Fluid: A Framework for Approximate Concurrency via Controlled Dependency Relaxation

Huaipan Jiang
Pennsylvania State University
State College, USA
hzj5142@psu.edu

Haibo Zhang*
Pennsylvania State University
State College, USA
huz123@psu.edu

Xulong Tang
University of Pittsburgh
Pittsburgh, USA
tax6@pitt.edu

Vineetha Govindaraj
Pennsylvania State University
State College, USA
vzg99@psu.edu

Jack Sampson
Pennsylvania State University
State College, USA
jms1257@psu.edu

Mahmut Taylan Kandemir
Pennsylvania State University
State College, USA
mtk2@psu.edu

Danfeng Zhang
Pennsylvania State University
State College, USA
zhang@psu.edu

Abstract

In this work, we introduce the *Fluid* framework, a set of language, compiler and runtime extensions that allow for the expression of regions within which dataflow dependencies can be approximated in a disciplined manner. Our framework allows the eager execution of dependent tasks before their inputs have finalized in order to capitalize on situations where an eagerly-consumed input has a high probability of sufficiently resembling the value or structure of the final value that would have been produced in a conservative/precise execution schedule. We introduce controlled access to the early consumption of intermediate values and provide hooks for user-specified quality assurance mechanisms that can automatically enforce re-execution of eagerly-executed tasks if their output values do not meet heuristic expectations. Our experimental analysis indicates that the fluidized versions of the applications bring 22.2% average execution time improvements, over their original counterparts, under the default values of our fluidization parameters. The Fluid approach is largely orthogonal to approaches that aim to reduce the task effort itself and we show that utilizing the Fluid framework can yield benefits for both originally precise and originally approximate versions of computation.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; *Object oriented frameworks.*

Keywords: Eager Execution, Approximate Computing

ACM Reference Format:
Huaipan Jiang, Haibo Zhang, Xulong Tang, Vineetha Govindaraj, Jack Sampson, Mahmut Taylan Kandemir, and Danfeng Zhang. 2021. Fluid: A Framework for Approximate Concurrency via Controlled Dependency Relaxation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454042

1 Introduction

Many recent works have examined models of approximate computation as a means of improving performance [57] and/or energy efficiency [29, 57] for error-tolerant and self-correcting applications, by coping with unreliable components [34], or executing inherently stochastic algorithms [13]. Common approaches among these models include eliding the execution of certain tasks, or replacing an approximable task with an entirely different computation that is easier to execute [26]. A different flavor of approximation has been studied in the form of data-race tolerant and other scheduling-robust algorithms under weakened synchronization assumptions [7]). For an approximate computation approach to be viable, the intermediate approximated values consumed by the following tasks must be close enough to the values generated by precise computation. Note, however, that once approximation is allowed and soundness constraints have therefore been relaxed, there is no obvious reason that the

*Work was done as a student at Pennsylvania State University

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454042

communicated value in question only be visible at the ***end*** of an approximate computation; *the time at which an approximate output becomes visible to a subsequent consumer is at least as amenable to approximation as the process of the production of the value itself.*

This observation leads directly to the exploration of forwarding the eager executed data. Conceptually, the execution of any workflow can be considered as a sequence of interdependent kernels and each kernel, in turn, as a sequence of interdependent tasks. Each task can be viewed as consuming a set of inputs and can act in turn as a producer of the values consumed by other tasks. In a precise computation, any particular execution represents a schedule, serial or parallel, that obeys the dependencies expressed among these producers and consumers. In an approximate computation, however, there can be opportunities to forward the approximate data between the tasks. The key challenge in describing and exploiting these opportunities lies in annotating the dependency between the tasks, as well as ensuring some user-defined quality controllers at the boundaries between approximate and precise computations.

To boost such approximable applications, recent studies either break the data dependency or relax the synchronization to increase the degree of parallelism [11, 14, 17, 24, 38, 42, 44, 49, 62, 69, 71, 84]. Unfortunately, none of these works presents a general framework which allows users to specify the condition of breaking the dependency. As a result, in this paper, we present *Fluid*, a novel paradigm for expressing "*eager execution*" by relaxing the inter-kernel or inter-task data dependency constraints. We adapt an *object-oriented* model to encapsulate all accesses to the set of data structures utilized within a particular region of approximately concurrent execution, and to control the data forwarding between precise and approximate portions of execution. Fluid also employs a user-specified "satisfaction (quality) function", as a means of providing a heuristic for throttling or abandoning a particular path of eager computation to mitigate error. Specific **contributions** of this work include:

• We propose a *Fluid Framework*[1] for data dependency relaxation which can operate on "partially computed" values, and provide interfaces that let users readily express executions exploiting *Fluid Concurrency* for performance while controlling the magnitude of approximation-induced errors.

• We give the details of the programming language, compiler and runtime support to realize Fluid as a set of pragma extensions to C++. Our compiler consumes these pragmas to produce standard C++ programs that interact with the Fluid runtime system. We report that, on average, one needs to insert only 12.4 pragmas per application program, which corresponds to 3.9% of the total program lines.

• We evaluate the fluidized versions of eight applications and show, on average, 22.2% latency improvements, over their precise counterparts, with 1.4% reduction in accuracy, for empirically-chosen Fluid valve hyperparameters.

• We demonstrate that Fluid approximation cleanly composes with multi-threaded applications (Edge Detection and K-means) indicates that Fluid brings additional performance benefits over conventional multithreading.

[1]The source code of Fluid is available at https://github.com/j9650/Fluid.git.

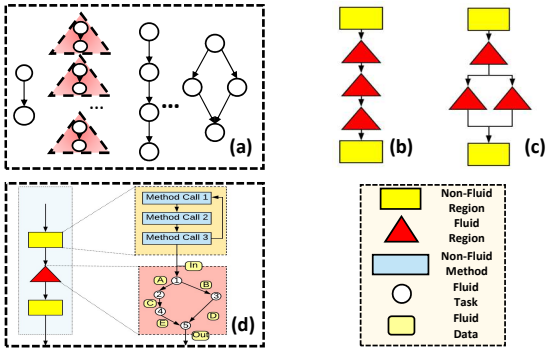


Figure 1. (a). Different types of task graphs in applications, red triangles are Fluid regions while the white circle represent tasks; (b). Multiple Fluid regions between non-Fluid regions; (c). A Fluid region with multiple output regions; (d). Tasks invocations within a Fluid/non-Fluid region.

2 Approximated Data Dependency

In an execution model that schedules a set of producers and consumers, there are three potential opportunities for deriving benefits from scheduling a consumer *early*. First, if the input to the consumer has already attained its "final value" before all possible updates have been applied, then starting at the earliest point where this is true would be beneficial for any consumer. The second case is where, while subsequent updates will change the value, the current value is "indistinguishable" from the final value from the perspective of the consumer. The consumer will generate the identical output no matter if it consumes the current value or the final value. Third, while subsequent updates to input X to a consumer task C will change the value of X, and do so in ways that change $C(X)$, C is inherently approximate and the difference between $C(X_{current})$ and $C(X_{final})$ is within the "approximation tolerance".

Clearly, not all code regions are amenable to approximation. Real applications consist of both precise and approximable code. For instance, precise business logic may interact with outputs produced by approximable natural language processing kernels within a larger decision-making workflow. Additionally, approximability and producer-consumer relationships can be hierarchical, e.g. inter-method approximation is possible within an approximate kernel. Figure 1(a) shows a set of example data dependence graphs present

within and among kernels. These range from simple single-producer-consumer inter-kernel sequences (leftmost) to kernels decomposed into pairs of producer-consumer approximable methods in a non-approximable sequence (center-left), long chains of consumers that are producers for the following kernel in a workflow (center-right), and multi-producer-multi-consumer relationships (rightmost). While clearly non-exhaustive, even these few examples point to the diversity of dependency approximation scenarios and the potential benefits of eager execution.

Figure 1(b) and (c) show different possible relationships between precise (rectangle) and approximate (triangle) code regions. Without loss of generality, we consider these regions as single-entry: If a region is not naturally single-entry, an additional (header) task can be added on which all other potential points of task initiation are made dependent. We further constrain data visibility in our model such that, for any data that is an output of the region, there is an associated readiness function that can restrict accesses from other regions which will, at a minimum, ensure that no subsequent updates to the output value (which may be simple data or an entire data structure) will be made by the producing region. More restrictive readiness functions can be used to i) control inter-region parallelism (e.g., output barriers), ii) enforce quality constraints and heuristics, or iii) bound the scope required for reasoning about rollback or other speculation management mechanisms. In the following sections, we describe our specific approach to exploiting "intra-region" concurrency for approximable regions, *Fluid*, in detail and also describe our implementation of the associated compiler and runtime support to enable its unique features.

3 Fluid Execution

In this paper, we primarily focus on the potential improvements in concurrency within a single approximable region. Our approach employs a "guard" that manages the tasks of starting, terminating, and restarting. The start and end of each task are controlled by associated "valve" functions that communicate the satisfaction of eagerness and quality constraints to the guard managing that task.

3.1 Overview of Fluid Execution

Our Fluid framework builds atop an object-oriented programming model and introduces the new features described in Table 1. We list the Fluid concepts in the left column and their corresponding definitions in the right column.[2]

To see Fluid's major concepts in action, consider a high-level view of a program as a set of data-dependent tasks which consist of some serial invocation of methods according to the program logic. Figure 1(d) shows such a breakdown, expanding an inter-region sequence, showing an approximate (Fluid) region sandwiched between two precise, sequential task regions. While each rectangular region contains some fixed ordering of task invocations, each *Fluid region* consists of multiple schedulable Fluid tasks and their associated static dataflow graph. The nodes of this graph correspond to *tasks* (dynamic instances of methods) and an edge from node A to node B captures the *dataflow* between them. A valve function is associated with the *Fluid Data* between two tasks to determine whether this data is ready for consuming. Note that, unlike in the precise program, the valve function may return true even the data is not (yet) fully produced. A task in the Fluid region can start its execution as soon as *all* of the valves that control its input data are *satisfied.* Each leaf task in a Fluid region has *end valves* that collectively constitute its associated *quality function.* Eagerly computed data cannot leave a Fluid region until satisfying the quality function. It is interesting to note that setting all valves to require the completion of antecedents within the dataflow graph will result in a "precise execution" of the entire task graph. Since multiple valves (attached to different edges) can be satisfied independently (and in parallel), multiple tasks can execute *concurrently* in a Fluid region, resulting in what we call "Fluid Concurrency" (more on this later).

Consider the example shown in Figure 1(d). Execution starts with a *Non-Fluid region* and generates data *In*. *In* is sent to a *Fluid region* as the input. Within the Fluid region, task1 receives *In* as the input and generates *A* and *B*. At some point, task2 and task3 take their inputs and generate *C* and *D*, respectively. Note that task2 and task3 may start their executions *before* task1 has finished. Also, task2 and task3 can be started at different times and run in parallel. Later, task4 takes *C* as the input and generates *E*. Finally, when the valve functions are satisfied, task5 takes *D* and *E* as input and generates *Out*. If *Out* meets the end quality check, this Fluid region has finished and the next region starts.

3.2 Fluid Concurrency

Our proposed Fluid programming paradigm enables a new type of concurrency, called *Fluid Concurrency*, which comes in two flavors: *Intra-Region Concurrency* and *Inter-Region Concurrency.* In the former one, the different tasks in a Fluid region can be executed concurrently if the corresponding valves evaluate to true. Note, however, that valve satisfaction only implies the corresponding task becomes *schedulable* – exactly when it starts its execution depends on resource availability as well as the underlying scheduling strategy employed (Section 6 discusses our runtime system).

The initial data that triggers the execution of a region is *non-Fluid*, and the output data resulting from the execution of a region is also *non-Fluid.* As such, fluidity (that is, Fluid Concurrency) is *confined* within the boundaries of a *Fluid region.* Consequently, even an otherwise sequential (single

[2]Note that, while the way we envision and implement the Fluid execution blends very well with object-oriented computing, we believe the Fluid data concept can be exploited in other programming paradigms as well.

threaded) program can take advantage of the Fluid concurrency offered by our programming paradigm. While a Fluid region has only one input, it can have multiple outputs, each driving a Fluid or non-Fluid region. As a result, multiple Fluid regions can execute concurrently, as depicted in Figure 1(b), leading to inter-region concurrency.

In comparison, in a conventional multi-threaded/parallel programming, execution is governed by/constrained by strict data synchronizations between parallel computations, which limits potential parallelism due to the full-accuracy requirement. In Fluid, on the other hand, we can exploit concurrency between fluid tasks by the approximation of data (which is controlled by valves). We want to emphasize that, it is also possible for a program to take advantage of *both* conventional parallelism and Fluid parallelism, e.g., each individual thread of a multi-threaded program can employ intra- and inter-region Fluid concurrency. In our experiments presented later, we also evaluate such parallel application programs.

3.3 Discussion

• **Why do we need a new approximation framework?** Currently, the task of coding new approximate versions of algorithms is both labor intensive and not particularly portable. Rather than requiring a programmer to develop an entirely new approximate algorithm to replace a precise one, Fluid allows programmers to use high level, portable language constructs to encode approximation into sequence relationships among well-defined, existing tasks. While this can be done manually, without language support (akin to how one can write an object-oriented program in a non-object-oriented language), doing so for every approximation transformation would be prohibitive. Moreover, Fluid's approximation knobs can easily compose with existing approximate codes.

• **Why build Fluid into an OOP paradigm?** There are two key reasons that we have implemented Fluid as an extension to an OOP language. First, it is important that there be a means of isolating approximable Fluid data, as it goes through its intermediate, non-final states, from being accessed by any functions that were not explicitly written to operate on such Fluid data. The data encapsulation aspect of OOP is a sufficient and convenient means of providing this functionality. Second, the functions that operate on Fluid data are not invoked in a procedural fashion, but rather begin their execution based on properties of the associated (approximable) Fluid data. This naturally leads to a data-first organization of data and execution, many aspects of which are easily captured by existing OOP paradigms.

• **What types of approximation scenarios synergize with Fluid?** Fluid approximation is applicable whenever there is a producer-consumer relationship with the following property: The intermediate state of the producer can be interpreted to generate a "meaningful value" with the same type as the eventual producer output. This can be as simple, for example, as using the currently observed minimum in a min() function as the current state of the producer and exposing it, or as complex as exporting a snapshot of the state of all mesh elements in an iterative relaxation problem. In general, many optimization problems are fundamentally iterative, and therefore can conceivably be fluidized.

• **How do we expect programmers to use this framework?** The user needs to make three key decisions: i) What type of valve will control a Fluid dependency, ii) what threshold/condition setting will be used for that valve, and iii) what quality metric, if any, will be used to determine whether the consumers eagerly met expectations for output quality. Our framework allows these decisions to be made and applied in a disciplined and explicable fashion. It provides a set of standard valves to cover common scenarios and allows the user to easily produce, if desired, "application-specific" valves and quality functions to match the needs of their algorithms and inputs. Section 4 provides a concrete example of how an application programmer will fluidize an existing application. We also want to mention that, in general, only a small set of annotations need to be added to the program to achieve desired approximation-performance tradeoff.

More broadly, when programmers fluidize a program, they will have to consider how to integrate Fluid parallelism with existing environments and platforms. Specifically, we expect that most programmers will use Fluid in conjunction with standard libraries that may not have been designed for Fluid execution. If the library calls within the program are used in an iterative fashion to produce an output that progresses through a sequence of internally consistent states, then, even if the library call may not be trivially fluidized internally, then a fluid region can still be constructed around the sequence of calls. However, in some cases, large portions of the work in a program may be occurring within individual library calls. For example, a CNN developer may rely on the BLAS [10] library's SGEMM function. In the near term, we would expect developers interested in aggressive performance tuning to implement a Fluid version of the SGEMM function with valve, count and quality function support. Our vision, however, is for the collective work of such developers to lead to extended libraries that support both Fluid and non-Fluid versions of heavyweight functions suitable for fluidization.

• **Limitations of the current implementation.** Firstly, Fluid only focuses on true dependencies as they are the primary mechanism through which data are transferred across tasks/functions. Multiple tasks defining the same (storage-associated) output will need to use Fluid's sync function to preserve anti-dependency ordering. Secondly, Fluid is based on an OOP paradigm. As such, "class" is the central concept in our model. An advantage of using classes is that they encapsulate the Fluid data and the Fluid methods that operate on them. We want to emphasize, however, that our approach to concurrency (Fluid concurrency) is entirely different from existing concurrent OOP paradigms (e.g., Actor based programming [2]), where objects themselves are the primary

Table 1. Major concepts in the Fluid programming paradigm.

Concept	Definition
Fluid Valve	A condition function which returns *true* or *false*. It can be used to control the start and end of a task.
Fluid Guard	A processing entity that manages the execution sate of a Fluid task based on its valves or data dependence.
Fluid Member	A Fluid method or Fluid data.
Fluid Class	A type of class with Fluid members that encapsulates both the data and code subject to approximation.
Fluid Object	An instance of a Fluid class.
Fluid Method	A function defined in a Fluid class. Fluid methods may call non-Fluid methods, but the reverse is restricted.
Fluid Data	A data structure declared as Fluid. It can only be accessed by Fluid methods while in a non-final state.
Fluid Task	A dynamic instance of a Fluid method. Its execution is managed by a guard. It can be triggered by other tasks.
Fluid Region	Each Fluid object defines a Fluid region which is represented as a graph where each vertex corresponds to a Fluid task, and each edge is labeled by a Fluid data. Only leaf tasks can have end valves. Each Fluid object has a scheduler. Each Fluid task has a state machine.

concurrency primitives. In contrast, in our approach, objects are not executed in parallel; however, different tasks within an object (in the case of *intra-region parallelism*) as well as tasks that belong to different objects (in the case of *inter-region parallelism*) are executed concurrently. Thirdly, in our model, only the leaf nodes in a region tree have associated end valve functions, and the intermediate nodes do not. This is mainly because allowing intermediate nodes also to have end valve functions would make the determination of the termination of the intermediate nodes quite difficult. More specifically, the termination condition for intermediate nodes depends only on topology information (e.g., all of its children have completed). Fourthly, a region can have only one input but multiple outputs. Further, the input and all outputs should be non-Fluid. This is because, as discussed earlier, we wanted a region to be a *self-contained* entity, which can be plugged into any suitable place in a given program code or used as a modular kernel replacement in a larger workload. However, one can also imagine scenarios where a region would prefer Fluid data as input. We believe that, in most of such cases, the two regions involved can be combined into one region. Fifthly, the Fluid framework only supports 1-D arrays as Fluid data. However, the users can index the array by themselves, e.g., they can specify a 2-D N by M array by specifying a 1-D array with a size of N×M. Lastly, a Fluid program will suffer from a high overhead if the application contains many tasks since each task will create a new thread. Also, if the number of tasks is more than the number of the cores of the machine, Fluid will probably not achieve good speedup. Using a thread-pool will clearly mitigate these overheads, but that feature is not yet supported in the current version of Fluid.

4 Program Language Support

In this section, we describe the syntax and semantics of the proposed pragma-based Fluid extensions to C++, and provide a concrete example of fluidizing a real piece of code.

FluidStmt :: $FluidDef \mid PragmaStmt$
FluidDef :: __Fluid__ $class$
PragmaStmt :: $DataPra \mid ValvePra \mid CountPra \mid TaskPra$
DataPra :: #pragma data {data_type d; } | #pragma data {data_type $*\ d$; }
CountPra :: #pragma count {data_type ct; }
ValvePra :: #pragma valve {data_type v ($para$...); }
TaskPra :: #pragma task <<< $task\_name, SV$, EV , $Inputs, Outputs$ >>> $func$ ()

Figure 2. Syntax of the Fluid Language.

```
1  __Fluid__ class EdgeDetection{
2  public:
3    #pragma data {Image *d1;}
4    #pragma data {Image *d2;}
5    #pragma data {Image *d3;}
6    #pragma count {int ct;}
7    #pragma valve {ValveCT v1;}
8    #pragma valve {ValveCT v2;}
9    void Gaussian(Image *input_img, Image *output_img, count ct);
10   void Sobel(Image *input_img, Image *output_img);
11   void Region();
12   Image *input_img, *img_after_Gaussian, *output_img; };
13 void EdgeDetection::Region () {
14   d1->init(input_img);
15   d2->init(img_after_gaussian);
16   d3->init(output_img);
17   ct.init(0);
18   #pragma task <<<t1, {}, {}, {d1}, {d2}>>>Gaussian( input_img, img_after_gaussian, ct);
19   v1.init(ct, 0.4*input_img->size);
20   v2.init(ct, input_img->size);
21   #pragma task <<<t2, {v1}, {v2}, {t1}, {d2}, {d3}>>>Sobel(img_after_gaussian, output_img);
22   sync(t2); };
23 void main() {
24   EdgeDetection *S1 = new EdgeDetection, *S2 = new EdgeDetection;
...  ...
25   S1->Region();
26   S2->Region();
27   sync(); }
```

Figure 3. Programmer-level code using Fluid pragmas.

4.1 Syntax and Semantics

Figure 2 describes the syntax of the Fluid language extensions. For a given statement in an application source code, the programmer can employ fluidity through explicit use of *FluidDef* and *PragmaStmt*.

FluidDef: The `__Fluid__` keyword in a class declaration declares it to be a Fluid class. A Fluid class must satisfy the following properties: i) it must contain a public *Region()* method; ii) it must contain at least one Fluid data member and one Fluid method. Invoking *Region()* will, among other initialization functions, construct a *Fluid Task Tree* based on static data dependencies among the class's Fluid method functions (also referred to as "tasks"). Fluid data will be shared between a parent and a child. Only Fluid methods can take Fluid data as parameters; iii) there should be only one root task for the task tree, and there should at least one leaf task; and iv) a task can only be scheduled in the *Region()* function; other non-Fluid methods cannot invoke a Fluid method. However, a Fluid method can invoke a non-Fluid method.

PragmaStmt: We use **#pragma data**, **#pragma count**, and **#pragma valve** to annotate predefined types, and we use **#pragma task** to schedule a task in our Fluid framework.

The data pragma declares a Fluid data member that will be shared between two Fluid tasks, whose value may be used to trigger the execution of dependent Fluid tasks. There are two different ways to make the declaration. If the Fluid data is a variable, (e.g., an integer x), we can indicate that x is a Fluid data directly by using "**#pragma data** {`int` x};". Second, we can also indicate that an array is a Fluid object. Take an integer array "$int * A$", as an example. We declare a Fluid data by "**#pragma data** {`int` *d};", and then initialize d by "$d \leftarrow init(A)$", meaning that d is a Fluid data, and the value of d corresponds to the values of the elements in A.

The count pragma provides introspection on the state of Fluid data by counting related events or tracking key statistics. For example, it can be used to monitor the number of updates performed on a Fluid data in a Fluid task, and it can also be used to record the average value of an array of Fluid data. Specifically, a counter ct is declared as a predefined type `__count__<T>`. The template type T can be any generic type offered in C++. For example, we can declare an integer count ct by "**#pragma count** {`int` ct};". We can use ct to monitor *the number of updates* to Fluid data x by invoking "ct++;" after each update of x.

The valve pragma includes the declaration of a *valve*, which is a predefined class in our framework. It will check whether a Fluid data is satisfied and returns either true or false at any given time. For example, a *count valve* (`valveCT`) accepts two parameters: a count ct, and a threshold t. x monitors the number of updates to a Fluid data d. The check function, on the other hand, keeps comparing the value of x with a programmer-defined "threshold" t, and the count valve is said to be *satisfied* when $x > t$, which means that the Fluid data d has been updated more than t times.

The task pragma is used to invoke a Fluid task. A task is a proxy of a member function in the Fluid class, and it consists of two parts: a *guard (<<< >>>)* and a *function (func())*. The guard has five fields. The first field (*task_name*) names the task. The second field (SV) indicates the start valve set. The semantics is that $func()$ *cannot* start its execution until *all* its valves are satisfied. The third field, EV, is a set of valves for the end condition of this task – collectively, these valves implement an *output quality* function. Recall from the discussion in Section 3.3 that only a leaf task in a Fluid region can contain a non-empty set of end valves. A task is considered as having produced an output of sufficient quality only when *all* end valves are satisfied. Satisfaction of this quality property is used in re-execution decisions and task/region-level descheduling operations performed by the runtime system described in detail in Section 6. The fourth ($Inputs$) and fifth ($Outputs$) fields are two sets of Fluid data, which refer to the inputs and outputs of this task. Note that the input and output data for each task determine the topology of a Fluid region. For example, if a Fluid data d is listed in the $Outputs$ field of a task $t1$ and the $Inputs$ field of task $t2$, we can infer that there is a *data dependency* between $t1$ and $t2$. Task $t1$ is the parent node of task $t2$ in the *task tree* of this Fluid region. The function part *func()* must be a Fluid method that is a member of this Fluid class.

4.2 Synchronization APIs

We support synchronization APIs in our framework in order to control the parallelism between and within the Fluid regions. More specifically, the $sync(...)$ API is used to implement a barrier synchronization and guarantee that a specified set of tasks are finished at a given point in execution. It can take a single task as a parameter and blocks until the task finishes. It can also take an instance of a Fluid class as its parameter to wait for all the tasks in that Fluid instance to finish. If no parameter is specified, it blocks the program until *all* scheduled tasks in the program finish.

4.3 A Fluid Code Example: Edge Detection

We demonstrate the fluidization of Sobel filter-based Edge Detection [90] as an example of how an application source code is modified using our Fluid language extensions. In this edge detection algorithm, we first apply a Gaussian filter on all the pixels of the image to remove the noise and then use the Sobel filter to calculate the gradient for each pixel to select the edges based on the gradient value. Figure 3 shows the fluidized code. There are only two tasks ($t1$ and $t2$) in the task graph of the Fluid region. Fluid data $d1$, $d2$ and $d3$ correspond, respectively, to the input image, the internal image after Gaussian filter, and the output image. Note that $d2$ is shared between $t1$ and $t2$, which indicates that there exists a data dependence between the two tasks. Task $t1$ corresponds to *Gaussian()*. It takes $input\_img$ ($d1$) as input, applies the Gaussian filter on each and every pixel on the image, and stores the output in $img\_after\_gaussian$ ($d2$). Task $t2$ corresponds to the Fluid method *Sobel()*. It reads from $img\_after\_gaussian$ ($d2$) and outputs its result to $output\_img$ ($d3$). The count ct is used to monitor how many updates have been performed so far on $d2$. The two count valves, $v1$ and $v2$, check whether the number of updates is greater than the specified thresholds. The start valve $v1$ indicates that $t2$ cannot start its execution until at least 40% of $d2$ has been updated in $t1$. The end valve $v2$ indicates that the execution of $t2$ is not considered acceptable unless all pixels of $d2$ have been updated before $t2$ finishes executing. This enforces the requirement that, if only a few pixels are smoothed by the Gaussian filter before Sobel finishes (e.g. Sobel raced too far ahead of Gaussian), the execution considers the result inaccurate and $t2$ is *re-executed*.

4.4 Valve and Threshold Selection and Automation

Exploiting Fluid parallelism relies on identifying proxies for the maturation of Fluid data that can be encoded as

```
1  class EdgeDetection{
2  public:
3    fluid::data *d1;                  // #pragma data {RgbImage *d1;}
4    fluid::data *d2;                  // #pragma data {RgbImage *d2;}
5    fluid::data *d3;                  // #pragma data {RgbImage *d3;}
6    fluid::count<int> ct;             // #pragma count {int ct;}
7    fluid::ValveCT v1;                // #pragma valve {ValveCT v1;}
8    fluid::ValveCT v2;                // #pragma valve {ValveCT v2;}
9    void Gaussian(Image *input_img, Image *output_img, int *ct);
10   void Sobel(Image *input_img, Image *output_img);
11   void Region();
12   Image *input_img, *img_after_Gaussian, *output_img;
13   fluid::TaskScheduler *ts; }
14 void EdgeDetection::Region () {
15   d1->init(input_img);
16   d2->init(img_after_gaussian);
17   d3->init(output_img);
18   ct.init(0);
19   //#pragma task <<<t1, {}, {}, {d1}, {d2}>>>Gaussian(input_img, img_after_gaussian, ct);
20   auto tpb__t1 = std::bind(&EdgeDetection::Gaussian, this, input_img, img_after_Gaussian, ct.ct());
21   auto tp__t1 = ts->NewFunc<decltype(tpb__t1)>({d1}, {d2}, tpb__t1);
22   fluid::Task *t1 = ts->NewTask->newTask({}, {}, tp__t1);
23   //#pragma task <<<t2, {v1()}, {t1}>>>Sobel(img_after_gaussian, output_img);
24   auto v1_ = v1.init(ct, 0.4* input_img->size);
25   auto v2_ = v2.init(ct, input_img->size);
26   auto tpb__t2 = std::bind(&EdgeDetection::Sobel, this, img_after_Gaussian, output_img);
27   auto tp__t2 = ts->NewFunc<decltype(tpb__t1)>{{d2}, {d3}, tpb__t2};
28   fluid::Task *t2 = ts->newTask{{v1_}, {v2_}, tp__t2};
29   sync(t2); };
30 Void main() {
31   EdgeDetection *S1 = new EdgeDetection, S2 = new EdgeDetection;
...  ...
32   S1->Region();
33   S2->Region();
34   sync(); }
```

Figure 4. Code from Figure 3 after pragma translation.

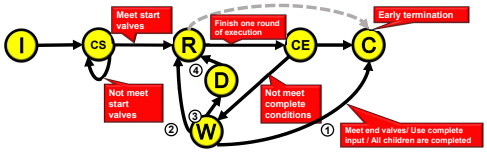


Figure 5. State machine for a Fluid task.

valves. Fluid directly supports iteration count and value stability proxies, but other tasks may have data-structure or algorithm-specific metrics that better indicate whether Fluid data is useful to consume. Correctly understanding what the best measures of partial completion will be to trigger each task requires either domain knowledge on the part of the programmer or a substantial training set from which ML techniques can be used to discover relevant features. In general, both valves and thresholds could be inserted and controlled, to some degree, by the compiler and runtime. Firstly, the user specifies a minimum degree of execution/fulfillment of a property by a producer of Fluid data and, therefore, any effective threshold value between the specified value and full serialization is valid. This means that a Fluid runtime is free to dynamically adjust the threshold based on whether this valve participated in a chain of tasks that failed its quality function. We will discuss more details in section 6.1. Also, ML-based policies could be deployed to auto-tune both the types of valves and the thresholds. This would only be safe to automate for task chains that end in user-specified quality functions that would act as implicit lower bounds on the thresholds generated by the runtime. Our implementation does not have these features yet. The evaluation of tradeoffs between runtime complexity and performance improvements from auto-tuning is a topic of future work.

5 Compiler Support

We implemented a source-to-source translator from scratch that *automatically* maps a pragma-based fluidized application code into an equivalent C++ code. Since our task scheduler always works with a valid topological sort of data dependencies, the fluidized code will be correctly compiled even if all pragmas are ignored – but in that case it will *not* make any use of approximate concurrency. Note also that a fluidized program compiled by the Fluid framework can specify the task scheduling declarations in any order and would still execute correctly. To show how our translator works in practice, we focus on the edge detection code from Figure 3. Our compiler automatically translates this user-written code to the equivalent C++ code shown in Figure 4.

Firstly, in the declaration of a Fluid class, the pragmas are unwrapped, and each type is translated to our pre-defined data type (lines 3 to 8). We add a "TaskScheduler" *ts* at the end of the declaration, which will be utilized in future scheduling of the tasks. *ts* also provides internal interfaces for binding Fluid methods and Fluid data into schedulable task-execution functions, and for generating task objects that couple the guard and task-execution functions together.

Secondly, our compiler translates the task scheduling statements. For example, task $t1$ is translated to the code encapsulated between lines 20 and 22. The first statement (line 20) binds the Fluid method function with its parameters. The second statement (line 21) generates a new task-execution function, with internal task-scheduler interface, and associates it with the Fluid data members specified in its input and output sets. The third statement (line 22) uses an internal function $newTask()$ to construct a new, schedulable task entity, visible to the task scheduler, by coupling the guard thread and the task-execution function together. If the task contains start or end valves (e.g., $t2$), we make a new instance of the valves before we construct the task. That is, in lines 24 and 25, we create two new instances of ValveCT ($v1$ and $v2$) with the parameters of the valve. The valve will be passed to the task in line 28 by setting the parameters of its guard.

Thirdly, since some of the tasks take count as a parameter, we slightly change the variable type for the count. When we pass the count to a method, we also pass the address of the memory where we store the count value. We use "$ct.ct()$" to obtain the address of the memory where we store the value. Also, for each Fluid method that takes count as a parameter, we change the type from count to the corresponding type of pointer. For example, since ct is a count variable with the type of "int", we change its type to "int*" in line 9.

6 Runtime Support

To support the Fluid, our runtime system includes three interrelated components: *guard thread*, *Fluid states*, and *Fluid state machine*. Each task passes through specific states during its execution, as specified by the Fluid state machine. The execution of each task is controlled by its own guard thread. The guard thread is launched upon the initiation of a Fluid task and is terminated when the Fluid task finishes its execution.

6.1 State Machine for Fluid Tasks

As shown in Figure 5, a Fluid task is always in one of seven states: (**I**) initialization, (**CS**) start checking, (**R**) running, (**CE**) end checking, (**C**) completion, (**W**) waiting, and (**D**) dependence-stalled (waiting for antecedent tasks to update task inputs before re-executing). A Fluid task is initialized in **I** when the execution flow reaches its constructor within a *Region*() call. A separate guard thread is created by our framework to provide the guard functionality for each new task. This guard thread continuously checks the start valves before the task can start its execution. Once all valves are satisfied, the task is eligible to run (**R**). When a Fluid task finishes its execution, it will be in the **CE** state where end conditions are checked. The transition from **CE** to **C** can be triggered by any of the following three conditions: i) a non-empty set of end valves all return true (i.e., this task is a leaf node and the output satisfies all quality measure); ii) all inputs to this task were, prior to the start of the execution, computed with non-Fluid values (which means that the output of this task is already identical to that which would have been produced in a precise execution); or iii) all the descendant Fluid tasks of this task in the task graph are in the completion state. If none of these three conditions are satisfied, the Fluid task transitions to **W**, waiting for further signals and preparing for potential re-execution. During the stay in the **W** state, three state transitions can potentially take place, depending on the signal received. First, a transition to the **C** state (①) can be triggered by the completion signals received from the descendant tasks, indicating that all the descendant tasks are in the completion state and there is no need to re-execute this task. Second, the task will transition to the running state (②) when it receives an input data update signal from its parent tasks. Since the parent tasks have generated more accurate input data, the task will use the updated data to re-execute and itself generate new outputs. It might happen that the task starts its re-execution and then all of its descendant tasks send completion signals. To address such cases, we also implemented an early termination mechanism to stop the task's execution and relinquish the occupied resources for other waiting tasks. Third, the task will transition to the **D** state (③) if it receives a request signal from *any* of its child tasks. This is because, when the request signal is received, the task needs to re-execute to generate more accurate results for its descendant tasks. However, the task should not start its re-execution until a more accurate input data is produced by its parent tasks. Therefore, it enters the **D** state to wait for its parents to send signals indicating that a more accurate data has actually been generated. Upon receiving this signal, it transitions to the **R** state for re-execution (④). In the current implementation, the Fluid data are shared between producer and consumer tasks within the same memory locations.

We want to emphasize that our task re-execution mechanism helps users automatically adjust/modulate valve thresholds. Specifically, the programmer specifies the *minimum* thresholds. Our runtime would then dynamically adjust the threshold between the programmer-specified threshold and full serialization, since increasing the threshold increases the accuracy of the output. More specifically, we re-execute the consumer with more precise input data if it fails the quality check. For example, consider a dependency-chain (A1->A2->A3). If A3 fails the quality check, it re-executes with more precise data from A2 (since A2 keeps executing while A3 executes). Similarly, if A3's quality check fails even when executing after A2 finishes, A2 re-executes with a more precise version of input from A1. All tasks keep getting triggered until the final task meets the quality-check, although a task will not re-execute if its input is the result of fully serialized ancestor tasks. For example, after A1 finishes its execution, this means it has generated complete/precise data (as it would generate in a non-Fluid execution), and never re-executes. If A2 uses that complete data for one round of re-execution, after this round finishes, A2 has also generated its complete (most accurate) data. In the worst-case, all tasks run with complete/precise data and finally generate precise output, which would be identical to the output of the corresponding non-Fluid program, and the quality check is overridden.

6.2 Fluid Region Scheduling

Recall from the discussion in Section 3 that both the input and output of a Fluid region are *non-Fluid* data. Therefore, we adopt a *first-come-first-serve* policy to schedule the regions. Specifically, a Fluid region starts its execution when i) its input data are ready, and ii) there are available execution resources. One may envision more sophisticated scheduling mechanisms for multiple regions based on the region characteristics, such as shortest-job-first scheduling [72], or data locality concerns. However, we found that FCFS performs well enough for our tested applications, and postpone the exploration of more sophisticated scheduling strategies to future work.

7 Evaluation

We evaluate our proposed Fluid framework using eight applications (or computational kernels). We want to emphasize that our goal in this section is *not* to defend a particular fluidization strategy for a given benchmark, and we do not claim that the particular approach used in each benchmark to fluidize it is the best one.[3] Instead, our goal is to demonstrate that Fluid computation can be used to encode various approximate computing opportunities across different applications, and that doing so can lead to reductions in execution times

[3]We postpone automatic fluidization of non-Fluid applications to a future study. The application programs in this work have been hand-fluidized.

Table 2. Characteristics of our fluidized workloads.

Application	Producer	Consumer	How to fluidize	tot/pragma (app)	tot/pragma (region)
K-means [90]	Assign Cluster for each pixel	Re-calculate the cluster centers	Start calculating the center before all pixels are assigned a cluster	489 / 12 / 2.5%	146 / 11 / 7.5%
Bellman-ford (BF) [78]	One relax iteration	Next relax iteration	Start next relax iteration before relax all vertices	188 / 13 / 7.0%	85 / 13 / 15.0%
Graph Coloring (GC) [82]	Find local maximum vertex	Color the vertices	Coloring selected nodes before find out all local maximum vertices	307 / 8 / 2.6%	118 / 7 / 5.9%
Edge Detection (ED) [90]	Noise removal filter	Edge detection	Start detecting edges with noisy images	245 / 9 / 3.7%	128 / 8 / 6.2%
FFT [90]	Sin/Cos value	Calculate FFT	Calculate FFT with approximate sin/cos values	459 / 17 / 3.7%	180 / 16 / 8.9%
DCT [90]	Cos value	calculate sum	Calculate sum with approximate cos values	325 / 14 / 4.3%	246 / 13 / 5.3%
Neural Network (NN) [52, 79]	Previous layer	Next layer	Start next layer before all feature calculated	427 / 17 / 4.0%	263 / 16 / 6.1%
MedusaDock (MD) [39, 93]	Calculate Docking energy of poses	Select lowest energy poses	Start selecting poses when the a portion of the poses are processed	200 / 9 / 4.5%	148 / 8 / 5.4%

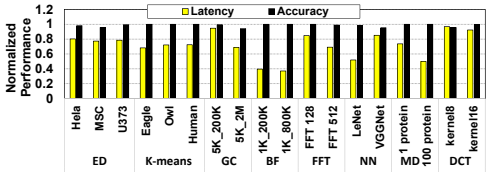


Figure 6. Fluidized accuracy and latency, *normalized* to original version.

without much loss in accuracy. Further, we do not exhaustively tune the parameters associated with each valve – the parameters are selected illustratively rather than optimally.

7.1 Applications and Methodology

Applications: We use a 20-core Intel Xeon-based platform with 32GB memory in our experiments. We evaluate Fluid on eight applications from different areas considering both *intra-kernel* and *inter-kernel* level fluidization. Table 2 lists the important characteristics of the applications used in this study. For each application, we show the corresponding producer/consumer tasks and how we fluidize the application. We also list the number of pragmas in the Fluid version of the code and its ratio to the entire program lines. At a high level, we can divide our applications into 4 classes, based on the type of the task graphs they possess. As shown in Figure 1(a), the first class of applications only has two tasks in a single Fluid region, where the first task is the *producer* and the second one is the *consumer*. This class includes i) edge detection where we first remove the noise on the image and then catch the edges on the image, and ii) MedusaDock [93], which first calculates energy for each docking pose (in the context of drug discovery) and then selects several lowest energy poses. The second class of applications contains multiple Fluid regions, each containing a producer task and a consumer task. This class includes K-means and Graph coloring, both iterating over invocations of a producer-consumer pair. The third class includes Neural Network (NN) and Bellman-Ford which contain multi-task chains within a single Fluid region – within the Fluid region, all but the first and last tasks in the chain are both an eagerly invocable consumer and a producer of data for another dependent task. For the last class of applications, the task dependency graph features either or both of multi-consumer (two or more child tasks with independent start conditions on the same data structure) and multi-producer (a task dependent on multiple valves, each relating to updates to a different data structure) topologies. This class includes FFT and DCT where we calculate the value of cos/sin functions. K-means, Edge detection, FFT and DCT are implemented based on Axbench [90]. The graph coloring is based on Big-graph with the implementation in [82]. We used our own implementations for Bellman-ford and NN, and used the implementation of MedusaDock from [39].

Inputs: K-means uses three input images with different pixel diversities. For Bellman-Ford and graph coloring, we generate input graphs with different sizes and densities to study the input sensitivity of our framework. For edge detection, we choose three EM images from a publicly-available dataset [56]. For FFT and DCT on the other hand, we generate input vectors/tensors with different sizes. For NN, we use Mnist [52] for LeNet [52] and ImageNet [45] for VGGNet [79] as the testing data with different networks, and we use pdbbind [89] to evaluate MedusaDock.

Error Metrics: For K-means, we compute the squared Euclidean distance to the cluster centroid for each pixel and sum those distances together. For Bellman-Ford, we first normalize the path length for each destination vertex to the actual shortest path, and then compute the average error. For graph coloring, the error metric we employ is the graph's spectral number, *normalized* to the spectral number produced by the original (already approximate) algorithm. For edge detection, we use PSNR (Peak Signal to Noise Ratio) as our error metric. Additionally, we use normalized MSE (Mean Squared Error) of the output as the error metric for FFT and DCT. Finally, We use prediction accuracy as the error metric for NN and MedusaDock. When comparing the fluidized version of an application to the baseline, we calculate the normalized accuracy as: $ABS(fluid\_ErrorMetric - baseline\_ErrorMetric)/baseline\_ErrorMetric$

7.2 Fluid Execution Time and Accuracy

Figure 6 plots the normalized latency and accuracy results for all applications. Here, the start condition of the consumer task uses the percentage valve, which means that the dependent tasks start their executions when a certain fraction

Table 3. Runtime statistics.

APP	TASK	Average number access to the state						Average time stay in the state / us					
		Init	StartCheck	Running	EndCheck	Wait/Stall	Complete	Init	StartCheck	Running	EndCheck	Wait/Stall	Complete
Edge Detection	Gaussian	1	1	1	1	0	1	8	55	393146	81	0	23
	Sobel	1	1	2	2	1	1	6	299329	83743	32	10502	41
Bellmanford	Relax	1	1	4.62	4.62	4.5	1	26.19	6377668.54	1126602.08	84.07	19702.16	75.12
K-means	AssignCluster	1	1	1	1	0	1	3.15	116.46	76414.33	147.41	0	63.41
	Recenter	1	1	1	1	0	1	3.23	25945.36	25945.36	33.73	0	24.26
Graph Color	Select	1	1	1	1	0	1	3.73	113.24	22600.75	58.07	0	31.59
	Coloring	1	1	1.14	1.14	1.01	1	3.87	11404.89	146	121.39	11119.27	57.91
FFT	FFTsin	1	1	1	1	0	1	14.5	866.75	1388.25	210	0	135.2
	FFTcos	1	1	1	1	0	1	6.75	2102.75	1122.25	244	0	162.5
DCT	cos	1	1	1	1	0	1	1.70	67.81	157.07	104.75	0	60.48
	sum	1	1	1	1	0	1	1.89	122.77	160.45	64.13	0	39.80
NN	layer1	1	1	1	0	0	1	31	77	66384002	0	0	211
	layer2	1	1	1	1	1	1	7	64845464	32214	44	1506357	30
	layer3	1	1	1	1	1	1	426	64877226	1500895	80	5130	95
	layer4	1	1	1	1	0	1	3	66377938	5165	40	0	154
Medusa	medusa_dock	1	1	1	1	0	1	5.46	74.97	11122276.46	71.47	0	53.76
	select pose	1	1	1	1.51	0.51	1	7.27	2283527.73	228.03	40.15	17563000.20	45.33

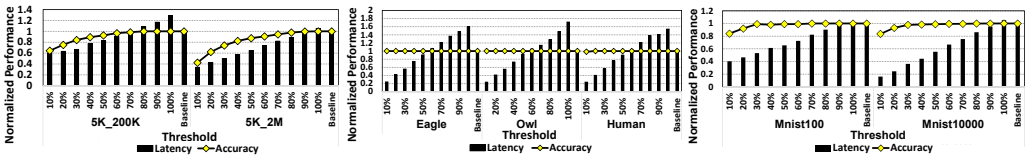
of the payload of the producer task has completed. We observe that the fluidized version of Bellman-Ford matches the precise output: Since, for non-pathological graphs, each vertex tends to only update its neighbors very few times [27], skipping some of the execution does not affect the final result in any significant way. Fluidized K-means also exhibits accuracy similar to that of precise execution: In K-means, it is known that most pixels are unlikely to change their cluster membership after the first few iterations [46]. Note that the benefit of Fluid for K-means comes from overlapping the execution of the producer (assign cluster) and the consumer (re-calculate centroid) tasks, not from reducing the number of epochs. For graph applications (Bellman-Ford and Graph Coloring), we observe that the Fluid framework achieves better speedups on dense graphs (5K_2M/1K_800k) than on sparse (5K_200K/1K_200K), as dense graphs require more computation (in the original, non-Fluid version) and, consequently, the fluidized version can skip more computations. Similarly, for FFT, DCT, and MedusaDock, larger input sizes lead to better results than smaller input sizes as the payload for large input vectors is comparatively greater than for a small input vector. For NN, LeNet achieves better performance than VGGNet since approximation on small datasets/networks has less impact on accuracy. On average, Fluid brings 22.2% execution time improvements.

7.3 Overhead and Sensitivity Profiling

To provide deeper insights into fluidized execution, we explore the sensitivity to valve parameters, valve type and producer/consumer task workload, and characterize the overheads for Fluid tasks. We evaluate three of our applications and sweep the start valve threshold to observe the changing latency-accuracy tradeoffs. After that, we evaluate the two different valve types on two of our applications to study the performance with *different types of valves*. Additionally, for one of the applications, we change the algorithms used in the producer and consumer tasks to show that our framework works on diverse workload chains. Finally, we provide statistics revealing, for each task of each application, how frequently each runtime state is visited during Fluid execution and how much time is spent in each state. We also summarize the total overheads during the transitions between the different states of the state machine for each application.

Valve threshold: Figure 7 shows the results for sensitivity study on changing the start valve thresholds. One can make the following observations from these results. First, as the threshold value decreases, the execution time reduces for all applications, and the accuracy drops for two applications (GC and NN). For K-means, the accuracy is not sensitive to the threshold. Second, with larger inputs, the performance gains are more sensitive to the threshold modulation (e.g., 5K_2M for GC and Mnist10000 for NN). This is because the workload in each task when using the large input is heavier than that when using the small input, and therefore, task execution time is more significant compared to the task launch overhead. Third, for all applications, the programmer may find several operation points with a significant speedup boost without much accuracy drop.

Valve type: Figure 8 shows the performance of MedusaDock and K-means with two different valve types (*percentage* and *convergence* valves). For both applications, the percentage valve returns true when a certain fraction of the producer task is complete. For MedusaDock, the convergence valve returns true when the producer cannot find a lower energy pose during certain iterations. For K-means, we run both the original and Fluid versions with the same number of epochs. For the baseline evaluation, we train K-means until convergence. For the Fluid version, we run the same number of epochs and observe the accuracy drop. The convergence valve returns true when a certain fraction of pixels on the image have not changed their categories during last several iterations. From these results, one can observe that the proposed Fluid framework can work along with different



(a) Result for Graph Coloring. **(b)** Result for K-means image. **(c)** Result for NN.

Figure 7. Latency and accuracy performance with different thresholds for the valve.

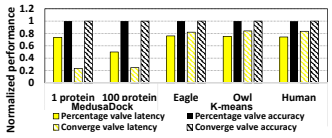


Figure 8. Percentage valve and converge valve performance for MedusaDock and K-means. All results normalized to non-Fluid version.

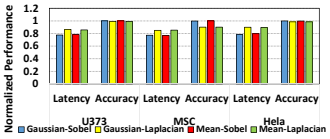


Figure 9. Different producer and consumer workload for Edge Detection. Results normalized to non-Fluid version of each workload chain.

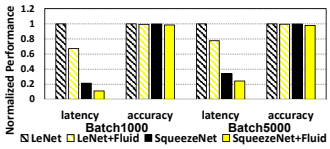


Figure 10. Combining Fluid and other approximation techniques (Fluid atop Squeezenet approximation of LeNet).

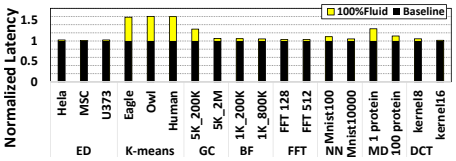


Figure 11. Overhead of the Fluid framework.

types of valves based on the application. For example, MedusaDock prefers the convergence valve since the lowest pose energy will be converged at an early stage for many proteins, whereas K-means is more compatible with the percentage valve because it will take more time for stability checking.

Different workload chains: Recall from Table 2 that the producer task filters noise from the image while the consumer task detects edges on the de-noised image with a gradient filter. Here, we implement two noise removal filters (Gaussian and Mean) and two gradient filters (Sobel and Laplacian) for Edge Detection. We show the results for all four (two by two) workload possibilities in Figure 9. One can observe that Sobel achieves higher latency benefits than Laplacian. This is mainly because Laplacian run faster than Sobel filter. As a result, the execution overlap between producer and consumer (which is the latency saving) contributes more to the overall execution time. On MSC images, the accuracy of Laplacian is more sensitive to the noise filtering since this input contains more noise than the others.

Runtime Overhead and Scheduling Statistics: To investigate the overheads involved in our framework, we set **all** start valve thresholds to 100%, i.e., the producer task will execute the same as in the non-Fluid version. The normalized latency is plotted in Figure 11. The yellow bar indicates how much the framework overhead contributes to the total application execution time. We find that the overhead is only significant in K-means, Graph-Coloring (GC), and MedusaDock (MD). In general, K-means and GC launch many more threads than the other applications, and the thread launching overhead is significant in those two applications. For MedusaDock, the task performs heavy disk and memory accesses (100GB), and the memory/disk access issue becomes more serious in the parallel execution scenario.

Table 3 presents statistics describing the task state machines (see Figure 5 in Section 6). Specifically, for each invocation of the task, it shows i) the number of times the task visits each state and ii) the time spent in each state. From these results, one can make the following observations: i) Each task accesses the Init, StartCheck and Complete states only once, whereas the Execution, Endcheck and Wait/Depend states are visited multiple times by some tasks due to end-valve (quality) failures and subsequent re-execution; ii) Non-root tasks spend long time in the StartCheck state, indicating some latency in valve checking; iii) For all but three tasks, the re-execution overheads are small. In Bellman-Ford, we invoke the same task multiple times, which generates a long task chain. Some internal tasks re-execute several times, and with high variance, due to chained re-execution on any quality-function failure. In Edge Detection, using a 40% threshold for the start valve, we fail in the quality function check. In MedusaDock, we do not allow pose selection to start if we only check pose energy a few times, to guarantee the software invest on enough poses. However, around 51% proteins fail this check; and finally, iv) In NN, the first layer does not finish even when the last layer has finished. Hence, we do early termination for the first layer. The second and third layers stall in the Wait state since the fourth layer has not finished yet. They need to wait for notification of whether re-execution is required or not.

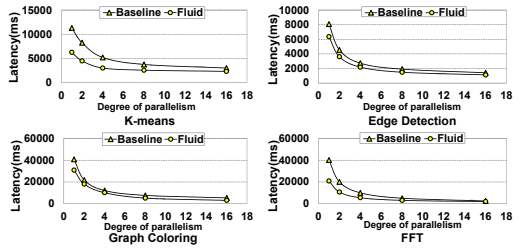


Figure 12. Fluid results of multi-threading applications.

7.4 Combining Fluid with Other Approximations

The Fluid framework is general and can be applied in conjunction with many other approximate computing strategies. To demonstrate this and explore the marginal utility of Fluid for already approximate codes, we use Lenet [52] as an example application and Mnist as input dataset, with different batch sizes. As shown in Figure 10, we first fluidize Lenet and achieve about 28% execution time saving. Next, we consider Squeezenet [36], which can be considered an approximate version of Lenet (Squeezenet is actually an approach for accelerating 3 by 3 convolution kernels). Squeezenet achieves 72% latency reduction over Lenet. Finally, we fluidize Squeezenet and achieve around 82% execution time saving without much accuracy drop. This small exercise clearly shows that our Fluid framework can be used on original (fully accurate) applications as well as on their approximate counterparts.

7.5 Fluidizing Multithreaded Codes

We evaluate Fluid on four applications (K-means, Edge detection, Graph Coloring and FFT). Figure 12 plots the performance results. We can observe that fluidization benefits the multithreaded K-means significantly when the degree of parallelism is low, while the magnitude of improvement is reduced in higher degrees of parallelism. On the other hand, in edge detection and graph coloring, Fluid achieves consistent latency reductions across the different degrees of parallelism. In K-means on the other hand, we have multiple Fluid regions in the entire program, which collectively generate many work-threads and guard-threads. As the degree of parallelism is increased, the workload for each thread decreases, and the overheads brought by these work- and guard-threads become more pronounced in the multithreaded versions. In FFT, when the degree of parallelism becomes 16, we almost reach the limits of the underlying compute resource (CPU cores). Further, our Fluid framework achieves 22.9%, 20.6%, 43.4% and 21.6% latency reductions, respectively, for K-means, Edge Detection, Graph Coloring and FFT, when using 16 threads, which means Fluid parallelism is complementary to the conventional parallelism based on multithreading.

8 Planned Future Work

Accommodating dynamic task-graphs. Our current implementation does not handle all cases required for dynamic task graph execution, including producer early-termination with non-fixed consumer count. We therefore present results limited to regions where the nodes (tasks) and edges (dependencies) between the tasks in a Fluid region can be fully specified *before* the task-graph starts execution. Extending the Fluid framework to dynamically generate the task-graph and the Fluid runtime scheduler to robustly process dynamic condition sets during runtime is part of ongoing work.

Integrating Fluid with existing frameworks. Currently, the user has to structure the program in a graph style which can be compiled by Fluid. We plan to integrate our framework with other existing task-graph frameworks such as Intel TBB [43], OneAPI [37], and OpenCilk [76]. This can be achieved by either reimplementing Fluid's features on top of these other frameworks, or by providing Fluid support for existing frameworks. We believe such integration can reduce the effort for a user to rewrite applications in a Fluid style.

Exploring other programming language paradigms. Our current implementation employs OOP primarily to isolate the Fluid and Non-Fluid parts of the program, where only Fluid objects have access to Fluid data. We believe Aspect-Oriented Programming (AOP) paradigm is another option using which Fluid can be implemented. Hence, we plan to explore implementing Fluid on top of AOP as well. In such implementation, each major Fluid concept (e.g., data, task, valve) would typically be implemented as a module.

9 Related Work

Approximate computing: Recent works have focused on approximate computing [8, 31–33, 51, 58, 63, 70] that target applications for which one may accept less-than-exact output. At the software level, Bornholt et al. [12] suggested an "Uncertain" type as an abstraction for exposing uncertain data and leveraged runtime sampling and hypothesis tests to evaluate computation and conditionals. Carbin et al. [15] formalized and proposed programming language support for relaxed programming. Zhu et al. [95] investigated a novel computation model for accuracy-aware transformations, and proposed a randomized optimization algorithm to approximate such computation model. StreamApprox [67] executes a given program with a sampled subset of input stream data based on the resource budget. Sampson et al. [74] proposed probabilistic assertions using which the programmer can express probabilistic correctness properties. At the architecture level, Venkataramani et al. [87] investigated a vector processor architecture, QUORA, that balances energy consumption and result quality. Esmaeilzadeh et al. [26] studied an architecture that augments a traditional processor with a neural processing unit (NPU). Topaz [1] proposed a task-based language support for the applications running on the platforms which generate arbitrarily inaccurate results. Several recent works [16, 60] allow users to reason about the behavior of the applications running on approximate computing platforms. At the circuit level, Kulkarni et al. [50]

designed a power efficient 2 × 2 approximate multiplier, and Venkataramani et al. [88] proposed Substitute-And-SIMplIfy (SASIMI) for approximate circuits.

Eager execution: There have been a lot of works in the past based on eager execution with various approaches proposed for predication and speculative execution [21, 40, 64, 65, 80, 85, 92]. Rinard et al. [71] focused on solving processor idleness by terminating the threads which run longer than most other threads. A distortion model helps the user to evaluate the effect of terminating some of the application threads early. Lin et al. [55] proposed both selective push-pull and statistical barriers primitives to eliminate the synchronization requirements for the mainstream applications. Klauser et al. [47] explored selective eager execution, which overcomes mis-speculation penalties by executing all possible paths of the branch. August et al. [6] proposed explicitly parallel instruction computing (EPIC), which uses a compiler to perform control speculation, data dependence speculation, and predication. Paszke et al. [66] employed eager execution in deep learning frameworks, and Lim et al. [54] evaluated both eager and lazy solutions for the set cover problem.

Concurrent object-oriented programming (COOP): Prior works have explored OOP with concurrency [19, 20, 30]. For example, Agha [3], Cantor [5], Pony [22], and P [25] employed the actor model as a framework for concurrent systems. Lee et al. [53] presented an object-oriented parallel programming paradigm, where programmers can build distributed structures. Yonemwa et al. proposed a COOP model called ABCL/1 [94] for modeling and concurrent systems. It incorporated three message-passing patterns. COOL [18] is an object-based programming language model for parallel computing. Its runtime system automatically manages task creation and scheduling.

Continuations/coroutines: Several compilers employ continuation passing style (CPS) terms as their intermediate representation for programs [4, 73, 77, 86]. All procedures in CPS take a *continuation* that represents the rest of the computation. Continuation is conceptually a special case of Fluid computation, where the rest of computation starts when the current procedure is *fully* evaluated. The coroutine is a useful language abstraction for describing a consumer-producer relationship, where the producer may "yield" to the consumer whenever new items are created. In the simplest case, the producer may yield for every new version of data.

Parallel frameworks Recent works studied on increasing the degree of thread/task-level parallelism by removing select data dependencies. STATS [24] groups tasks by input data and generates temporary inputs with auxiliary code. However, it mainly focuses on streaming benchmarks. ALTER [84] allows programmers to annotate a program to relax data dependencies and increase the parallelism of loop execution. However, it is limited to instruction-level approximation. HELIX-UP [14] presents compiler and runtime support for improving the scalability of the parallel code by relaxing dependencies. There also exist several standard frameworks for thread-level parallel programming [11, 17, 49].

Shared memory system for chaotic access: SAM [75] provides a shared memory access system for distributed machines. This system supports approximate computations if the task can access the object before it could be fully computed. It implements a global name space and caching which is essential for chaotic memory accessing and parallel. SAM primitives are intended to support different systems that provide chaotic accesses [9, 35, 59, 68].

Value prediction: Recent works have proposed to predict approximate values via architectural support [58, 81, 83, 91]. Most of such approaches focus on value prediction in the memory hierarchy. More specifically, dependent instructions can start their execution with predicted data values, instead of waiting for long dependency chains to be resolved.

Lock removal: Recent approaches have targeted removing synchronizations in a selective fashion. Such approaches include adaptive locking [38], speculative lock elision [44], relaxing synchronizations by trading off performance for quality whenever necessary [69], and automatically exploring parallelization opportunities with a given output error bound [62]. Some other studies remove locks at the beginning of the concurrent program with data race, and then add the lock back when the statistical accuracy guarantee is satisfied [61] or the concurrency bugs/violations are fixed [41].

Self-adaptive QoS systems: Self-adaptive systems [23, 28, 48] dynamically tune user-specified parameters in a program to meet the requirements on output quality. However, they focus primarily on signal-processing applications, with well-studied kernels while Fluid targets general-purpose applications. Having said that, such self-adaptive systems have developed robust mechanisms that control multiple knobs over time, which we can certainly adopt in Fluid.

How do we differ? Compared to prior approximate computing works focusing either on particular application domains or on realizing approximation in select system layers, Fluid is more general, and represents in any application that employs a producer-consumer execution pattern. Further, by integrating compiler support with runtime management, our Fluid framework is able to improve application performance without significantly compromising output quality. The thread-level-parallelism works are orthogonal to Fluid since the task concept employed in our Fluid region can be a thread of a multi-threaded application. Compared to the value prediction-based efforts, Fluid does not require any new hardware support, and it can proceed with eager values for code regions that are much larger than a single instruction. Finally, compared to the lock removal approaches that try to increase parallelism by eliminating synchronizations, Fluid gives the user more flexibility in controlling the start condition of a single task based on each dependent task.

10 Conclusions

This paper introduces Fluid computation, a new approach to eager computation based on Fluid classes that can expose intermediate versions of select (Fluid) data members to their Fluid member methods, allowing the relaxation of traditional data dependency ordering constraints. It presents the language extensions needed to support Fluid computations, automated compiler support to translate a program augmented with Fluid pragmas into a conventional C++ code consumable by any C++ compiler, and an accompanying runtime system that controls task scheduling for Fluid methods. Our experimental results show that Fluid computation can be an effective way of harnessing the benefits of approximate computing in a disciplined fashion.

Acknowledgments

The authors sincerely thank Dr. Martin Rinard for shepherding the paper and the reviewers for their constructive feedback. This work is supported by NSF grants #1908793, #1629915, #1629129, #1763681, #2028929, #2008398, #1931531, and startup funding from the University of Pittsburgh.

References

[1] Sara Achour and Martin C Rinard. 2015. Approximate computation with outlier detection in topaz. *Acm Sigplan Notices* 50, 10 (2015), 711–730. https://doi.org/10.1145/2814270.2814314

[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[3] Gul Agha. 1990. Concurrent Object-oriented Programming. *Commun. ACM* 33, 9 (Sept. 1990), 125–141. https://doi.org/10.1145/83880.84528

[4] Andrew W Appel. 2007. *Compiling with continuations.* Cambridge University Press.

[5] W. C. Athas and N. J. Boden. 1988. Cantor: An Actor Programming System for Scientific Computing. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-based Concurrent Programming.* https://doi.org/10.1145/1806596.1806620

[6] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen-mei W. Hwu. 1998. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *ISCA.* https://doi.org/10.1109/ISCA.1998.694777

[7] D. P. Bertsekas and D.A Castañon. 1999. Rollout algorithms for stochastic scheduling problems. In *Journal of Heuristics.* https://doi.org/10.1023/A:1009634810396

[8] Filipe Betzel, Karen Khatamifard, Harini Suresh, David J Lilja, John Sartori, and Ulya Karpuzcu. 2018. Approximate communication: Techniques for reducing communication bottlenecks in large-scale parallel systems. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1. https://doi.org/10.1145/3145812

[9] Roberto Bisiani and Alessandro Forin. 1988. Multilanguage parallel programming of heterogeneous machines. *IEEE Trans. Comput.* 37, 8 (1988), 930–945. https://doi.org/10.1109/12.2245

[10] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, and Greg Henry. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* (2002), 135–151. https://doi.org/10.1145/567806.567807

[11] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69. https://doi.org/10.1145/209937.209958

[12] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: A First-Order Type for Uncertain Data. In *ASPLOS.* https://doi.org/10.1145/2541940.2541958

[13] L. Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proc. 19th Int. Conf. Comput. Statist.* https://doi.org/10.1007/978-3-7908-2604-3_16

[14] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing program semantics to unleash parallelization. In *CGO.* IEEE. https://doi.org/10.1109/CGO.2015.7054203

[15] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. *ACM SIGPLAN Notices* 47, 6 (2012), 169–180. https://doi.org/10.1145/2345156.2254086

[16] Michael Carbin, Sasa Misailovic, and Martin C Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. *ACM SIGPLAN Notices* 48, 10 (2013), 33–52. https://doi.org/10.1145/2544173.2509546

[17] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java.* 51–61. https://doi.org/10.1145/2093157.2093165

[18] Rohit Chandra, Anoop Gupta, and John L Hennessy. 1994. COOL: An object-based language for parallel programming. *Computer* 27, 8 (1994), 13–26. https://doi.org/10.1109/2.303616

[19] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting Actor Programming in C++. *Comput. Lang. Syst. Struct.* 45, C (April 2016), 105–131. https://doi.org/10.1016/j.cl.2016.01.002

[20] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. 2013. Native Actors: A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control.* https://doi.org/10.1145/2541329.2541336

[21] George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. In *ISCA.* https://doi.org/10.1145/279361.279378

[22] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control.* https://doi.org/10.1145/2824815.2824816

[23] Rogério De Lemos, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, et al. 2017. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances.* Springer, 3–30. https://doi.org/10.1007/978-3-319-74183-3_1

[24] Enrico A Deiana, Vincent St-Amour, Peter A Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional parallelization of nondeterministic applications. In *ASPLOS.* 432–447. https://doi.org/10.1145/3173162.3173181

[25] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/2499370.2462184

[26] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *MICRO.* 449–460. https://doi.org/10.1109/MICRO.2012.48

[27] Duan Fanding. 1994. A Faster Algorithm for Shortest-Path - SPFA. *Journal of Southwest Jiaotong University* 2 (1994).

[28] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2015. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 13–24. https://doi.org/10.1145/2786805.2786833

[29] K. Ganesan, J. San Miguel, and N. Enright Jerger. 2019. The What's Next Intermittent Computing Architecture. In *HPCA*. 211–223. https://doi.org/10.1109/HPCA.2019.00039

[30] Sashikumaar Ganesan, Volker John, Gunar Matthies, Raviteja Meesala, Shamim Abdus, and Ulrich Wilbrandt. 2016. An object oriented parallel finite element scheme for computations of PDEs: Design and implementation. *CoRR* abs/1609.04809 (2016). https://doi.org/10.1109/HiPCW.2016.023

[31] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3296979.3192400

[32] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *ASPLOS*. https://doi.org/10.1145/2694344.2694351

[33] Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi. 2014. Globally Precise-restartable Execution of Parallel Programs. In *Proceedings of the 35th Conf. on Programming Language Design and Implementation*. https://doi.org/10.1145/2594291.2594306

[34] J. Han and M. Orshansky. 2013. Approximate Computing: An Emerging Paradigm For Energy-Efficient Design. In *IEEE ETS*. https://doi.org/10.1109/ETS.2013.6569370

[35] Phillip W Hutto and Mustaque Ahamad. 1990. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings., 10th International Conference on Distributed Computing Systems*. IEEE Computer Society, 302–303. https://doi.org/10.1109/ICDCS.1990.89297

[36] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[37] Intel. 2020. Intel oneAPI Toolkits (Beta). https://software.intel.com/content/www/us/en/developtools/oneapi.html.

[38] Bashima Islam, Faysal Hossain Shezan, and Rifat Shahriyar. 2016. High Performance Approximate Computing by Adaptive Relaxed Synchronization. In *HPCC/SmartCity/DSS 2016*. IEEE, 1204–1210. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0168

[39] Huaipan Jiang, Mengran Fan, Jian Wang, Anup Sarma, Shruti Mohanty, Nikolay V Dokholyan, Mehrdad Mahdavi, and Mahmut T Kandemir. 2020. Guiding Conventional Protein–Ligand Docking Software with Convolutional Neural Networks. *Journal of Chemical Information and Modeling* 60, 10 (2020), 4594–4602. https://doi.org/10.1021/acs.jcim.0c00542

[40] Daniel A. Jiménez and Calvin Lin. 2002. Dynamic Branch Prediction with Perceptrons. In *HPCA*. https://doi.org/10.1109/HPCA.2001.903263

[41] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. https://doi.org/10.1145/1993498.1993544

[42] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2020. Aloe: verifying reliability of approximate programs in the presence of recovery mechanisms. In *CGO*. 56–67. https://doi.org/10.1145/3368826.3377924

[43] Alexei Katranov and Alexey Kukanov. 2016. Intel® threading building block (Intel® TBB) flow graph as a software infrastructure layer for OpenCL™-based computations. In *Proceedings of the 4th International Workshop on OpenCL*. 1–3. https://doi.org/10.1145/2909437.2909446

[44] S Karen Khatamifard, Ismail Akturk, and Ulya R Karpuzcu. 2017. On Approximate Speculative Lock Elision. *IEEE Transactions on Multi-Scale Computing Systems* 4, 2 (2017), 141–151. https://doi.org/10.1109/TMSCS.2017.2773488

[45] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. 2011. Novel dataset for fine-grained image categorization: Stanford dogs. In *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*, Vol. 2. Citeseer.

[46] Orhan Kislal, Piotr Berman, and Mahmut Kandemir. 2012. Improving the performance of k-means clustering through computation skipping and data locality optimizations. In *Proceedings of the 9th conference on Computing Frontiers*. ACM. https://doi.org/10.1145/2212908.2212951

[47] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald. 1998. Selective eager execution on the PolyPath architecture. In *ISCA*. https://doi.org/10.1109/ISCA.1998.694785

[48] Jeff Kramer and Jeff Magee. 2007. Self-managed systems: an architectural challenge. In *Future of Software Engineering*. IEEE, 259–268. https://doi.org/10.1109/FOSE.2007.19

[49] Alexey Kukanov and Michael J Voss. 2007. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 4 (2007). https://doi.org/10.1535/itj.1104.05

[50] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. 2011. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *Proc. of International Conference on VLSI Design*. https://doi.org/10.1109/VLSID.2011.51

[51] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the 37th Conf. on Programming Language Design and Implementation*. https://doi.org/10.1145/2908080.2908087

[52] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[53] J. K. Lee and D. Gannon. 1991. Object Oriented Parallel Programming: Experiments and Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/125826.105186

[54] Ching Lih Lim, Alistair Moffat, and Anthony Wirth. 2014. Lazy and Eager Approaches for the Set Cover Problem. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference*.

[55] Tsung-Han Lin, Stephen J Tarsa, and HT Kung. 2013. Parallelization primitives for dynamic sparse computations. In *5th {USENIX} Workshop on Hot Topics in Parallelism (HotPar 13)*.

[56] Martin Maška, Vladimír Ulman, David Svoboda, Pavel Matula, Petr Matula, Cristina Ederra, Ainhoa Urbiola, Tomás España, Subramanian Venkatesan, Deepak M.W. Balak, et al. 2014. A benchmark for comparison of cell tracking algorithms. *Bioinformatics* 30, 11 (2014), 1609–1617. https://doi.org/10.1093/bioinformatics/btu080

[57] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The Bunker Cache for Spatio-value Approximation. In *MICRO (MICRO-49)*. IEEE Press, Article 43, 12 pages. https://doi.org/10.1109/MICRO.2016.7783746

[58] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *MICRO*. https://doi.org/10.1109/MICRO.2014.22

[59] Ronald G Minnich and David J Farber. 1993. Reducing host load, network load and latency in a distributed shared memory. *Technical Reports (CIS)* (1993), 459.

[60] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. *ACM Sigplan Notices* 49, 10 (2014), 309–328. https://doi.org/10.1145/2714064.2660231

[61] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2010. *Automatic parallelization with statistical accuracy bounds*. Technical Report. MIT CSAIL/EECS.

[62] Sasa Misailovic, Stelios Sidiroglou, and Martin C Rinard. 2012. Dancing with uncertainty. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*. https://doi.org/10.1145/2414729.2414738

[63] Subrata Mitra, Manish K Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-aware optimization in approximate computing. In *CGO*. IEEE, 185–196. https://doi.org/10.1109/CGO.2017.7863739

[64] Andreas Moshovos and Gurindar S. Sohi. 1999. Read-After-Read Memory Dependence Prediction. In *MICRO*. https://doi.org/10.1109/MICRO.1999.809455

[65] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. 1992. Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS*. https://doi.org/10.1145/143365.143490

[66] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. *NIPS 2017 Autodiff Workshop* (2017).

[67] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: approximate computing for stream analytics. In *18th ACM/IFIP/USENIX Middleware Conference*. 185–197. https://doi.org/10.1145/3135974.3135989

[68] Umakishore Ramachandran and M Yousef A Khalidi. 1991. An implementation of distributed shared memory. *Software: Practice and Experience* 21, 5 (1991), 443–464. https://doi.org/10.1002/spe.4380210503

[69] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with relaxed synchronization. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*. 41–50. https://doi.org/10.1145/2414729.2414737

[70] Haris Ribic and Yu David Liu. 2014. Energy-efficient work-stealing Language Runtimes. In *ASPLOS*. https://doi.org/10.1145/2654822.2541971

[71] Martin C Rinard. 2007. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 369–386. https://doi.org/10.1145/1297027.1297055

[72] Jia Ru and Jacky Keung. 2013. An empirical investigation on the simulation of priority and shortest-job-first scheduling for cloud-based software systems. In *2013 22nd Australian Software Engineering Conference*. IEEE, 78–87. https://doi.org/10.1109/ASWEC.2013.19

[73] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* 1, 2 (2015).

[74] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2594291.2594294

[75] Daniel J Scales and Monica S Lam. 1994. *An efficient shared memory layer for distributed memory machines*. Computer Systems Laboratory, Stanford University.

[76] Tao B Schardl, I-Ting Angelina Lee, and Charles E Leiserson. 2018. Brief announcement: Open cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 351–353. https://doi.org/10.1145/3210377.3210658

[77] Hashim Sharif, Prakalp Srivastava, Muhammad Huzaifa, Maria Kotsifakou, Keyur Joshi, Yasmin Sarita, Nathan Zhao, Vikram S Adve, Sasa Misailovic, and Sarita Adve. 2019. ApproxHPVM: a portable compiler IR for accuracy-aware optimizations. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 186. https://doi.org/10.1145/3360612

[78] Alfonso Shimbel. 1954. Structure in communication nets. In *Proceedings of the symposium on information networks*. Polytechnic Institute of Brooklyn, 119–203.

[79] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv* (2014).

[80] Samantika Subramaniam and Gabriel H. Loh. 2006. Store Vectors for Scalable Memory Dependence Prediction and Scheduling. In *HPCA*. https://doi.org/10.1109/HPCA.2006.1598113

[81] Mark Sutherland, Joshua San Miguel, and Natalie Enright Jerger. 2015. Texture cache approximation on GPUs. In *Workshop on Approximate Computing Across the Stack*.

[82] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *HPCA*. 649–660. https://doi.org/10.1109/HPCA.2017.14

[83] Bradley Thwaites, Gennady Pekhimenko, Hadi Esmaeilzadeh, Amir Yazdanbakhsh, Jongse Park, Girish Mururu, Onur Mutlu, and Todd Mowry. 2014. Rollback-free value prediction with approximate loads. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 493–494. https://doi.org/10.1145/2628071.2628110

[84] Abhishek Udupa, Kaushik Rajan, and William Thies. 2011. ALTER: exploiting breakable dependences for parallelization. In *Proc. of the 32nd Conference on Programming language Design and Impl.* https://doi.org/10.1145/1993498.1993555

[85] Augustus K. Uht, Vijay Sindagi, and Kelley Hall. 1995. Disjoint eager execution: an optimal form of speculative execution. In *MICRO*. https://doi.org/10.1109/MICRO.1995.476841

[86] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *MICRO*. IEEE Press, 42. https://doi.org/10.1109/MICRO.2016.7783745

[87] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *MICRO*. https://doi.org/10.1145/2540708.2540710

[88] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2013. Substitute-and-Simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits. In *DATE*. https://doi.org/10.7873/DATE.2013.280

[89] Renxiao Wang, Xueliang Fang, Yipin Lu, Chao-Yie Yang, and Shaomeng Wang. 2005. The PDBbind database: methodologies and updates. *Journal of medicinal chemistry* 48, 12 (2005), 4111–4119. https://doi.org/10.1021/jm048957q

[90] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2016. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test* 34, 2 (2016), 60–68. https://doi.org/10.1109/MDAT.2016.2630270

[91] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2016), 1–26. https://doi.org/10.1145/2836168

[92] Tse-Yu Yeh and Yale N Patt. 1992. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News* 20, 2 (1992), 124–134. https://doi.org/10.1145/146628.139709

[93] Shuangye Yin, Lada Biedermannova, Jiri Vondrasek, and Nikolay V Dokholyan. 2008. MedusaScore: an accurate force field-based scoring function for virtual drug screening. *Journal of chemical information and modeling* 48, 8 (2008), 1656–1662. https://doi.org/10.1021/ci8001167

[94] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. 1986. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices* 21, 11 (1986), 258–268. https://doi.org/10.1145/960112.28722

[95] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. 2012. Randomized accuracy-aware program transformations for efficient approximate computations. *ACM SIGPLAN Notices* 47, 1 (2012), 441–454. https://doi.org/10.1145/2103621.2103710