# Towards An Event-Driven Programming Model for OpenMP

Xing Fan, Oliver Sinnen and Nasser Giacaman
fxin927@aucklanduni.ac.nz, {o.sinnen,n.giacaman}@auckland.ac.nz
Department of Electrical and Computer Engineering
University of Auckland, New Zealand

***Abstract*—The event-driven programming pattern is pervasive in a wide range of modern software applications. Unfortunately, it is not easy to achieve good performance and responsiveness when developing event-driven applications. Traditional approaches require a great amount of programmer effort to restructure and refactor code, to achieve the performance speedup from parallelism and asynchronization. Not only does this restructuring require a lot of development time, it also makes the code harder to debug and understand. We propose an asynchronous programming model based on the philosophy of OpenMP, which does not require code restructuring of the original sequential code. This asynchronous programming model is complementary to the existing OpenMP fork-join model. The coexistence of the two models has potential to decrease developing time for parallel event-driven programs, since it avoids major code refactoring. In addition to its programming simplicity, evaluations show that this approach achieves good performance improvements consistent with more traditional event-driven parallelization.**

***Keywords*-OpenMP, parallel programming model, event-driven programming, asynchronous programming**

## I. Introduction

OpenMP is the de facto specification for shared memory parallel programming. Its evolution and new specification extensions have gradually increased its popularity in recent years, and now OpenMP programming is widely used in different types of high-performance computing. However, there are still some barriers which make OpenMP not very suitable for an increasingly essential class of software development: the development of interactive desktop applications and mobile apps. As multi-core devices have become commonplace for the average consumer, especially in the era of ubiquitous computing, it is reasonable to draw the attention of the parallel programming model for the development of everyday applications. Achieving this will allow a larger subset of software apps to really experience the benefit of parallel execution on multi-core devices.

With the interactive nature of these desktop applications and mobile apps, the program flow is executed according to events generated during runtime, known as the event-driven model. Event-driven frameworks are examples of *inversion of control* [11], which assists developers by only requiring them to be responsible for implementing the event handlers (or callback functions). Although there are various frameworks that differ in regards to their implementing languages and supported platforms, the underlying mechanism is very similar. From the programmer's perspective, they do not need to understand the underlying runtime and its event dispatching, therefore the core part of the application development is implementing the handling routines to reach the required functionality of the application.

In an event-driven application, an event dispatching thread (EDT) is solely responsible to drive the event-loop. Once the application is launched, the runtime support listens for events generated, and queues the event if it is bound to any handling code or callback function the developer implemented. The callback function is then executed by the EDT. If a particular event handling callback function is time-consuming, the EDT will not be able to handle another event in the event-loop until it finishes execution of the callback function. The problem emerges when the callback function is CPU-intensive or I/O-bound, with the long execution time of the callback function affecting responsiveness of the overall application. For batch-like programs, the motivation for using parallelization techniques is the always aim to decrease the wall clock time. But when it comes to the event-driven programs, performance is not only evaluated by the reduction of wall clock time. Instead of focusing on execution speedup, maintaining a better responsiveness (and therefore positive user experience) is the main reason programmers incorporate concurrency.

Due to its focus of accelerating compute-intensive and batch-like programs, OpenMP mainly stresses on the parallelization of loops and symmetrical data processing. Under this consideration, the fork-join model has always been intimately infused into OpenMP, and continues to remain strongly integrated [24]. The fork-join model works well for batch programs and CPU-intensive computations; when the program is launched, its execution rarely interacts with I/O. This is because the workload and work flow is largely predefined, allowing for easier reasoning regarding the work distribution. Unfortunately, there are key drawbacks in the traditional OpenMP fork-join model making it incompatible with the co-use of the event-driven programming model.

By its nature, all callback functions are executed by the event dispatching thread (EDT) when the binding event is generated in the event-handling framework. The first challenge facing programmers is conceptually justifying whether a particular computation should be classified as a parallelization candidate. Traditionally, for batch-like programs, programmers would rarely consider parallelizing computations that last only a few seconds. But with interactive event-driven applications, even computations lasting only a few hundred milliseconds de-

mand concurrency to avoid the appearance of an unresponsive application. For OpenMP to be embraced for these mainstream applications, the introduction of additional overhead for the concurrency of shorter computational spurts needs to be less of a dilemma for programmers.

Regardless of the overhead, the fork-join model presents a much more fundamental issue for event-driven applications. Even with the potential speedup benefits, the traditional fork-join model forces the master thread (which would be the EDT in event-driven applications) to participate in the work-sharing region. This immediately goes against the policies of event-handling frameworks, as the EDT spends a noticeable amount of time away from the event-loop (thereby delaying responses for subsequent events in the application). The traditional way event-handling applications solve this responsiveness problem is by explicitly offloading the time-consuming execution to background threads and then enabling the EDT to return to the event loop to handle another event. While this has long been the standard practice in the realm of event-based applications, it is deprived of the elegance of OpenMP, particularly the paradigm of incremental parallelization that avoids major code restructuring.

Initially, it may appear that OpenMP presents an asynchronous solution with its task directive. However, a block surrounded by a task directive will be asynchronously executed by the OpenMP thread group; an orphaned task directive will execute sequentially unless it is surrounded by a parallel directive. This means the effectiveness of OpenMP tasks are confined within an OpenMP parallel region, conforming to the fork-join model that OpenMP adopts. Since the parallel directive does not provide any option to achieve asynchronization with the parallel region (for example, there is no nowait or async clause), this means that the main thread is forced to wait until every thread in the parallel team finishes its work. This inherently synchronous "join" aspect of OpenMP makes it difficult to integrate OpenMP with the event-driven programming paradigm.

Upon this event-driven programming background, as a clarification, we define **synchronous** if every event handler is directly processed by the EDT sequentially. **Asynchronous** is defined as the event handling being offloaded as a task from the EDT to a background thread, but the task is done sequentially. Parallelization is distinguished from asynchronous, and refers to execution of a handler with multiple background threads. In **synchronous parallel**, multiple worker threads are utilized, *but* this parallelization is foregrounded with the EDT assuming role of master thread. In comparison, **asynchronous parallel** means that the event handling code is offloaded to the background, *and* then executed in parallel. In this regards, the EDT does not participate in the parallelization.

In this paper, we first formally define the cumbersome, yet necessary, restructuring that is demanded to achieve concurrency in an event-driven application. We then propose a simple but expressive programming model for asynchronous programming, especially for event-driven programming. An *asynchronous executor* model is introduced in the spirit of OpenMP, to overcome the hassles associated with code restructuring. We show that using this model simplifies, as well as unifies, the parallelization and concurrency of event-driven applications. The integration of this model with the traditional fork-join model enables for a wider range of target applications for OpenMP. This allows applications that require both asynchronous execution (for event-handling responsiveness) and parallel acceleration (for reduced computational times) to seriously consider OpenMP. The semantic design pattern strictly follows the philosophy of OpenMP, in which adding directives does not influence the original correctness of the sequential execution.

These concepts have been implemented for Pyjama[1], an OpenMP-like compiling tool for Java [36]. Its source-to-source compiler and its runtime support help programmers to quickly develop applications with the asynchronization and parallelization support.

The reminder of this paper is structured as follows. Section II reviews the background of developing high-performance event-driven applications, and the difficulties are discussed especially for the development of GUI applications. Section III presents the proposed programming model, as an extension of OpenMP, and expatiates how this model can help programmers to develop responsive event handlers in an efficient way. In Section IV, an overview of the implementation of the compiler and its runtime is provided. Section V shows the evaluation of this proposed approach. Section VI discusses the related work and Section VII concludes.

## II. Background

This section mainly investigates the background of event-driven programming approaches, and also discusses the difficulties and challenges of developing high performance event-based applications.

### A. Event-driven programming

A wide range of applications are written based on the event-driven programming model, from desktop and mobile applications (apps) to web services. Different from traditional batch-type programs, event-driven applications do not have a predefined runtime execution sequence. For batch-type programs, given input data, the computations are generally executed until completion without requiring further input from the user. Furthermore, the computations performed tend to be rather regular, in that repetitive computations are performed on a vast amount of data (ideal candidate for parallelization). On the contrary, execution of an event-driven application is achieved by an infinite loop (known as the event-loop) with associated event listeners. When a registered event happens, the listener triggers the callback function implemented by programmers. Due to this major difference, "performance" can mean something different to batch-like programs as it does to event-driven programs. Batch-like programs mainly stress on absolute *execution time*, requiring computations are completed as quickly as possible. For event-driven programs, *responsiveness* (perceived performance) of the application's interactivity is a major key factor when evaluating its usability

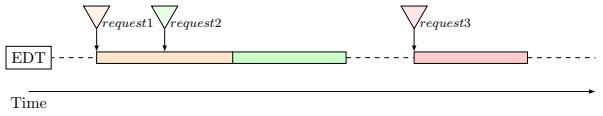
[1]Pyjama is an open-source project which is available at http://parallelit.org

([9], [19], [33], [35]). In Figure 1(i), each triangle represents an event request and the execution of its callback function is represented as a rectangle box with the same color the triangle has. The commencement of *request2* is delayed until the handling of previous events are completed, resulting in an unresponsive application.

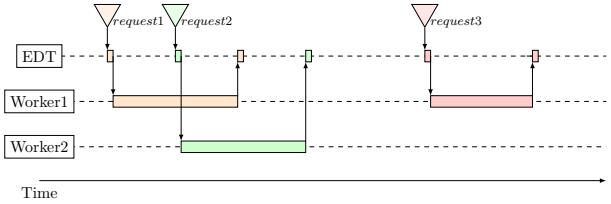
In order to achieve good event-dispatching performance and a better user-experience with regards to responsiveness, various solutions exist. The most traditional approach is known as thread-per-request [28]. In this approach, the time-consuming event handling is directly delegated to a newly-spawned background thread. This allows the EDT to directly exit from the event handler, enabling it to handle another event request, hence achieving the desired responsiveness. The first drawback is of this traditional approach is the heightened software development experience demanded to effectively multi-thread. There is also the salient drawback of non-scalability, since excessively creating threads could decrease the application's performance, as well as the overall system performance due increased scheduling demands and increased overhead associated with thread context switching [29].

Figure 1(ii) shows an improved solution making use of tasking concepts and thread pools, instead of creating a new thread preemptively for every event-handler. This involves submitting the long-running code as a task to an executor bound to a thread pool that limits the maximum number of concurrent threads. The executor manages the thread number, thus reducing the threading overhead and improving overall performance when a large number of tasks need to be executed. While this approach addresses the overhead concerns associated with the threading model, it still demands strong conceptual understanding and experience from software developers to parallelize their applications. The dominant conceptual challenge underpinning these models is that a task submission to an executor means operations depending on the result of the task are not allowed to be executed until the task is finished. While this dependency can be achieved by using a blocking waiting operation until the task is finished, it defeats the initial purpose of introducing concurrency if the waiting thread is the EDT. Therefore, the accepted practice is to bind a completion handler to the task, such that the continuing operations will be executed asynchronously when the task is finished.

In addition to the challenge discussed above, another restriction imposed on programmers is that graphical user interface (GUI) components are not thread-safe and access is strictly confined to the EDT. Inside the event handling code, programmers need to identify and separate code segments to ensure thread-safety. For example, in most GUI application frameworks, updates to the GUI should only be executed by the EDT. Disrespecting this rule could result in the user interface exhibiting inconsistency or even errors [38]. Consequently, this means that if handling code is submitted to a worker executor, the thread context may still need to be switched intermittently to the EDT for operations related to GUI updates. This requires further event posting to the EDT with binding callback functions for the display of intermediate results progress.




(i) Unresponsive single-threaded event processing




(ii) Responsive multi-threaded event processing

Figure 1: In an event-driven application, the EDT plays the role of the main thread responding to events. An essential requirement is to maximize the idleness of the EDT, so programmers are required to transform single-threaded event processing to multi-threaded event processing to increase the responsiveness of the EDT.

### B. Language-related dependencies

Implementing event-driven applications largely depends on the application's language and programming framework. The general aim of the application developer would be to achieve the logic shown in Figure 2. Here, a time-consuming computation is offloaded to the background, while progress updates and final notification still need to be executed by the EDT. Figure 3 and Figure 4 show two specific implementations using Java SwingWorker [26] and C# Asynchronous Programming Model (APM) [21] respectively. Java SwingWorker enables programmers to identify the operations need to be executed as background tasks or foreground updates, by implementing its class interfaces. As a comparison, the programming style of APM is known as Continuation Passing Style (CPS) [5] and all the continuations of the following operations are asynchronously triggered when the previous operations finish. However, the drawback of using CPS (especially for procedural languages) is prominent. The code refactoring required to achieve this functionality requires fragmenting the original callback function, where the fragmented statements are wrapped by auxiliary functions. As a consequence, even though the flow logic of the `ButtonOnClick()` callback function is exactly the same in both implementations, the code structures and API required to achieve this are very different.

This diversity makes it difficult for programmers to write uniform and consistent source code. When porting an application from one platform to another platform, although the programming logic remains the same, the code refactoring requires a great amount of work and programming knowledge.

## III. Programming Model

The motivation of the semantic design proposed in this section is to provide an OpenMP-like directive-based interface to facilitate event-driven programming. The proposal is in line with two principles. First, the directive addition conforms

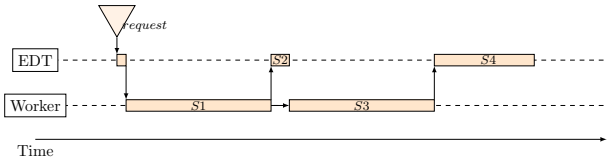



Figure 2: An example of event handling logic, where a time-consuming computation involves background components (S1 and S3), with a foreground progress update (S2), before a concluding foreground computation (S4).

```
void ButtonOnClick() {
   SwingWorker<String, Integer> worker =
   new SwingWorker<String, Integer>(){
      protected String doInBackground(){
         // S1
         publish();
         // S3
      }
      protected void process(List<Integer> updates){
         // S2
      }
      protected void done() {
         // S4
      }
   }
   worker.execute();
}
```

Figure 3: Java asynchronous programming with SwingWorker.

```
public class AsyncWorker{
   public IAsyncResult BeginS1(){
      // S1
   }
   public IAsyncResult BeginS3(){
      // S3
   }
}
void S1CallBack(IAsyncResult result) {
   Dispatcher.BeginInvoke(()=>{
      // S2
      worker.BeginS3(S3CallBack);
   });
}
void S3CallBack(IAsyncResult result) {
   Dispatcher.BeginInvoke(()=>{
      // S4
   });
}
void ButtonOnClick() {
   AsyncWorker worker = new AsyncWorker();
   worker.BeginS1(S1CallBack, result);
}
```

Figure 4: C# AMP-style programming.

with the philosophy of OpenMP, by which the directives can be directly applied on the original sequential version of the code without code restructuring. When the directives are triggered by a supported compiler, the execution benefits from concurrent execution. When the directives are disabled or ignored by unsupported compilers, the code still retains its correctness when executed sequentially. Second, the newly introduced directives are compatible with existing OpenMP directives. With the combination of different directives, programmers are able to express different forms of parallelization and concurrency logic.

### A. Directive Syntax Extensions

The proposed syntax (Figure 5) is inspired by the Accelerator Model introduced to the OpenMP 4.0 specification, namely the `target` directive. The purpose of the `target` directive is to utilize available accelerators in addition to multi-core processors on the system. The `target` directive offloads the computation of its code block to a specified accelerator, if a `device` clause is followed. If the target device is not explicitly specified, the target code block will be submitted to the default accelerator, which is decided by the ICV (Internal Control Variable) *default-device-var*.

**Virtual target.** The original `target` directive can only be validated when the host has accelerators (e.g. GPU), which means a valid *target* must be a physical device. However, our proposed extension of the target syntax introduces the concept of *virtual target*, by which a `target` directive can be followed by a `virtual` clause, instead of a `device` clause. A *virtual target* means the computation is not offloaded to a real physical device. Instead, it is a software-level executor capable of offloading the target block from the thread which encounters this `target` directive. Conventionally, a *device target* has its own memory and data environment, therefore the data mapping and synchronization are necessary between the host and the target. That is why normally some auxiliary constructs or directives such as `target data` and `target update` are used when using `target` directives. In contrast, a *virtual target* actually shares the same memory as the host holds, so the data context remains the same when entering the target code block. Generally, a *virtual target* is a syntax-level abstraction of a thread pool executor, such that the `target` block is executed by the executor specified by the *target-name*.

**Target block scheduling.** By default, an encountering thread may not proceed past the target code block until it is finished by either the device target or virtual target. However, a more flexible and expressive control flow of the encountering thread can be achieved by adopting the *scheduling-property-clause*. The consideration behind this is, a target block can also be regarded as a task with an asynchronous nature. Section III-C will specifically explain the different scheduling clauses that influence the processing of the program.

```
void buttonOnClick() {
   Panel.showMsg("Started EDT handling");
   Info info = Panel.collectInput();
   //#omp target virtual(worker) nowait
   {
      int hscode = getHashCode(info);
      downloadAndCompute(hscode);
      //#omp target virtual(edt)
      Panel.showMsg("Finished!");
   }
}
void downloadAndCompute(int hs) {
   Buffer buf = networkDownload(hs);
   Image img = formatConvert(buf);
   //#omp target virtual(edt)
   Panel.displayImg(img);
}
```

Figure 6: Semantic example of using virtual target directive.

**#pragma omp target** *[clause[[,]clause]...]*
*structured-block*

clause:
*target-property-clause*
*scheduling-property-clause*
*data-handling-clause*
*if-clause*

where *target-property-clause* is one of the following:
**device(***device-number***) virtual(***name-tag***)**

where *scheduling-property-clause* is one of the following:
**nowait name_as(***name-tag***) await**

Figure 5: Extended target directive.

### B. Semantic Model

Since our implementation is based on Java, and Java does not support pragma conditional compilation, the directive begins with `//#omp`. It means that compilers that do not support the semantics will safely ignore the directives by regarding them as comments. On the contrary, a supporting compiler will interpret the directives and compile the code as a parallelized version.

We demonstrate the usage of `target virtual` directives by showing a piece of pseudo code of an event handler implementation. In Figure 6, when a button is clicked, the callback function `buttonOnClick()` is triggered. Firstly the function updates a message to GUI to indicate the start of the processing. Then a series of time-consuming operations are processed according to the inputs from the GUI. In this example the operations involve downloading a file from the network and then performing image processing on the downloaded raw data. Afterward, the image is rendered to the GUI and a finished message is updated.

If the directives are ignored, the entire code will be executed by the thread which invoked the callback function, i.e. the EDT. For a compliant compiler, the entire callback function will be executed by the cooperation of two virtual target executors (`edt`, `worker`). Under this circumstance, the handling time of the EDT decreases because the EDT only spends time on the operations which should be necessarily executed by the EDT. Other operations are smartly offloaded to the worker executor, without breaking the original code structure and logic. The benefit of using virtual target semantics involves four key aspects:

**Thread-context awareness.** A code block guarded by a specified `target virtual` directive shows its preference of execution by a specified type of thread, or executor. If the encountering thread has the same property as the virtual target specified, the `target virtual` directive is simply ignored. Otherwise, the directive compels the encountering thread to relinquish control of the code block and do a runtime thread-context switch to the specified target. The thread-context awareness property of the `target virtual` construct smartly confines the authorization of the code block execution to specified type of thread. For example, for GUI applications, a GUI update code block guarded by a `target virtual(edt)` will ensure that all the operations related the GUI are executed by the event dispatching thread.

**Execution offloading.** Delegating code to another virtual target offloads work from the current thread, therefore alleviating the computational burden from the encountering thread. For a function invocation, if some parts of the function are delegated to other virtual targets, the actual execution time for the thread which invoked the function will be decreased. This aspect is extremely important in the scenario of event dispatching. Work offloading enables the EDT to spend less time on the event handler, allowing it to dispatch more events in the application.

**Data-context sharing.** Using a standard `target` directive means offloading the code to an actual hardware accelerator, therefore the data transferring and data synchronization is necessary. Instead, using a virtual target means code is offloaded to a software-level executor. Since all virtual targets share the same memory, there is no need to copy data from main memory to the accelerators memory. This simplifies usage of the `target virtual` directive, since it is not necessary to do heavy data copying or even variable passing when using a virtual target switch (if the OpenMP `default(shared)` data clause is specified). All the operations inside a target block share the intuitive data context as if the target directive does not exist.

**Intuitive continuation-passing.** Adding `target virtual` directives modifies the source code from a sequential version to a parallel and asynchronous version, while still maintaining clean programming logic. The end of a target block is intuitively followed by operations which depend on it. Although a target block has the nature of asynchronous execution when the operations following it should not be executed until the target block is completed, the continuation of the target block does not require any code refactoring for a completion-event callback function binding. Since the continuation logic is still represented in the sequential code, it dramatically reduces the work of code refactoring to achieve asynchronization and parallelization.

### C. Asynchronous Execution Model

The purpose of a `target virtual(worker)` directive is to offload work from the current thread to a virtual target

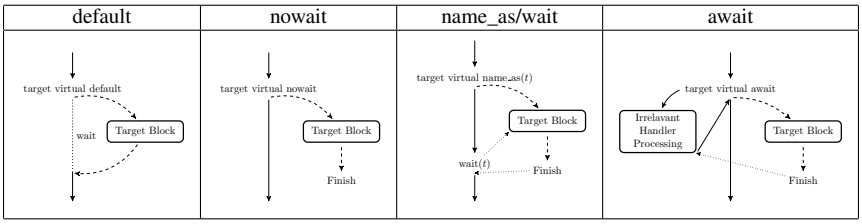



Table I: Different asynchronous modes, by using different scheduling-property-clauses.

executor. If the current thread cannot proceed during execution of the target block, and simply halts its execution, there is no actual performance advantage from the target block offloading. Therefore, instead of busy waiting, an asynchronous execution is applied for the target block. The asynchronous execution can be categorized into the three types illustrated in Table I, by using different modes for the *scheduling-property-clause*:

- Default. If there is no *async-property-clause* specified, then no asynchronous execution occurs. The encountering thread will busy-wait until the target code block is finished by the specified target. If the executing time of the target block is noticeably long, this is not the ideal approach because the encountering thread cannot do anything useful during the period. Also, in the case of event handling threads, this causes an unresponsive application. However, choosing to wait corresponds to the standard OpenMP behavior of the `target` directive.
- `nowait`. The encountering thread directly skips the target block and leaves the target block as an asynchronous task, then continues executing statements following the block. There is no notification when the task is finished. The `nowait` clause is usually used when there is no further operations which depend on the result of the asynchronous task. Therefore, the code block can be safely invoked and ignored. This is useful for broadcasting interim updates, where the broadcasting thread does not need to wait for a response from listeners.
- `name_as/wait`. The encountering thread directly skips the target block and leaves the target block as an asynchronous task, then continues executing statements following the block. Unlike `nowait`, a task identifier *name-tag* is created that enables the encountering thread to explicitly synchronize with the task by using the associated `wait(`*name-tag*`)` clause later in the code. Notice that different target blocks are allowed to share the same *name-tag*, such that when an `wait` clause is applied with that *name-tag*, the encountering thread suspends until all the *name-tag* asynchronous target block instances finish.
- `await`. The `await` asynchronization policy is a wait policy, with the important modification that during the wait period other events are processed by the event-loop. While the target block is being executed by the respective virtual target, the encountering thread returns to the event loop in search of another event to process. When the target code block has completed, the code following it is continued. The code dependency is naturally represented by the code sequence, for which the continuation of an asynchronous execution is intuitive and there is no need for an explicit completion event handler binding.

### D. Parallel Execution Model

A virtual target is essentially a thread pool executor, and its lifecycle lasts throughout the program. Conceptually, a virtual target represents a type of execution environment defining its thread affiliation (to ensure operations not thread-safe are only executed by a specified thread), and scale (confine the number of threads of a thread pool). This design enables programmers to flexibly submit different code snippets to different execution environments.

In the current experimental version of Pyjama, there are two types of virtual targets that can be created. For example, for GUI applications, the master thread is the EDT. At the initializing stage (e.g. the constructor of the graphical user interface) of the application, in order to inform the compiler of the EDT virtual target and worker virtual target, the runtime functions of Table II are required to be invoked with specific parameters.

## IV. Implementation Overview

This section discusses the implementation of the proposed programming model in Pyjama, which is an OpenMP-like implementation for Java. Pyjama mainly constitutes two parts. First, the source-to-source compiler, which supports traditional OpenMP directives and the extended directive proposed here, transforming the sequential Java source code into parallel code. Second, the runtime system, which provides underlining thread-pool creation, management and the task scheduling, as well as all the OpenMP runtime functions.

### A. Compilation

The Pyjama compiler does a source-to-source translation, and all the code blocks annotated with OpenMP directives will be transformed and refactored for the purpose of parallelization. Here, we provide an overview of how a target block is converted to the destination code. Consider the following code snippet that uses the extended target model:

```
Label.setText("Start Processing Task!");
//#omp target virtual(worker) await
{
    compute_half1(); // S1
    //#omp target virtual(edt) nowait
    {
        Label.setText("Task half finished"); // S2
    }
```

| Name | Parameters | Description |
|---|---|---|
| `virtual_target_register_edt()` | `tname`:String | The thread which invokes this function will be registered as a virtual target named `tname`. |
| `virtual_target_create_worker()` | `tname`:String, `n`:Integer | Creating a worker virtual target with maximum of `n` threads, and its name is `tname`. |

Table II: Runtime functions for supporting asynchronous parallelism model.

```
   compute_half2(); // S3
}
Label.setText("Task finished"); // S4
```

The compiler will restructure a target block as a runnable TargetRegion class, with its run() function implementing the user code. The data context and variables referenced by the user code are passed into the generated class (for simplicity, the demo code omits the details). The target region instance is then submitted to the Pyjama runtime, which is responsible for dispatching the target code block to the appropriate virtual target.

```
class TargetRegion_0() implements Runnable {
  public void run() {
    compute_half1(); // S1
    TargetRegion _omp_tr_1 = new TargetRegion_1();
    PjRuntime.invokeTargetBlock("edt", _omp_tr_1);
// S2
    compute_half2(); // S3
  }
}
Label.setText("Start Processing Task!"); // S4
TargetRegion _omp_tr_0 = new TargetRegion_0();
PjRuntime.invokeTargetBlock(
          "worker", _omp_tr_0, Async.await);
Label.setText("Task finished");
```

### B. Runtime

During execution of the program, target blocks are dynamically dispatched by the Pyjama runtime. The logic of invoking a target block is presented in Algorithm 1. Firstly, the runtime routine checks if the submitting thread is already a member of the virtual target executor's thread group (line 6). If yes, it means the target block is already in the context of the virtual target execution environment, so it is executed synchronously by the current thread (line 7). If the *async-property-clause* is `nowait` or `name_as` (line 10), the main thread exits the procedure (line 11) to directly execute the statements following the target block. Otherwise, it waits for the target block to finish (line 14 or 17).

**Processing other event handlers.** For the `await` asynchronous execution of the target blocks, the current thread cannot execute the statements following the target block until the asynchronous target block finishes. In order to avoid a blocking of the current thread (and therefore resulting in an unresponsive application if current thread is EDT), the implementation adopts a "logical barrier" (line 14). This enables the thread to process other event handlers or tasks in the system (line 15), which may be irrelevant to the logic of the current handling code. As for the worker virtual target, it is achieved by processing another runnable task in Pyjama's task queue. With regard to the GUI EDT, as a proof-of-concept implementation, the current experimental version of Pyjama achieves this by slightly modifying the event queue dispatching mechanism in the Java AWT runtime library.

**Algorithm 1** Target block code execution.

1: $\mathcal{T}$: current thread
2: $\mathbf{E}$: target executor
3: $\mathcal{B}$: target block
4: $\mathbf{a}$: asynchronous property
5: **procedure** INVOKETARGETBLOCK($\mathcal{T}$,$\mathbf{E}$,$\mathcal{B}$,$\mathbf{a}$)
6: **if** $\mathcal{T} \in \mathbf{E}$ **then**
7: $\mathcal{B}$.exec() ▷ execute $\mathcal{B}$ synchronously by $\mathcal{T}$
8: **else** $\mathbf{E}$.post($\mathcal{B}$) ▷ post $\mathcal{B}$ to $\mathbf{E}$ asynchronously
9: **end if**
10: **if** $\mathbf{a}$ is *nowait* or *name_as* **then**
11: return
12: **end if**
13: **if** $\mathbf{a}$ is *await* **then**
14: **while** $\mathcal{B}$ is not finished **do** ▷ logical barrier
15: $\mathcal{T}$.processAnotherEventHandler()
16: **end while**
17: **else** $\mathcal{T}$.wait() ▷ default option
18: **end if**
19: **end procedure**

## V. Evaluation

This section provides a performance evaluations of the proposed approach for event-driven programming. Two types of case studies are presented.

### A. Java GUI event handling

Modern real-world applications/apps usually require a high computational ability without losing any responsiveness. For example, consider a mobile visual-realism application constantly capturing images from the camera and then applying the image rendering or processing (e.g. augmented reality) for the user. In order to achieve a smooth user experience, the processing of each frame should be as short as possible, especially when many images are captured in a short period. Here, scenarios are simulated in which a GUI application is under different loads of event handling, and the benchmarks measure the ability of handling events by different approaches.

The first evaluation compares the different methods for offloading time-consuming work to the background, while maintaining a responsive GUI. Since the benchmarks are performed under the Java Swing GUI framework, three different approaches are compared: SwingWorker, ExecutorService (using SwingUtilities when necessary) and Pyjama. Each benchmark adopts a computational kernel selected from the Java Grande Benchmark suite [34] (since the kernel can be parallelized by using traditional OpenMP directives), to simulate the time-consuming computational work within event handlers. Selected were Crypt, RayTracer, MonteCarlo and Series. There are GUI updates before and after the kernel

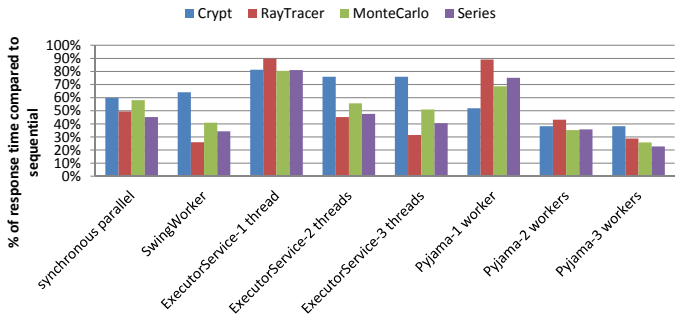



Figure 7: Average event response time, as a proportion of the sequential version, using different offloading approaches and computational kernels (lower is better).

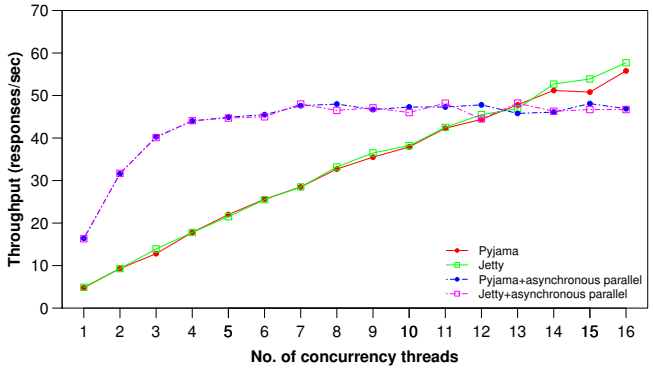



Figure 9: Throughput scaling comparison between Jetty and Pyjama.

execution. As discussed before, those GUI related operations are required to be executed in the EDT. As the application utilizes a GUI component, the benchmarks are performed on a typical desktop machine (in this case an i5-3570 quad-core Intel processor, with 8M cache, up to 3.90 GHz clock rate). Oracle's Java 1.8.0_66 VM is used throughout the benchmarks.

The benchmarks are categorized by the kernels. For each benchmark, the event is bound with an execution of its kernel. Every benchmark is run 10 rounds with different request loads, ranging from 10 requests/sec to 100 requests/sec. The response time shows the time flow from the event firing to the finish of its event handling. The average response time of all events shows a general efficiency of processing of event handling. To show how different approaches decrease the average response time, compared to the sequential version, different offloading approaches are presented. We also show the synchronous parallel version (in default using 3 worker threads), in which only the computational kernels are parallelized and the EDT still does part of the computing job when handling the events. Therefore, the EDT in the synchronous parallel approach is actually unresponsive for a longer time compared to other approaches. The underlying implementation of SwingWorker maintains a default 10-thread-max thread pool. Figure 7 depicts the results, showing the average response times in proportion to sequential versions. The results show that Pyjama has a comparable (or in some cases better) event response time compared to the other manual approaches, especially when three worker threads execute the kernels in the background. It is also interesting to observe that the execution of kernels in parallel (but synchronously) is inferior to an asynchronous execution with the same number of threads when comparing the response times.

In a GUI application, if responding to an event trigger exceeds 5 seconds, the application is deemed unusable [35]. Using this rule of thumb, Figure 8 counts the number of event responses that complete within 5 seconds, depending on different event request loads. Event requests are kept consistent for each sequential version, since it reaches the maximum handling ability of single-threaded sequential versions. SwingWorker shows inconsistent performance as the request load differs, which may be attributed to its underlying scheduling policy of its thread pool tasks. The ExecutorService shows a performance degradation when more events happen in the same time unit. It may be attributed to the accumulated overhead by task submissions of the underlying implementation of ExecutorService. In contrast, Pyjama's virtual target offloading keeps a consistent and high response rate. This shows that the implementation of Pyjama's runtime is more suitable for offloading more tasks under the scenario of heavy workloads of event handling.

### B. *Web Service event handler*

The purpose of this benchmark is to evaluate the scalability of Pyjama's virtual targets runtime for a different type of event. We implement an HTTP service that provides data encryption to web users. Every time a user sends input data with an HTTP request, the server performs a calculation and returns the result via the HTTP response. The encryption computation can be parallelized by adopting traditional OpenMP directives. The web server is implemented using two approaches. The first uses Pyjama's virtual target to offload the time-consuming computations to worker threads. The second uses Jetty's [2] thread-pool framework, which adopts a thread-per-request policy but reuses a fixed number of threads from a thread pool. This experiment is run on a 16-core Intel Xeon 2.4GHz SMP machine with 64 GB memory, and Java 1.8.0_66 HotSpot 64-Bit Server VM.

The load benchmark is set up with 100 virtual users, with each user sending a constant number of requests. The throughput measures the application's ability to process requests. Figure 9 describes that both Jetty and Pyjama have good scaling performance as the number of concurrency worker threads increases. When the parallelization of each event (using //#omp parallel) is used in combination with either Jetty or Pyjama, it initially results in dramatically better throughput. Yet, as the number of concurrency worker threads is increased, the throughput levels off at just under 50 responses/sec. The non-parallelized versions achieve better throughput when the number of concurrency workers gets above 13. This result is reasonable, because every parallelization computation spawns its own set of worker threads. With the increased amount of computation requests, the total number of threads in the system

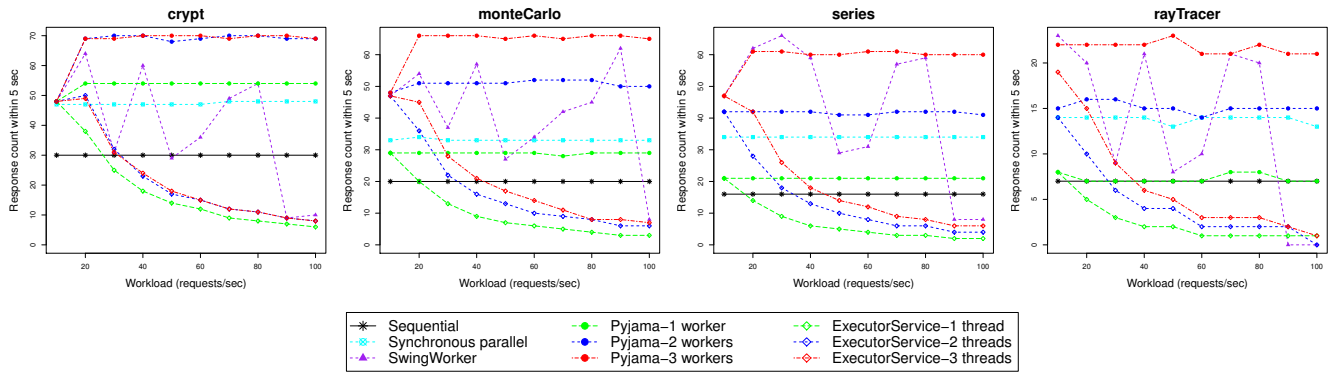



Figure 8: 5-second response count under different request work loads (higher is better).

soars to a high value and it leads to a great overhead of thread scheduling.

## VI. Related Work

### A. Asynchronization

Asynchronous programming is traditionally used in single-threaded applications to achieve cooperative multitasking [10]. Unlike parallel programming that creates multiple threads, this programming model employs a single background thread. As such, the purpose of introducing asynchronization is not to make the program run faster. Instead, it is used when an event handling thread needs to wait for time-consuming computations or I/O. In this manner, the thread can still progress since the control flow is switched to another task.

**Libraries.** Many languages provide build-in or extended library interfaces to support asynchronous programming. For example, C++11 provides std::async, while Java provides the Future interface [25] building asynchronous computations. Java NIO libraries [27] provide non-blocking and asynchronous I/O operations. Microsoft .NET provides three types of asynchronous programming patterns [22]: (1) Asynchronous Programming Model (APM); (2) Event-based Asynchronous Pattern (EAP); (3) Task-based Asynchronous Pattern (TAP).

**Frameworks.** The implementation of an asynchronous task usually applies an event-driven programming pattern, of which the continuation of the task is transformed as a callback function which will be triggered when asynchronous operations finish. This idea has been adopted to many different languages and frameworks, especially for the sake of high-performance network server developing. For example, libevent [20] is an asynchronous event-based network application framework written in C, adopting proactor pattern [30], which is an object behavioral pattern of the combination of I/O multiplexing [32] and asynchronous event dispatching. Similarly, other frameworks written in other languages (e.g. [3], [1], [4]) become increasingly popular in recently years.

**Languages support.** Unlike libraries, language-level support for asynchronization tends to require less code restructuring. Fischer et al. [12] proposed TaskJava, a backward-compatible extension to Java. By introducing new keywords (i.e. spawn, async, wait), TaskJava expresses the complicated asynchronous logic control flow using intuitive sequential programming style. Similarly, the .NET framework also introduces paired async/await keywords [23]. New language designs also tend to support asynchronization. For example, P [8] is a domain-specific language for the modeling of state machines, and all machines communicate via asynchronous events. Eve [13] is another parallel event-oriented language for the development of high-performance I/O applications. Other language-level concepts such as the actor model [17], [16] and co-routines [7] provide variations to asynchronization.

### B. Task-based Parallelism

The task-based parallelization model is usually implemented to overcome the performance issues of the threading model. A fixed thread pool substitutes preemptive thread-creation when a computational task is needed. The thread pool technique encapsulates the underlying threading and scheduling [31] and provides interfaces for task submissions. Some languages support tasks at a language level, such as Cilk [14] and JCilk [6]. While OpenMP provides the `task` directive [24], the lifetime of a task is confined inside a parallel region. In addition to the actual parallelization, handling task dependencies and code restructuring is another challenge faced. Parallel Task [15], as a language extension of Java, supports task creation and dependency handling. OoOJava [18] and DOJ [37] both introduce the task keyword to achieve out-of-order execution of the code blocks, with the support of automatic dependency analysis between tasks.

## VII. Conclusion

This paper proposed a hybrid model for the combination of asynchronization and parallelization, as an extension of OpenMP. The idea is implemented in Pyjama, an OpenMP compiler and runtime support for Java. The model facilitates the development of event-driven programs, especially for GUI applications, to achieve better responsiveness and event handling acceleration. Strictly following the philosophy of OpenMP, the semantic design of this model does not interfere with the original sequential programming logic. With the help of a supporting compiler, the additional directives

generate event handling code to execute asynchronously and offload computations away from the event dispatching thread. Evaluations show that single-threaded event dispatching can be quickly upgraded to a higher performing multi-threaded event dispatching, by reducing event handling response time. Performance achieved by the proposed directive based approach is equal and often superior to manual implementations. A further extension of this work involves a more universal implementation to support more event-driven frameworks and integrating non-blocking I/O and asynchronous I/O into this model.

## REFERENCES


[1] Grizzly project. Available at: https://grizzly.java.net/, February 2016.

[2] Jetty project. Available at: https://eclipse.org/jetty/, February 2016.

[3] Netty: an asynchronous event-driven network application framework. Available at: http://netty.io, February 2016.

[4] Nodejs. Available at: https://nodejs.org/, February 2016.

[5] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[6] John S. Danaher, I.-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147 – 171, 2006. Special issue on synchronization and concurrency in object-oriented languages.

[7] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[8] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[9] David Duis and Jeff Johnson. Improving user-interface responsiveness despite performance limitations. In *Compcon Spring'90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 380–386. IEEE, 1990.

[10] Ralf S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[11] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997.

[12] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 134–143. ACM, 2007.

[13] Alcides Fonseca, João Rafael, and Bruno Cabral. Eve: A parallel event-driven programming language. In *Euro-Par 2014: Parallel Processing Workshops*, pages 170–181. Springer, 2014.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[15] N. Giacaman and O. Sinnen. Task parallelism for object oriented programs. In *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on*, pages 13–18, May 2008.

[16] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[17] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[18] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. Ooojava: Software out-of-order execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 57–68, New York, NY, USA, 2011. ACM.

[19] Milan Jovic and Matthias Hauswirth. Listener latency profiling: Measuring the perceptible performance of interactive java applications. *Science of Computer Programming*, 76(11):1054–1072, 2011.

[20] Nick Mathewson and Niels Provos. libevent. Available at: http://libevent.org, February 2016.

[21] Microsoft. Asynchronous Programming Model. Available at: https://msdn.microsoft.com/en-us/library/ms228963%28v=vs.110%29.aspx, February 2016.

[22] Microsoft. Asynchronous Programming Patterns. Available at: https://msdn.microsoft.com/en-us/library/jj152938%28v=vs.110%29.aspx, February 2016.

[23] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1117–1127. ACM, 2014.

[24] OpenMP Architecture Review Board. OpenMP application program interface 4.5, November 2015.

[25] Oracle. Java 7 future interface. Available at: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html, February 2016.

[26] Oracle. Java 7 swingworker. Available at: http://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html, February 2016.

[27] Oracle. Java I/O, NIO, and NIO.2. Available at: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html, February 2016.

[28] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference, General Track*, pages 199–212, 1999.

[29] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R Cheriton. Comparing the performance of web server architectures. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 231–243. ACM, 2007.

[30] Irfan Pyarali, Tim Harrison, Douglas C Schmidt, and Thomas D Jordan. Proactor-an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. 1997.

[31] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM.

[32] Douglas C Schmidt. Reactor–an object behavioral pattern for demultiplexing and dispatching handlers for synchronous events. 1995.

[33] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR)*, 16(3):265–285, 1984.

[34] L.A. Smith, J.M. Bull, and J. Obdrizalek. A parallel java grande benchmark suite. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 6–6, Nov 2001.

[35] Niraj Tolia, David G Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer*, 39(3):46–52, 2006.

[36] Vikas, Nasser Giacaman, and Oliver Sinnen. Pyjama: OpenMP-like implementation for java, with gui extensions. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 43–52, New York, NY, USA, 2013. ACM.

[37] Stephen Yang, James C Jenista, Brian Demsky, et al. Doj: Dynamically parallelizing object-oriented programs. In *ACM SIGPLAN Notices*, volume 47, pages 85–96. ACM, 2012.

[38] Sai Zhang, Hao Lü, and Michael D Ernst. Finding errors in multithreaded gui applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 243–253. ACM, 2012.