Supporting Asynchronization in OpenMP for Event-Driven Programming

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

1 Introduction

OpenMP is the de facto standard 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* [13], 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 always 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 [27]. 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 pre-defined, 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 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 demand 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 applica-

tions. 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 open-source tool for OpenMP-like directives for Java [43]. 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 2 reviews the background of developing high-performance event-driven applications, and the difficulties are discussed especially for the development of GUI applications. Section 3 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 4, a discussion of the implementation of the compiler and its runtime is provided. Section 5 shows the evaluation of this proposed approach. Section 6 discusses the related work and Section 7 concludes.

2 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.

2.1 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

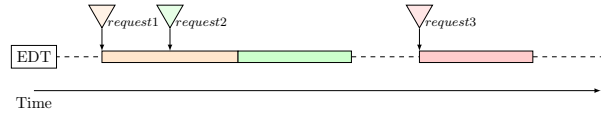
[1]Pyjama is an open-source project which is available at `http://parallelit.org`

performance) of the application's interactivity is a major key factor when evaluating its usability ([10, 22, 40, 42]). 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.

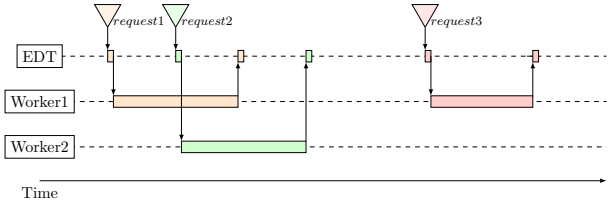
In order to achieve good event-dispatching performance and a better user-experience with regards to responsiveness, various solutions exist. The most traditional approach is known as thread-per-request [34]. In this approach, the time-consuming event handling is directly delegated to a newly-spawned 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 [35].

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 [45]. 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.

2.2 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 [29] and C# Asynchronous Programming Model (APM) [24] 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.

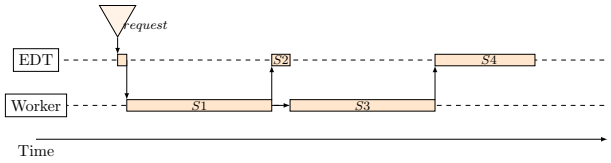


Figure 2: An example of event handling logic, where a time-consuming computation involves background components (S1 and S3), with a foreground progress update (S2), before a concluding foreground computation (S4).

```
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.

2.3 Notation Representation

This section formally defines the restructuring required to achieve concurrency in an event-driven application. The basic components are firstly defined as follows:

- $e \rightarrow \mathcal{F}$ is defined as an event handler binding in which every time event e occurs, the callback function $\mathcal{F}$ is to be invoked.
- $\mathcal{F}(\mathcal{T})$ specifies that the function call is invoked by thread T.
- $\mathcal{F} ::= \{S_1, S_2, S_3, ..., S_n\}$ represents the expansion of function $\mathcal{F}$ to represent a total of n statements in that function.
- Each statement S can be a primitive statement P or another function call $\mathcal{F}$, namely $S ::= P|\mathcal{F}$.
- $S(\mathcal{T})$ indicates that statement S needs to be executed by thread $\mathcal{T}$.

If there is no concurrency expressed within a function call $\mathcal{F}(\mathcal{T})$, all the statements inside $\mathcal{F}$ are executed by thread $\mathcal{T}$. In this regards, the total execution time of this function is the sum of the execution of each statement: $t(\mathcal{F}(\mathcal{T})) = t(S_1(\mathcal{T})) + t(S_2(\mathcal{T})) + t(S_3(\mathcal{T})) + ... + t(S_n(\mathcal{T}))$. If the invoking thread $\mathcal{T}$ is the EDT, denoted as $\mathcal{T}_{edt}$, this means the application is unresponsive for the period of $t(\mathcal{F}(\mathcal{T}_{edt}))$. As the EDT is executing $\mathcal{F}$, it cannot respond to other events or requests during this time.

In regards to refactoring and asynchronization:

- $\mathcal{F} \Longrightarrow_r \mathcal{F}'$ is defined as a code refactoring of function $\mathcal{F}$.
- $\mathcal{F}|_{\mathcal{A}}$ is defined as an asynchronous execution of function $\mathcal{F}$.

For the approach of preemptive multi-threading or task submission, for each $e \rightarrow \mathcal{F}(\mathcal{T}_{edt})$ there is a refactoring $\mathcal{F}(\mathcal{T}_{edt}) ::= \{S_1, S_2, S_3, ..., S_n\} \Longrightarrow_r \mathcal{F}'(\mathcal{T}_b)\ |_{\mathcal{A}}$ such that $\mathcal{F}'(\mathcal{T}_b) ::= \{S_1, S_2, S_3, ..., S_n\}$. Here, $\mathcal{T}_b$ is a background thread that has been delegated all statements of the original function. Ideally the handling time of the EDT is decreased to zero ($t(\mathcal{F}(\mathcal{T}_{edt})) = 0$), allowing the EDT more free time to handle other events. However, the refactoring is typically more complicated as some statements are thread-affiliated. For example, the formal definition of Figure 2's callback function can be represented as $\mathcal{F}(\mathcal{T}_{edt}) ::= \{S_1, S_2(\mathcal{T}_{edt}), S_3, S_4(\mathcal{T}_{edt})\}$, where statements S_2 and S_4 must only be executed by the EDT. For this situation, a correct and efficient computing offloading of S_1 and S_3 could lead to a great amount of code refactoring and programming effort:

$$\mathcal{F} ::= \{S_1, S_2(\mathcal{T}_{edt}), S_3, S_4(\mathcal{T}_{edt})\} \Longrightarrow_r$$

$$\mathcal{F}' ::= \{e_{complete}(\mathcal{F}_1(\mathcal{T}_b)) \rightarrow \mathcal{F}_2(\mathcal{T}_{edt}), \mathcal{F}_1(\mathcal{T}_b)|_{\mathcal{A}}\ \}, \quad (p\,1)$$

$$\mathcal{F}_2 ::= \{S_2, e_{complete}(\mathcal{F}_3(\mathcal{T}_b)) \rightarrow \mathcal{F}_4(\mathcal{T}_{edt}), \mathcal{F}_3(\mathcal{T}_b)|_{\mathcal{A}}\}, \quad (p\,2)$$

$$\mathcal{F}_1 ::= \{S_1\}, \mathcal{F}_3 ::= \{S_3\}, \mathcal{F}_4 ::= \{S_4\} \quad (p\,3)$$

Firstly, the original callback function above is refactored such that S_1 becomes an asynchronous call by wrapping it with a newly created auxiliary function $\mathcal{F}_1$, executed by $\mathcal{T}_b$. The completion of S_1 is then bound with the asynchronous invocation of S_2, which need to be executed by the EDT (*p1*). Similarly, after finishing S_2, the completion of $\mathcal{F}_3$ is bound to the invocation of $\mathcal{F}_4$, while executing $\mathcal{F}_3$ asynchronously (*p2*).

This approach successfully decreases the handling time of the EDT to $t(\mathcal{F}(\mathcal{T}_{edt})) = t(S_2(\mathcal{T}_{edt})) + t(S_4(\mathcal{T}_{edt}))$. However, the code refactoring required to achieve this functionality requires fragmenting the original callback function, where the fragmented statements are wrapped by auxiliary functions ($\mathcal{F}_1 \tilde{} \mathcal{F}_4$).

This example illustrates that a high-performance event-driven handler implementation is difficult, especially with the combination of parallelization and asynchronization. Besides the creation of thread pool executors and submitting tasks to the appropriate executors, excessive callback function binding, nesting and efforts for the thread executing switching make the logic of the control flow obscure. This traditional approach makes it difficult to get clean and maintainable code for multi-threaded event-based programs. Another salient drawback of using callback function listening is its increased debugging complexity.

3 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 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.

3.1 Directive Syntax Extensions

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

clause:
target-property-clause
asynchronous-property-clause
data-handling-clause
if-clause

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

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

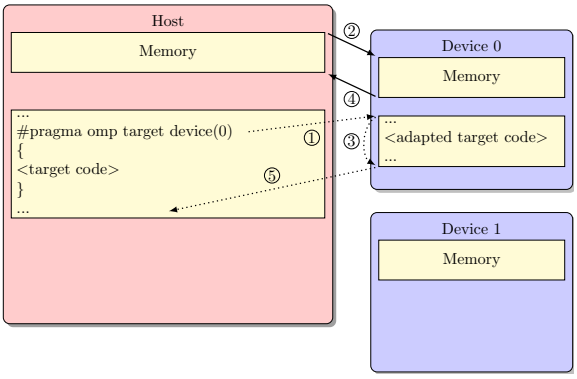
where *data-handling-clause* is one of the following:
firstprivate(*list***) shared(***list***)**

and *if-clause* is:
if(*scalar-expression***)**

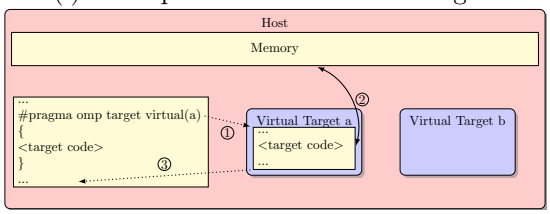
Figure 5: Extended target directive.

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



(i) Conceptual model of device target.



(ii) Conceptual model of virtual target.

Figure 6: Conceptual difference between virtual target and device target.

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. Figure 6 shows the conceptual differences between the device target and virtual target. 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 *asynchronous-property-clause*. The consideration behind this is, a target block can also be regarded as a task with an asynchronous nature. Section 3.2.2 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 7: Semantic example of using virtual target directive.

3.2 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.

3.2.1 Semantics of target offloading

We demonstrate the usage of `target virtual` directives by showing a piece of pseudo code of an event handler implementation. In Figure 7, 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`). In this situation, 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 to 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 an asynchronous (and possibly parallel) 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.

3.2.2 Semantics of asynchronous execution

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

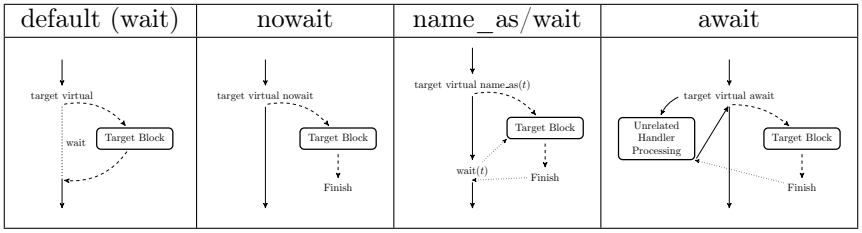


Figure 8: Different asynchronous modes, by using different *asynchronous-property-clause*s.

illustrated in Figure 8, by using different modes for the *asynchronous-property-clause*:

- Default (wait). If no *asynchronous-property-clause* is 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 this time. Also, in the case of event handling threads, this results in an unresponsive application. However, this 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 the `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 difference that during the wait period the control flow jumps out of the current function and back to its caller. When the target code is finished, the function resumes its execution from where it previously suspended. Conceptually, the purpose of using `await` is, while the target

block is being executed by the respective virtual target, the encountering thread is able to return to the event loop (to process other events) or to the task pool (to process other tasks); this is known as Unrelated Handler Processing. This has the advantage of keeping the encountering thread active by processing other meaningful workload instead of blocking or busy waiting. Also, if the encountering thread is the EDT, it enables the EDT to process more events during this time. The code dependency is also naturally represented since the original code sequence is preserved for each event handler; the continuation of an asynchronous execution is intuitive without any need for explicitly binding completion event handlers. Since nested function calls are possible within an event handler, functions may recursively suspend their executions due to an innermost await target block. A more detailed explanation of this situation is discussed in Section 3.2.3.

3.2.3 Semantics of using await and async function

For most cases, the await asynchronization policy is the most convenient way for programmers to express the dependencies between code. Unlike `wait`, which requires programmers to manually guarantee the completion of its corresponding `name_as` target code blocks, the code followed by an await target block will be automatically scheduled by the runtime when this await target block finishes.

The await property of a target block changes the behavior of the control flow in the current function, since the function yields its execution to the caller when the target block is being executed. As a consequence, the function containing await has a resumable (asynchronous) nature itself, and is therefore known as an *async function*. Calling an async function is different from calling an synchronous function, because asynchronous functions "return immediately" to the caller before the function's computation is actually completed (due to the nature of asynchronous execution). Therefore, the caller selects the call type, denoting whether it should (i) invoke the function asynchronously (and therefore continue on the statement following the function call without waiting for the completion of the asynchronous function), or (ii) also yield its execution and the control flow back to the caller's caller and resuming only when the asynchronous function completes. Recursively, a function invoking an async function using the latter approach is endowed with a resumable(asynchronous) nature also.

#**pragma omp async-call** *asynchronous-property-clause*(*function-declaration [[,]function-declaration]...*)
structured-block

where *asynchronous-property-clause* is one of the following:

nowait name_as(*name-tag*) **await**

Figure 9: Definition of async-call directive, for the purpose of calling async function in different ways.

In order to distinguish the two types of ways to invoke an async function,

```
void int bar1(Info info) {
    before();
    a = foo(1) + foo(2);
    after();
}

//#omp async
void int bar2(Info info) {
    before();
    //#omp async-call await(int foo(int))
    {
        a = foo(1) + foo(2);
    }
    after();
}
//#omp async
int foo(int work) {
    //#omp target virtual(worker) await
    {
        cpu_bound_computation(work);
    }
}
```

Figure 10: Semantic example of using await directive.

the async-call construct (Figure 9) is introduced. The directive is followed by an *asynchronous-property-clause*, then a list of function declarations, and then a code block. All the function calls in the code block that are declared in the function list, will be invoked according to the *asynchronous-property-clause*. If nowait or name_as is applied, the function is invoked asynchronously and the caller continues with the statements following the function call. The wait(*name-tag*) directive is able to explicitly wait for the completion of its paired name_as(*name-tag*) asynchronous function call. If await is applied, the caller yields its execution until the declared async function is completed, and then the the caller gets the return value from the function and the execution continues.

Figure 10 shows an example of the use of the async-call await construct. In this example, the function foo() is marked as an async function because inside this function, a target virtual await block is used. In function bar1(), the foo() is invoked in a normal way, which means function bar1() synchronously invokes foo() functions and get the return values. As a comparison, the function bar2() invokes function foo() inside an await construct, which causes the suspension of bar2() when waiting the return from foo() invocations. At the same time, since bar2() is endowed as a resumable(asynchronous) nature, when using await construct, function bar2() is also marked as an async function.

Formally, we define a function as an *async function* when:

1. There are one or more omp target virtual await constructs inside this function; and/or

2. There are one or more omp async-call await constructs inside the function.

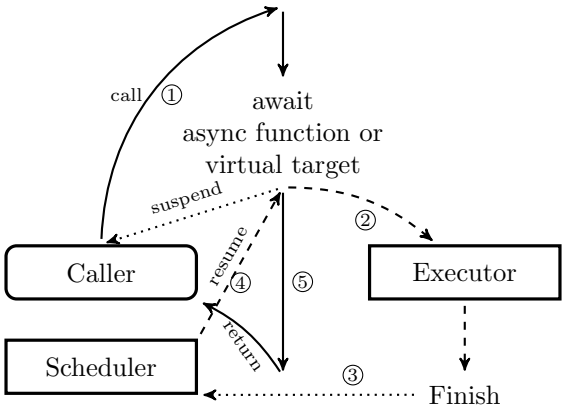


Figure 12: The control flow of using of awaits directive.

```
#pragma omp async
    function-declaration
```

Figure 11: Definition of async directive, for the purpose of annotating an async function.

An async function should be explicitly annotated, to inform the programmers that this function can be invoked in different ways. This is achieved using omp async, to notify the compiler to do the necessary prepossessing (Figure 11). It also informs programmers that the invocation of this function may be different from normal function calls. Invoking this function in the standard way causes it to execute synchronously as would be expected. Instead, if this function is invoked with an async-call await construct, the caller function also becomes an async function.

Figure 12 illustrates the conceptual map of awaiting an async function. ① indicates a caller invokes an async function by using async-call await. This async function is supposed to contain one or more async-call await or target virtual await directives. During the execution of this async function, if there is another async function awaiting or a target block should to be awaited, the execution is delegated by the corresponding executor (can be a specified thread or a thread group), and the caller suspends its execution (②). When the function/target block completes (③), the runtime scheduler enables the caller to resume from the previous suspended point. Then the caller continues its execution (④) with the following execution. This suspend/resume procedure may repeat several times, depending on the number of await directives. Finally the caller gets the return from the async function call (⑤).

3.2.4 Semantics of using exception handling

Exception handling is one type of error recovery mechanism which is widely used in high level languages such as Java. By using try-catch blocks, it presents a readable and intuitive control flow since the error-handling code is separated from the normal execution code, and can be categorized according to the type of exceptions/errors. However, the semantics of try-catch is designed for the synchronous execution model: when an exception occurs inside a try block, the thread encountering the exception immediately checks for a corresponding catch block. If it exists, the encountering thread executes the error-handling code and then resumes to the normal execution flow. Otherwise, if there is no appropriate handler found, it propagates this exception to its caller.

When using target virtual blocks, problems may emerge. First, a target virtual directive may change the thread which executes the following block; the thread that encounters the exception could be different from the thread that handles the exception. Second, if asynchronization is applied, the execution of a target virtual block may happen in a future time, therefore when an exception happens from the target virtual block, it already loses the control of its surrounded try-catch block; although the try and catch blocks are placed lexically correct, semantically the handler is too late due to the async nature.

To overcome these problems and provide a clear specification, the semantic rules of using the try-catch block together with target virtual blocks are defined below:

- If the exception is supposed to be handled and recovered within the target block, a try-catch block should be used inside the target virtual block. This does not break the synchronous execution model since the exception handing thread is the same as the exception encountering thread. This rule is consistent with OpenMP's exception handling requirements [12];

- If an exception escapes from its inner most surrounding target virtual block, the exception will be re-thrown at the nearest synchronization point. A synchronization point is defined as the point which ensures that the execution of the target block is finished. For `await`, the synchronization point is the point directly after the target code block. For `name_as`, the synchronization point is where its paired `wait` directive is placed. If an uncaught exception occurs during execution of the target block, the runtime support does not throw it immediately. Instead, this exception is stored and will be re-thrown until the synchronization point is met. Notice the thread which initially threw the exception can be different from the thread which encounters the re-thrown exception, because of the possible thread context changes.

- If a target virtual block uses the `nowait` asynchronous clause, there is no chance for it to be re-thrown, since there is no way to verify its completion. In this case, all the exceptions escaped from the target block will be automatically handled by the runtime by printing the stack trace,

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

Table 1: Runtime functions to create virtual targets in Pyjama.

and allowing the program to continue. This is consistent with exception handling in event-handling frameworks such as Java Swing's EDT.

3.3 Runtime Library Routines

Parallelism in the traditional OpenMP fork-join model is triggered by a `parallel` directive. The lifecycle of a thread group is strictly confined within a parallel region, so the parallelism cannot span two or more function calls. On the contrary, in order to suit the event-driven programming model and for all event handlers to co-use a parallel region, the task executor model is used. A virtual target is essentially a thread pool executor, or an event dispatching thread, 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. This section describes the additional OpenMP APIs supported by Pyjama, which are used for managing virtual targets at the runtime.

Every virtual target used in the directive requires either a registration or a creation (Table 1). For example, in a Java Swing GUI application, the master thread is the Event Dispatching Thread. In order to notify the compiler regarding the master thread as a virtual target `edt`, a registering function `virtual_target_register(“edt”)` should be executed at the initializing stage (e.g. the constructor of the graphic interface). Similarly, creating a new virtual target can be achieved by a creation function. For instance, invoking function `virtual_target_create(“worker”, 5)`, will create a new virtual target called `worker` which has maximum thread number of 5.

Registering dispatching mechanisms Registering an EDT as a virtual target requires specifying the task dispatching mechanism to the Pyjama runtime, otherwise the runtime cannot post runnable tasks to the EDT targets. However, the dispatching mechanism depends on the GUI framework. In the current experimental version of Pyjama, three types of event-driven GUI framework are already supported, namely Java Swing [31], Android [18] and JavaFX [33]. For other frameworks, programmers are required to specify the task dispatching interface that the framework provides.

4 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 is the source-to-source compiler which supporting traditional OpenMP directives and the extended directives this paper proposed, transforming the sequential Java source code into parallel code. Second is the runtime system, which provides the underlining thread-pool creation, management and task scheduling, as well as all the OpenMP runtime functions.

4.1 Compilation

The semantics of Pyjama is essentially a standard Java with a comment-based OpenMP extension. The Pyjama compiler performs a source-to-source translation. The standard Java code (i.e. the code not annotated with Pyjama's OpenMP directives) is retained without any changes. All other code blocks annotated with OpenMP directives are transformed into parallel Java code. The final Java file generated is compiled into Java bytecode using a standard Java compiler.

The compiler's underlying parser is generated using JavaCC [32] (the parser generating tool), where the OpenMP directives extended the standard Java grammar. The parsing generates an abstract syntax tree (AST) of the Pyjama source code, where the Visitor Pattern [39] is used to traverse the AST to generate the target source code. This section provides an overview of how an OpenMP target block or an async method is converted to the destination code.

Auxiliary Class Generation An auxiliary class is an inner class of the current compilation unit, which contains all the running information of a target block or an async method. In general, each target virtual code block is refactored into an inner class, and this inner class inherits an abstract class called TargetTask. The abstract interface call() is implemented to include the user code. Meanwhile, the class contains several data fields which store the information and track the execution status of this target block; Figure 13 illustrates all the noticeable fields. Meanwhile, all the variables which are used in the target block are also required as field variables in the auxiliary class, and they should be passed in and initialized by the auxiliary class constructor.

Generating code with states If a target block or a method contains any await target blocks, or await constructs, the generated auxiliary class may contain states. The interface call() will be implemented with states, by which the control flow of call() can be resumed to different positions according to the state number. This makes the target block or async function flexible enough to suspend and resume during its execution at appropriate continuation points.

```
//#omp async
int asyncCall()
```

```
public abstract class TargetTask<T> implements Callable<T>{
    //which virtual target should invoke this task.
    private VirtualTarget caller;
    //the callback function should be triggered when this task finishes.
    private CallbackInfo callWhenFinish;
    //the flag indicating if this task is finished.
    private volatile boolean isFinished;
    //the return value, only available for async functions.
    private T result;
    //the current state of the execution, only available when the call needs awaits.
    private int state;
    //the interface the subclass should implement.
    public abstract T call() throws Exception;
}
class CallbackInfo {
    //the continuation call.
    TargetTask<?> callback;
    //who calls this continuation.
    VirtualTarget caller;
}
```

Figure 13: The overview of the TargetTask Class.

```
{
    int result;
    //#omp await (int foo(int a))
    {
        result += foo(1) + foo(2);
    }
    //#omp target virtual(edt)
    {
        Panel.update(result);
    }
    return result + bar(1);
}
```

For each await invocation, an instance of its paired auxiliary class is initialized with arguments. The entire process is separated by states. The state of the control flow cannot process until the current awaiting call is finished. During this waiting, the control flow returns back to its caller. By using an on-completion-handler, the control flow resumes to the appropriate continuation point according to its current state. Finally, at the end of the process, the result of the async method is settled by using the setResult() method:

```
public void call() {
  switch(OMP_state) {
  case 1:
    OMP_AwaitFunctionResult_foo_0 = new _OMP_StateMachine_foo(1);
```

```
    OMP_AwaitFunctionResult_foo_0.setOnCompleteCall(this,
           PjRuntime.getVirtualTargetOfCurrentThread());
    PjRuntime.runTaskDirectly(OMP_AwaitFunctionResult_foo_0);
    this.OMP_state++;
    return null;
  case 2:
    OMP_AwaitFunctionResult_foo_1 = new _OMP_StateMachine_foo(2);
    OMP_AwaitFunctionResult_foo_1.setOnCompleteCall(this,
           PjRuntime.getVirtualTargetOfCurrentThread());
    PjRuntime.runTaskDirectly(OMP_AwaitFunctionResult_foo_1);
    this.OMP_state++;
    return null;
  case 3:
    result += OMP_AwaitFunctionResult_foo_0.getResult() +
          OMP_AwaitFunctionResult_foo_1.getResult();
    OMP_TargetTaskRegion_0 = new _OMP_TargetTaskRegion_0();
    if (PjRuntime.currentThreadIsTheTarget("edt")) {
      PjRuntime.runTaskDirectly(OMP_TargetTaskRegion_0);
      this.OMP_state++;
    } else {
      OMP_TargetTaskRegion_0.setOnCompleteCall(this,
          PjRuntime.getVirtualTargetOfCurrentThread());
      this.OMP_state++;
      return null;
    }
  default:
    this.setResult(result + bar(1));
    this.setFinish();
    return null;
  }
}
```

Target block invocation In the generated code, the invocation of every target block or async method is converted to the invocation of its paired auxiliary class. First, an instance of its auxiliary class is initialized, with proper arguments. Second, the block is scheduled by the runtime routine according to its *asynchronous-property-clause*. For example, consider the following code snippet:

```
Label.setText("Start Processing Task!");
//#omp target virtual(worker) await
{
    compute_half1(); // S1
    //#omp target virtual(edt) nowait
    {
        Label.setText("Task half finished"); // S2
    }
    compute_half2(); // S3
}
Label.setText("Task finished"); // S4
```

The call() interface is implemented by code inside the target code block, in favor of generating the auxiliary class which extends TargetTask. The data context and variables referenced by the user code are stored into this generated class (for simplicity, the demo code omits the field variables). 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() extends TargetTask {
  public void call() {
    compute_half1(); // S1
    TargetRegion _omp_tr_1 = new TargetRegion_1();
    PjRuntime.invokeTargetBlock("edt", _omp_tr_1, Async.nowait); // S2
    compute_half2(); // S3
  }
}
Label.setText("Start Processing Task!"); // S4
TargetRegion _omp_tr_0 = new TargetRegion_0();
OMP_TargetTaskRegion_0.setOnCompleteCall(this, "worker");
PjRuntime.invokeTargetBlock("worker", _omp_tr_0, Async.await);
Label.setText("Task finished");
```

4.2 Runtime

The runtime support includes the runtime functions and the underlining target task dispatching mechanism.

4.2.1 Target block scheduling

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. The runtime routine first checks if the submitting thread is already a member of the virtual target executor's thread group (line 7). 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 8). If the *asynchronous-property-clause* is `nowait` or `name_as` (line 11), the main thread exits the procedure (line 12) to directly execute the statements following the target block. If the *asynchronous-property-clause* is await, the caller suspends its execution; once the target block is finished, the caller continues its execution from where it was suspended, which will be executed by the same thread or executor (lines 15-17). Otherwise, the thread waits for the target block to finish (line 18).

4.2.2 Post to different virtual targets

Using Pyjama runtime requires a mechanism to dispatch events and tasks to the EDT. The approach of posting tasks to the EDT is decided by the particular GUI framework and platform. Figure 2 reveals how the current Pyjama version

Algorithm 1 Target block code execution.

```
1: 𝒯: current thread
2: C: current caller function
3: E: target executor
4: ℬ: target block
5: a: asynchronous property
6: procedure INVOKETARGETBLOCK(𝒯,C,E,ℬ,a)
7:     if 𝒯∈E then
8:         ℬ.exec()                          ▷ execute ℬ synchronously by 𝒯
9:     else E.post(ℬ)                        ▷ post ℬ to E asynchronously
10:    end if
11:    if a is nowait or name_as then
12:        return                            ▷ directly return to caller
13:    end if
14:    if a is await then
15:        C.suspend()
16:        ℬ.setOnCompleteCall(C,𝒯)          ▷ continuation binding
17:        return
18:    else 𝒯.wait()                         ▷ default option
19:    end if
20: end procedure
```

Algorithm 2 Post target block to different types of virtual target.

```
1: E: target executor
2: ℬ: target block
3: procedure POST(E,ℬ)
4:     if E is registeredEDT then
5:         switch E do
6:             case Swing
7:                 SwingUtilities.invokeLater((Runnable)ℬ);
8:             case Android
9:                 Handler uiHandler = new Handler(Looper.getMainLooper());
10:                uiHandler.post((Runnable)ℬ);
11:            case JavaFX
12:                Platform.runLater((Runnable)ℬ);
13:    else E.post(ℬ)                        ▷ post ℬ to a non-EDT target
14:    end if
15: end procedure
```

supports three different types of GUI frameworks. The Pyjama compiler generates the appropriate target code according to the platform the programmer specifies.

4.2.3 Exception handling support

In order to support the asynchronous exception handling, the user code is surrounded by a try-catch block in the implementation of the call() function. Any exception escaped from the target virtual block is stored by the runtime, and then the target block is marked as completed. After any appropriate synchronization point is reached, the getResult() function is invoked. Before returning the result, the runtime checks for any stored exception. If yes, the exception is re-thrown at this point, and then the encountering thread is able to handle it.

```
public void call(){
  try{
    userCode.run();
  } catch (Exception) {
    this.thrown = new Throwable(e);
    this.setFinish();
  }
}
public T getResult(){
  if (null != this.thrown) {
    throw thrown;
  }
  return this.result;
}
```

5 Evaluation

This section provides the evaluations of the proposed approach for event-driven programming. Three case studies are presented.

5.1 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

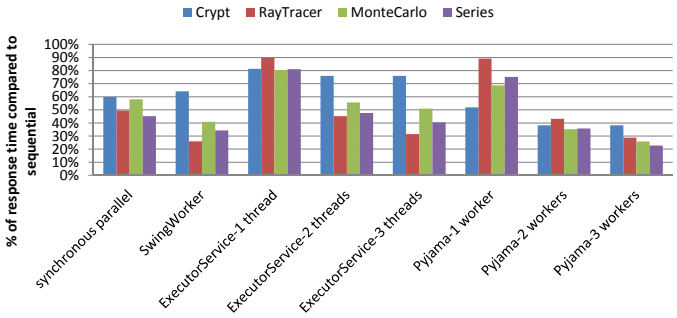


Figure 14: Average event response time, as a proportion of the sequential version, using different offloading approaches and computational kernels (lower is better).

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 [41] (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 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 14 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

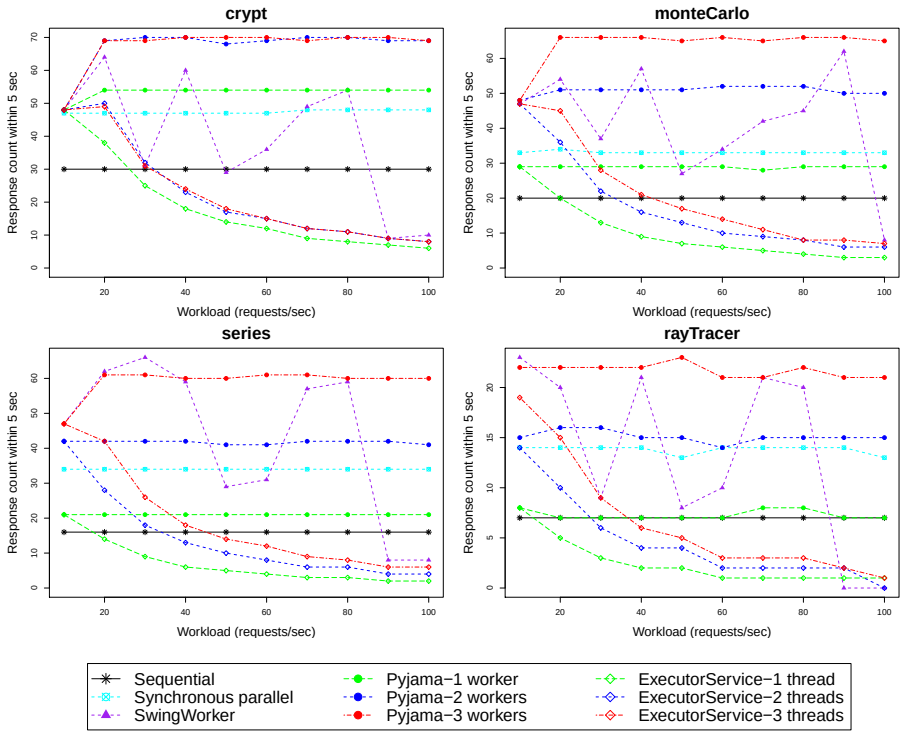


Figure 15: 5-second response count under different request work loads (higher is better).

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 [42]. Using this rule of thumb, Figure 15 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.

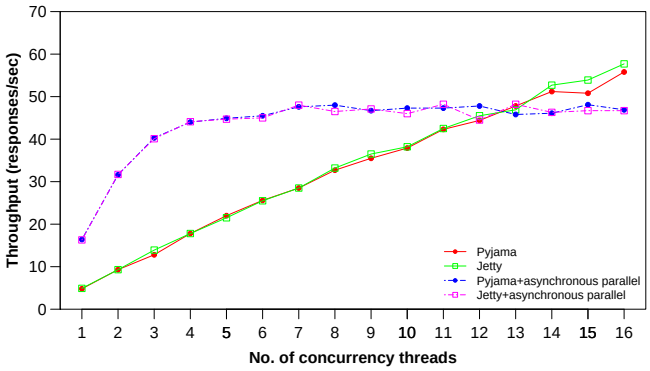


Figure 16: Throughput scaling comparison between Jetty and Pyjama.

5.2 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 16 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 soars to a high value and it leads to a great overhead of thread scheduling.

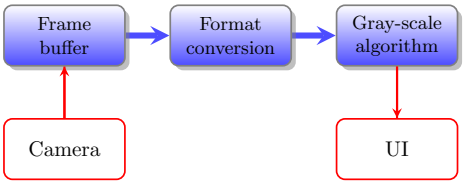


Figure 17: The work flow of the experimental app.

Name	Lines of Code	Refactoring	Frames Processed Per Minute	Responsive UI
EDT(single thread)	17	-	78	no
AsyncWorker	29	yes	103	yes
Pyjama	19	no	106	yes

Table 2: The comparison of three approaches, with regard to programming effort and performance.

5.3 Android application

This section presents a case study of using Pyjama on the Android platform. The purpose of this case study is to evaluate the effectiveness of using Pyjama to boost the performance and responsiveness of a mobile application.

The scenario of this experiment is using a demo Android app to process the images collected from the camera device on the fly. When the app starts, the program constantly picks one frame from the camera frame buffer and applies a gray scale image processing using Catalano Framework [6], then the app updates the result of the processing to the user interface. This application was evaluated on a Nexus 4 with 1.5 GHz quad-core, running Android 4.4.4 OS. For a single round of the image processing, in which the frame size is 460800 bytes, it takes around 200ms to convert camera compatible format NV21 to image processing library compatible format Bitmap. Then the gray scale processing takes approximately 250ms and the UI update takes about 20ms (Figure 17).

The event handler is implemented in three different approaches, and for each the LoC (Lines of Code) and FPPM (Frames Processed Per Minute) are compared in Table 2. The first approach is the single-threaded implementation, in which case the EDT is responsible for all the background computations and UI updates. This leads to low performance and bad responsiveness of the UI, since the UI is frozen during the background computation. The second approach uses Android's AsyncWorker class to offload the computation asynchronously to a background thread. While this gains better performance in terms of FPPM, but it requires code refactoring. The final approach, using Pyjama, gains the competitive performance improvement but without the refactoring effort required in AsyncWorker.

Since Pyjama is directive-based, it promotes an incremental programming approach enabling programmers to retain the sequential code. Therefore, it

avoids code restructuring, the variable scope and programming context also do not change. In summary, Pyjama is effectively automatically generating asynchronous code native to the application, this means programmers witness the equivalent performance boosts they can expect from a manual refactoring, but with minimal programming effort over the original single-threaded implementation.

6 Related Work

6.1 Asynchronization

Asynchronous programming is traditionally used in single-threaded applications to achieve cooperative multitasking [11]. 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 [28] building asynchronous computations. Java NIO libraries [30] provide non-blocking and asynchronous I/O operations. Microsoft .NET provides three types of asynchronous programming patterns [25]: (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 [23] is an asynchronous event-based network application framework written in C, adopting proactor pattern [36], which is an object behavioral pattern of the combination of I/O multiplexing [38] 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. [14] 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 [26]. New language designs also tend to support asynchronization. For example, P [9] is a domain-specific language for the modeling of state machines, and all machines communicate via asynchronous events. Eve [15] is another parallel

event-oriented language for the development of high-performance I/O applications. Other language-level concepts such as the actor model [20, 19] and co-routines [8] provide variations to asynchronization.

6.2 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 [37] and provides interfaces for task submissions. Some languages support tasks at a language level, such as Cilk [16] and JCilk [7]. While OpenMP provides the `task` directive [27], 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 [17], as a language extension of Java, supports task creation and dependency handling. OoOJava [21] and DOJ [44] both introduce the task keyword to achieve out-of-order execution of the code blocks, with the support of automatic dependency analysis between tasks.

7 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.

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] Diego Catalano. Catalano Framework: a framework for scientific computing for Java and Android. `https://github.com/DiegoCatalano/Catalano-Framework`, 2016.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] Xing Fan, Mostafa Mehrabi, Oliver Sinnen, and Nasser Giacaman. Exception handling with OpenMP in object-oriented languages. In *International Workshop on OpenMP*, pages 115–129. Springer, 2015.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Google. Android. Available at: `https://www.android.com/`, June 2016.

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

[20] 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.

[21] 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.

[22] 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.

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

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

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

[26] 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.

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

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

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

[30] 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.

[31] Oracle. Java Swing Package. Available at: `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`, June 2016.

[32] Oracle. JavaCC Project. Available at: `https://java.net/projects/javacc`, June 2016.

[33] Oracle. JavaFX Client Platform. Available at: `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm`, June 2016.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] Markus Schordan. The language of the visitor design pattern. *Journal of Universal Computer Science*, 12(7):849–867, 2006.

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

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.