` are compatible to the chosen concurrency models.

We implemented the four concurrency models for both the example server presented throughout the paper and the ORBacus[16] object request broker (ORB) using CAL. ORBacus is implemented in Java. It supports the full CORBA 2.4 specification[17] and is being commercially deployed. It consists of around 2000 Java classes and 200K lines of code. The network communication components of ORBacus use the thread-per-connection model to serve the incoming clients. Refactoring was first performed to remove the native concurrency implementation from these components. We then make them concurrency-aware by removing the loop structures and the synchronization code. The relevant method definitions in the original implementation do not have data dependencies over the callers. They also have well defined termination conditions defined by state

[15] Due to the technical difficulty of sharing `Selector` across threads, we implemented the pool version of the event-driven model essentially as to balancing the load among concurrently running `Selector` event loops.

[16] The ORBacus ORB. URL: `http://www.iona.com/products/orbacus.htm?WT.mc_id=1234517`

[17] CORBA 2.4 URL: `http://www.omg.org/corba`

variables. These variables are accessible by AspectJ constructs and checked in the library user code. Applying CAL to the simple server allows us to better assess the performance characteristics of CAL without being influenced by the operational complexity of the server. ORBacus, on the other hand, serves as an experimentation of how our concurrency externalization approach benefits non-trivial and sophisticated server programs.

4.2 Coding effort

To assess the coding effort of the concurrency implementation using CAL, we examine the static code structures of the CAL user code for both the micro-benchmark and ORBacus. Our hypothesis is that, if CAL effectively supports model variations, the effectiveness can be reflected in two ways: (1) One does not need to write a lot of code to use a concurrency model and (2) one does not need to change the code dramatically to switch to a different concurrency model. Note that the server code stays the same for all of the models.

In Table 1, we enumerate the AspectJ language elements used in the user code of CAL as a way of reflecting the coding effort as well as the structural similarity in dealing with the four concurrency models. Each model, including the pool version, is supported by one aspect module, corresponding to a row in the table. For ORBacus, we also report the aspect-oriented synchronization implementations for both the thread-based models and the JCSP model[18]. We observe that for both cases, in addition to the light coding effort[19], the coding structure among the bounded as well as the unbounded (pool) versions are almost identical. In fact, the actual code only differs from each other for extending different interface types. Interested readers are invited to verify this themselves by obtaining a copy of our implementation[20]. The simplified coding effort reflects the effectiveness of the high-level abstractions provided by CAL in matching code with design. The similarity of the coding structures shows that the CAL abstractions capture the common characteristics of the chosen concurrency models.

4.3 Runtime assessment

To assess the runtime overhead of concurrency through CAL, we measure the server throughput as well as the client-side fairness on server machines based on both single-core and multi-core CPUs. To benchmark the two servers, we simulate four types of client/server communication patterns: the sustained (CPU) or the periodic (CPU short) connections for CPU-bound factorization requests, and the sustained (IO) or the periodic connections (IO short) for sending data chunks of 80K bytes. In all experiments, we use the thread-per-connection model,

[18] Recall that the synchronization semantics in JCSP is based on channels.

[19] The event-driven model needs more coding effort to associate asynchronous activities with dispatching keys as illustrated in Figure 9.

[20] The examples can be downloaded from `http://www.cse.ust.hk/~charlesz/concurrency`

Model	LOC		pointcut		type ITD		method ITD		advice	
	MB	ORB	MB	ORB	MB	ORB	MB	ORB	MB	ORB
event	47	37	3	3	1	4	3	6	1	0
executor	10	46	2	3	1	3	0	3	1	1
thread	9	46	2	3	1	3	0	3	1	1
csp	9	46	2	3	1	3	0	3	1	1
exe.pool	12	50	2	3	1	3	1	4	1	1
thread.pool	12	50	2	3	1	3	1	4	1	1
csppool	12	50	2	3	1	3	1	4	1	1
thread sync	N/A	37	N/A	1	N/A	1	N/A	0	N/A	2
csp sync	N/A	39	N/A	1	N/A	1	N/A	0	N/A	2
Ave.	16	45	2	3	1	3	1	3	1	1

Table 1. Structural comparison for the micro-benchmark (MB) and ORBacus (ORB). LOC:lines of code. ITD: inter-type declaration.

implemented in vanilla Java as the baseline for comparison. For each server, we produce 8 runs. Due to space limits, we selectively report 4 runs for each case. The fairness is calculated from the average, m, and the standard deviation, δ, of the number of messages sent by each client as follows: $fairness = (m - \delta)/m$. It is a measure of how much clients deviate from the mean in successfully being serviced by the server.

Micro benchmark: The measurements for the micro-benchmark are reported in Figure 10. The first conclusion we draw from our observations is that CAL does not incur noticeable performance overhead because the baseline performance is not consistently better than any CAL implementations in any case. For long CPU-bound connections, the event-driven model (`async`) significantly outperforms the other models when the number of clients increases on the single CPU server. This is not true on the dual-core machine (`CPU(Multi)`), as the single-threaded event-driven model (`async`) becomes significantly more costly to use when the number of connections is less than 2000. This is because thread-based models can be boosted by the dual-CPU configuration. The problem is solved by our load-balanced event-driven version (`asyncpool`). For the I/O-bound periodic connections (`IO Short (Single)`), the baseline is at par with other models except the executor framework . The sustained I/O-bound connections on the dual-core machine (`IO (Multi)`) are well serviced by thread-based models, whereas the event-driven models, whether load balanced or not, suffer severely.

In summary, our experiments confirm that, for this simple server, none of the concurrency models we chose to implement scales well for all connection types and processor profiles. Fortunately, due to the externalized approach, we are able to flexibly choose the most appropriate concurrency implementations and always outperform the baseline.

ORBacus: For the performance evaluation of ORBacus, we created two CORBA server objects, one performing the factorization task and the other

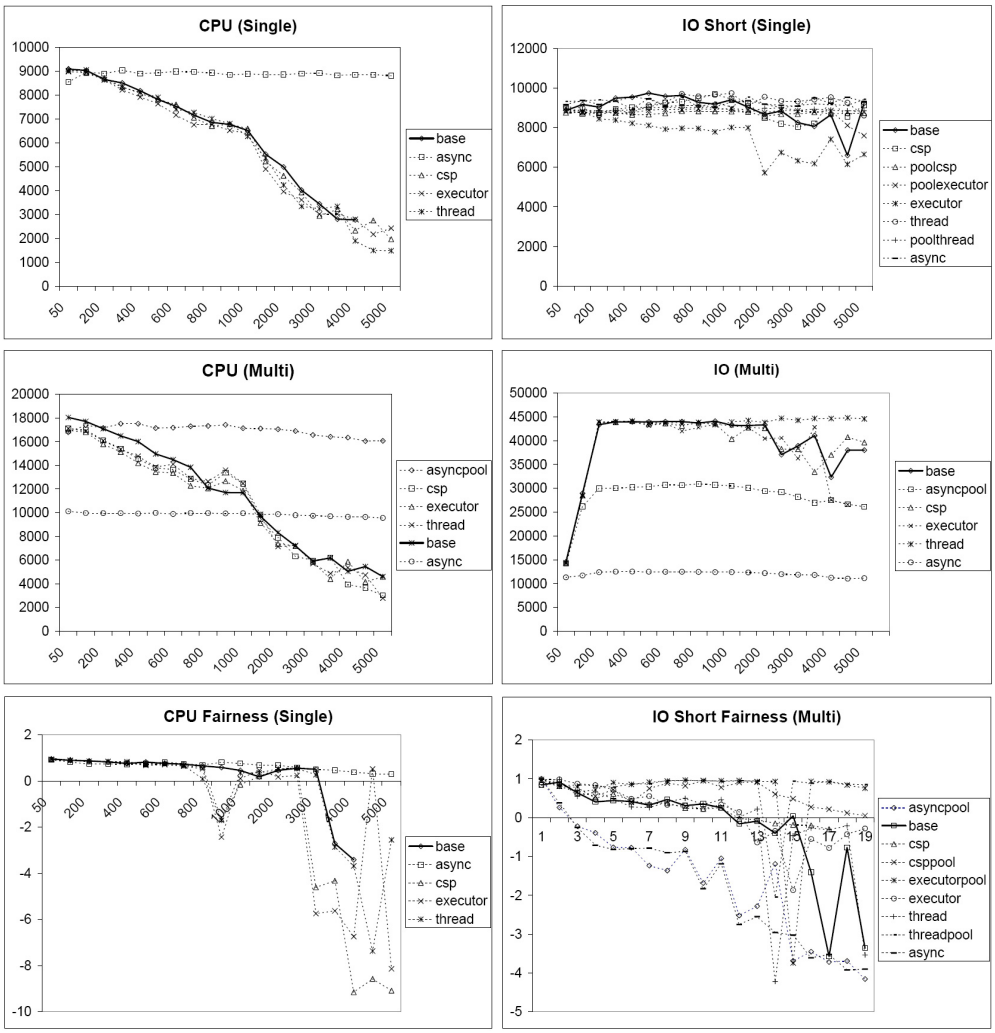

Fig. 10. Measurements of the example server for the CPU intensive connections on both types of servers, the short I/O connections on the single CPU server, and the sustained I/O connections on the dual-core server. In all charts, the baseline measurements are plotted as thick solid lines for the ease of visual comparison. The technical settings of the single-core experiments are identical to the ones presented in the introduction section. The multi-core experiments are conducted on two server machines with Intel Xeon dual core CPUs clocked at 1.8GHz with 4MB on-chip cache and 3GB physical memory. These two servers are both running Linux 2.6 kernels and connected by a 1Gbps switched network. All experiments are carried out by the JRockit R27.3.1-jre1.6.0_01 JVMs with 1M memory limitations (Xmx flag) and 156K maximum stack size (Xss). The pool size is fixed at 100.

simply receiving the inbound data chunks. We observe that, in accordance to the case of the micro-benchmark, the CAL-based approaches do not incur perceivable performance overhead. For the I/O-bound measurements (`IO (Single)` and `IO Short (Multi)`), the server throughput generally degrades as the number of client ORBs increases. This behavior is different compared to the micro-benchmark version in which case the server scales well. This is due to a particular demarshalling mechanism used in the request broker. The demarshalling process is a CPU-bound operation carried out for each incoming data chunk for decoding the middleware frames from the network data. The demarshalling process is in contention with the thread context switches on the CPU resources. The sustained CPU-bound connections (`CPU (Multi)`), in accordance to the case of the micro-benchmark, are well serviced by the load balanced event-dispatching model. In the case of `CPU Short (Single)`, we observe that the thread pooling is very effective in servicing periodic short CPU-bound requests as all of the three pooling versions have near constant throughput. The event-driven model is most suitable for both kinds of CPU-bound connections. As for the fairness measures, the Executor and JCSP frameworks have consistently the worst fairness measures compared to all other versions. We suspect this is due to the intrinsic mechanisms of these frameworks not to the use of CAL because the fairness of the baseline is not consistently better than the rest of the models.

In summary, our experiments reveal that we made significant improvements over the original implementation for the continuous and periodic CPU-bound requests by changing the concurrency models to either the event-based or the pool-based. The improvements range from around 50% on the single CPU server to about 4-10 times on the dual-core CPU. The benefit of changing concurrency models for the I/O-bound requests, however, is not significant compared to the original implementation of ORBacus. On the one hand this shows that the use of CAL achieves the same performance as the conventional approach and, at the same time, exhibits the customization flexibilities. On the other hand, it poses new challenges for us to improve processing for requests that are both I/O and CPU intensive. Our future work will address this issue.

5 Conclusions

The ubiquitous trend of network-based computations require the architecture of concurrency in today's server programs to adapt to the large variations of runtime conditions. Consequently, server programs often need to employ multiple concurrency models that can be very different in nature. In this work we propose CAL, the Concurrency Aspect Library, as a way of both raising the level of abstraction for programming concurrency and, more importantly, separating the specific semantics of concurrency models from the server architecture.

We have presented CAL as both a design methodology and a prototype implementation. Server programs leveraging CAL must maintain a set of coding invariants to become "concurrency-aware". This is an effective compromise between the conventional approach of invasive concurrency programming and the seman-

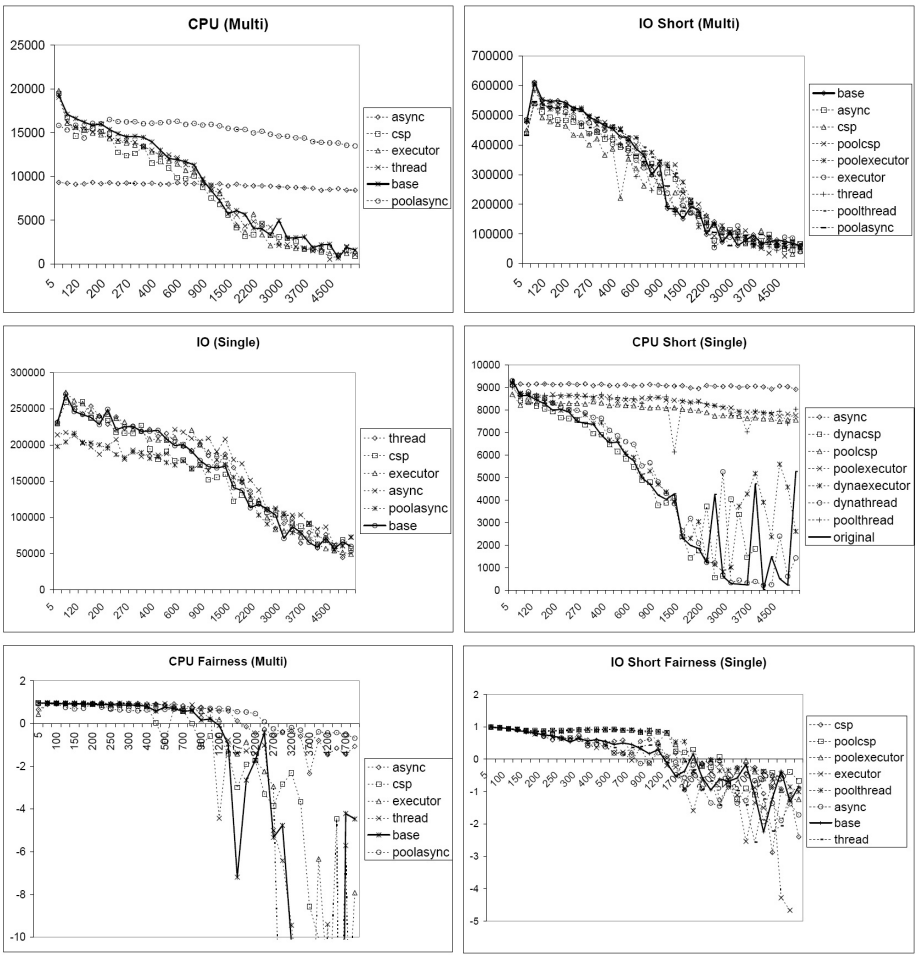

Fig. 11. Measurements of ORBacus: Sustained CPU-bound (`CPU (Multi)`) and periodic I/O-bound (`IO Short (Multi)`) requests on dual-core servers; Sustained I/O-bound (`IO (Single)`) and periodic CPU-bound (`CPU Short (Single)`) requests on single-core servers. For fairness measures, we present the plots for the sustained dual-core CPU-bound and the periodic single-CPU I/O-bound. The technical settings are identical to the previous case except that the Xss flag is not used.

tic obliviousness of concurrency that is highly desired but difficult to achieve. Concurrency-aware server programs are amenable to the basic interaction assumptions of many intrinsically different concurrency models. CAL provides a uniform programming interface consisting of both static type and dynamic execution abstractions for programmers to work with the concurrency-aware server code and CAL hides the semantic details of individual concurrency models.

Our general conclusion is that the externalization of concurrency mechanisms using CAL is effective. The effectiveness of the CAL approach is based on two factors: the imposed programming invariants need to give applications the customization and performance advantages; the API abstractions of CAL need to reduce the programming effort in coding architecture customizations for the concurrency libraries. To evaluate these design objectives, we implemented the CAL support for four different concurrency models and applied CAL on two server programs. Our observation is that even for a sophisticated server program the coding effort of programming concurrency with CAL is both light and consistent in spite of model differences. The server code, without being modified, can be composed with the four concurrency libraries either individually or in combination. We have also shown that the code written based on CAL has closer correspondence to the design of concurrency as compared to conventional approaches.

We presented the evaluation of the performance of the CAL-based concurrency on both single and multi-core CPU hardware platforms. We conclude that CAL does not incur observable runtime overhead on both kinds of platforms while enabling customization and flexibility. Compared to conventional approaches, the CAL-based concurrency customization can produce speedups as much as ten fold. The load-time transformation capability of AspectJ allows us to make very delayed concurrency customization decisions for servers after gathering sufficient information about how the servers are being used in deployment.

We are only at the initial stage of the concurrency externalization research. We plan to continuously gain experience with the applications of CAL by evaluating more broader types of server programs including Web servers and database servers. We will continue to validate our design by enriching the concept of concurrency awareness and the capabilities of CAL. In particular, we plan to focus on the externalization of synchronization and intend to explore the meaning of "synchronizable code" as well as the structure of a synchronization aspect library. We also plan to study more sophisticated load characteristics and think about how concurrency customizations can help with server loads that are not exclusively CPU or I/O bound.

References

1. Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.
2. Frank Buschmann and Regine Meunier. *A System of Patterns.* John Wiley & Sons, 1997.

3. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *USENIX OSDI*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.
4. Gregor Kiczales Cristina Videira Lopes. D: A Language Framework for Distributed Programming. TR SPL97-010, P9710047 Xerox PARC.
5. Carlos A. Cunha, ao L. Sobral Jo and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD*, pages 134–145, New York, NY, USA, 2006. ACM.
6. Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *GPCE*, pages 79–88, New York, NY, USA, 2006. ACM.
7. Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *ACM OOPSLA*, pages 161–173. ACM Press, 2002.
8. Max Haustein and Klaus-Peter Löhr. JAC: Declarative Java Concurrency. *Concurrurrent Computing: Practice & Experience*, 18(5):519–546, 2006.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
10. Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *ACM OOPSLA*, pages 353–369. ACM Press, 1999.
11. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072, pages 327–355, 2001.
12. Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail C. Murphy. Open implementation design guidelines. In *IEEE ICSE*, pages 481–490, 1997.
13. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.
14. Jörg Kienzle and Rachid Guerraoui. AoP – does it make sense? the case of concurrency and failures. In *ECOOP*, Lecture Notes in Computer Science 2374, Springer Verlag, pages 37 – 54, 2002.
15. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transaction of Software Engineering*, 3(2):125–143, 1977.
16. Bertrand Meyer. Design by contract. *Advances in Object-Oriented Software Engineering*, pages 1–50, 1991.
17. D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–58, December 1972.
18. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, Ltd, 1 edition, 1999.
19. Peter H. Welch, Neil C. Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *CPA*, pages 349–369, July 2007.
20. Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SOSP*, pages 230–243, New York, NY, USA, 2001. ACM.
21. Matt Welsh, Steven D. Gribble, Eric A. Brewer, , and David Culler. A design framework for highly concurrent systems. UC Berkeley Technical Report UCB/CSD-00-1108.
22. Charles Zhang and Hans-Arno Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, November 2003.