Parallel Gesture Recognition with Soft Real-Time Guarantees☆

Stefan Marr, Thierry Renaux, Lode Hoste, Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Abstract

Using imperative programming to process event streams, such as those generated by multi-touch devices and 3D cameras, has significant engineering drawbacks. Declarative approaches solve common problems but so far, they have not been able to scale on multicore systems while providing guaranteed response times.

We propose PARTE, a parallel scalable complex event processing engine that allows for a declarative definition of event patterns and provides soft real-time guarantees for their recognition. The proposed approach extends the classical Rete algorithm and maps event matching onto a graph of actor nodes. Using a tiered event matching model, PARTE provides upper bounds on the detection latency by relying on a combination of non-blocking message passing between Rete nodes and safe memory management techniques.

The performance evaluation shows the scalability of our approach on up to 64 cores. Moreover, it indicates that PARTE's design choices lead to more predictable performance compared to a PARTE variant without soft real-time guarantees. Finally, the evaluation indicates further that gesture recognition can benefit from the exposed parallelism with superlinear speedups.

Keywords: multimodal interaction, gesture recognition, Rete, actors, soft real-time guarantees, non-blocking, complex event processing, multicore

1. Introduction

In order to improve how users and computers interact, multi-touch input, gesture recognition, and speech processing are becoming common in consumer hardware. To power more natural user interfaces, primitive sensor readings from multiple input devices need to be correlated to form higher-level events. A wide range of applications has been proposed to utilize such sensors by extracting meaningful information from the raw data, commonly called gesture recognition. Examples include discovering when a phone is dropped, detecting whether the user is "throwing" data to another device,[1] and performing multi-touch gestures to quickly access information.[2]

In such multimodal systems with many possible interactions, the required computational power easily outgrows what today's processors provide in terms of sequential performance. This is problematic for real-time server-sided pattern recognition, for instance to process surveillance camera-input, as well as for embedded devices and mobile phones with various sensors such as an accelerometer, gyroscope, multi-touch, proximity-sensor, and a microphone. Fusing these primitive and higher-order events easily becomes excessive for a single processing unit, such that utilizing the steadily rising degree of available parallelism becomes a necessity to provide the required degree of real-time interactivity to the users.

☆This paper is an extension of: Renaux, T.; Hoste, L.; Marr, S. & De Meuter, W. (2012), Parallel Gesture Recognition with Soft Real-Time Guarantees, in 'Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions', pp. 35–46 .

Email addresses: `smarr@vub.ac.be` (Stefan Marr), `trenaux@vub.ac.be` (Thierry Renaux), `lhoste@vub.ac.be` (Lode Hoste), `wdmeuter@vub.ac.be` (Wolfgang De Meuter)

[1] *Hoccer, exchanging data using gestures.*, Art+Com Technologies, access date: July 8, 2013 `http://www.youtube.com/watch?v=eqv8Q6M1O6Y`

[2] *Gesture Search for Android*, Google Inc., access date: July 8, 2013 `http://www.google.com/mobile/gesture-search/`

Another common problem is that imperative programming languages are considered to be cumbersome, error-prone, and lacking flexibility [1, 2] when complex event correlations need to be implemented. General purpose imperative languages do not provide the necessary abstraction to help the programmer to express event patterns conveniently. Hammond and Davis [3], Scholliers et al. [4], and Hoste et al. [1] demonstrate that declarative definitions for sketch recognition, multi-touch gestures, or multimodal correlation provide the necessary language constructs and improve over imperative approaches by providing better software engineering abstractions.

Unfortunately, automatized recognition with machine learning techniques has drawbacks as well. On the one hand, it requires large sets of training data to build a statistical model of a gesture. Gathering such data and annotating it for the training process can be prohibitively time intensive. On the other hand, machine learning techniques are typically black-box processes that do not give developers the necessary information to debug and change the way gestures are recognized directly. Consequently, declarative approaches are a better choice today.

Declarative approaches are based on inference engines that process incoming sensor events using the declarative rules that describe the gestures. The Rete algorithm [5] is one possible foundation for such an inference engine. It is a forward-chaining, state-saving algorithm that is used to build rule-based expert systems. Declarative gesture approaches benefit from it because the state-saving approach optimizes the execution of rules so that only the program part relevant to an incoming event is executed, which minimizes the necessary computation that has to be performed whenever a new event takes place. As such, it reduces the computational overhead of continuous pattern matching enabling the use of a wide range of input sources and complex patterns without causing unacceptable performance overheads.

We present here a variation of the Rete algorithm called PARTE, built on a graph represented by a set of actors, providing both scalability and responsiveness. The contributions of our work are:

PARTE: design and implementation techniques tailored towards parallel recognition of user interaction patterns with soft real-time[3] guarantees.

Validation of PARTE's real-time guarantees by analyzing the execution properties of the implemented algorithm, which ensures freedom of unbounded loops and uses only non-blocking concurrent interactions. Furthermore, an empirical evaluation assesses the soft real-time properties of PARTE and compares it with a non-real-time variation.

Validation of PARTE's practicality by demonstrating the scalability of the parallel implementation on up to 64 cores and demonstrating that the sequential overhead compared to CLIPS,[4] a highly optimized sequential Rete implementation, can be overcome in the parallel case.

The remainder of this paper is structured as follows: first, in section 2 we provide a detailed discussion of the context of multimodal input systems, their requirements and constraints, and the assumptions we can make. Section 3 describes the design principles, implementation techniques, and the parallel Rete algorithm of PARTE in detail. Section 4 evaluates PARTE in the context of gesture recognition and characterizes its execution semantics both with respect to non-blocking behavior and with respect to unbounded loops. PARTE's performance is evaluated in section 5. Finally, we contrast our approach with the related work in section 6 and summarize our conclusions and future work in section 7. The used benchmarks are characterized in Appendix A as a foundation for the evaluation.

2. Context and Requirements

The domain of gesture recognition comes with a set of properties that is different from many domains in which inference engines are commonly used. Since we rely on these particularities of the problem domain

[3] In *soft* real-time systems, the usefulness of results degrades past their deadline, whereas in *hard* real-time systems the usefulness drops to zero on a missed deadline. Hence, delays in a soft real-time system undermine the system's quality of service, while delays in hard real-time systems undermine the system's correctness.

[4] *CLIPS: A Tool for Building Expert Systems*, Gary Riley, January 25, 2013 `http://clipsrules.sourceforge.net/`

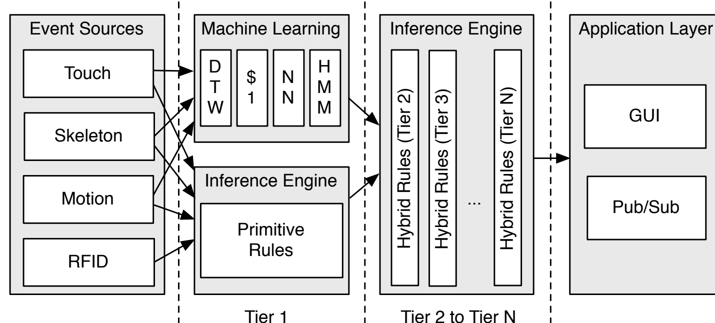


Figure 1: Contextual Framework

in the design of PARTE, we sketch the domain briefly and distill a list of requirements for inference engines in this domain.

2.1. Inference Engines for Gesture Recognition

Multimodal systems such as Mudra [1] embed inference engines that only tap the computational power of a single processing unit. However, the rise in sequential processing power offered by single processing units is stagnating, because efforts to increase clock-speed, instruction-pipeline depth, memory-bus width, and cache size offer diminishing returns. This severely limits the possible number of patterns (i. e., gestures), their complexity, and the rate of events the system can handle. The only way to recognize more complex user interaction patterns without undermining the user experience by increased delays, is to *embrace parallel processing power*.

To provide a high-quality user experience, an inference engine used for gesture recognition has to correlate events in a timely manner: when a user for instance interacts with a system through a multi-touch interface, changes should be reflected immediately, i. e., with *low latency*, and with a predictable delay, i. e., *predictable latency* to give users a natural experience and predictable feedback to their actions. The same is true for a broader multimodal interaction: when a user gives a series of voice and gesture commands, the right action should be performed without unexpected delays that confuse the user about whether the command has been accepted or not. This ensures that the system always feels interactive and responsive. Akscyn et al. [6] show that long delays in interactive systems can distract users, and even cause them to stop using the system altogether. Consider for instance a user of a multi-touch gesture recognition system, who taps at a certain location. If the user interface does not reflect this change within the time frame users have grown to expect, they will assume the command was not received, and may tap again. When the system then finishes processing the overdue gestures, the action will be executed twice. Users will rightfully blame the gesture recognition system for this mistake. To prevent such errors, the detection of complex user interaction patterns should happen within a time frame that can be predicted reliably up front. With standard programming techniques this is error-prone and requires substantial programming effort [2].

A resulting problem is that the requirements for low latency and predictable latency typically conflict: to offer the best performance, i. e., low latency on current hardware, the rule engine needs to use the available parallelism. However, to ensure predictable latencies the rule engine needs to provide soft real-time guarantees. Existing rule engines do not combine both requirements. They either are single-threaded in nature, or do not guarantee predictable worst-case execution times.

To give an example for a typical architecture of multimodal applications, figure 1 visualizes the data-flow of the approach presented by Hoste et al. [1]. Event sources such as multi-touch displays, skeletal tracking [7], accelerometer readings or the history of RFID tags contain potential valuable patterns that need to be processed. This unified architecture allows for low-level events to be processed using machine

learning-based approaches, as well as declarative definitions. Fusion and refinement of resulting higher-level events can be handled by consecutive declarative rules.

Given this approach to refining lower-level events gradually, we propose a *tiered architecture* for event processing. In this architecture, rules of *tier* N can consume only events that were generated by lower-level tiers (1 to $N-1$). It enables developers to easily modularize and compose their rules. For instance, a *hand-waving* gesture can be composed of two lower-level gestures *flick right* and *flick left*, which themselves were extracted from the low-level skeletal data, e. g., provided by a Kinect[5] system. A declarative approach enables this kind of efficient composition and helps developers to improve gesture recognition code. Enforcing tiering implies however that a rule of *tier* N cannot insert lower-level events as feedback for pattern classification on lower levels. While certain multimodal use-cases could benefit from using high-level event information to improve the accuracy of lower-level event detection [1], it is necessary to avoid feedback loops in order to achieve computational predictability.

Finally, the application layer in figure 1 uses a publish-subscribe model to register for high-level events processed by the inference engine. Depending on the application, it is useful to support different subscription modes. Topic-based subscriptions are used to filter by the type of the event and are the most common ones. However, for instance GUI components use content-based subscriptions to only react to events that happen at a specific spatial location. To enable such application-specific usage, the system needs to provide the necessary extensibility.

2.2. Requirements and Assumptions

Based on the discussion of the previous section, we derive the following set of high-level requirements for a parallel gesture recognition engine:

Soft Real-Time Guarantees The detection of user-interaction patterns has to complete within a time periode predictable by the user. This implies that rules have to be free of feedback cycles, i. e., a tiered architecture has to be used. Furthermore, event sources have to limit their event rate with an upper bound to limit the maximum load on the system.

Efficiency On top of providing predictable latency, the rule processing has to achieve sufficient efficiency to satisfy constraints on the response time, i. e., latency required for interaction with humans. Miller [8] identified three threshold levels in human attention, based on the order of magnitude of seconds that one has to wait. The lower two, response times in the order of tenths of seconds and response times in the order of seconds are perceived as respectively instantaneous and fluent interactions. For a system detecting user-interaction patterns, the interaction should at the very least be fluent, and preferably instantaneous, i. e., in the sub-second range.

Scalability on Multicore Hardware The performance of the system needs to improve with an increase in the number of available processing cores, relying on a shared-memory architecture. We focus on multicore hardware to be able to use the system in mobile or embedded solutions.

Optimized for Continuous Event Streams The production system has to be tailored for complex event processing on event streams with a bounded event rate. The event streams are assumed to be infinite and processing has to be online (in contrast to offline batch processing systems).

Extensibility and Embeddability The system needs to support user-defined functions to process and correlate events. Domain-specific tests are required to facilitate, for instance, testing of spatial properties of coordinates. For embedding into existing systems, it is necessary to produce the expected result format by invoking callbacks or sending messages to the consuming tier.

[5] *Xbox Kinect: Motion sensing input device*, Microsoft Corp., access date: July 8, 2013 `http://www.xbox.com/kinect/`

Based on these high-level requirements and the properties of the gesture recognition application domain, we derive a number of assumptions. These assumptions guide the implementation and enable us to achieve the desired soft real-time properties.

Since we target commodity multicore hardware, we assume shared memory between cores and a memory hierarchy with cache coherence. Distribution and the complexity that comes with it is therefore of no concern for the design of PARTE.

Another general assumption is that events are permanent. Thus, we distinguish events from facts in traditional production systems, which are true only until their validity is revoked. For the intended use case, fact retraction is not necessary. Therefore, events do not need to be retractable from the system as part of the action of a rule. Instead, we assume that the application level can always subsume events if necessary. This assumption enables us to avoid the need for *conflict resolution*: conflict resolution is commonly used in rule-based systems to order the execution of rule activations, enable retraction of facts, and subsumption of rule activations. For the intended use case, however, it is desirable that all matching rules will always be triggered and subsumption is deferred to a higher-level tier. Hence, the ordering does not determine the result, and conflict resolution serves no purpose.

The semantically indefinite validity of events entails further that the data structures representing events are not removed from working memory by the rules themselves. Therefore, they must be removed automatically by the system to prevent the working memory from growing unboundedly. For this, a sliding time window of events which are relevant to the current reasoning process can be used. We require events to be correlated by timestamp, so that the temporal dependencies between events can be used to statically compute their maximum useful lifespan: at any point in time, only those events can be part of a new pattern, for which there exist rules correlating them with other events that occurred within the lifespan of the first event.

While we require events to have a timestamp, the ordering of events is generally an application-specific issue and needs to be handled explicitly as part of the rules. Thus, automatic ordering is not done either in our *tiered architecture* (cf. subsection 2.1), because a higher-level event might require the timestamp of the first low-level event in a sequence, the last one, or the time span in which all the related lower-level events occurred. The choice of this timestamp or time span depends on the semantics of the declarative rule, so this choice cannot be automated. Such information therefore needs to be constructed and provided to the next tier explicitly, if temporal order between higher-level events needs to be known.

Classic rule-based engines are employed in business environments in long-running systems that need to be adaptable and allow changes to the rule set at runtime to avoid downtime. However, this leads to additional complexity and is not required for the given scenario. Thus, we assume that in most monitoring, games, and user interface applications only static sets of rules are used and that it is sufficient to determine the set of rules at startup time.

Summarized, the important assumptions are:

- Activations of different rules do not require ordering.
- Temporal dependencies are solved in an application-specific way by rules.
- Events never need to be retracted from the system. Event subsumption is done on a higher tier. Preventing memory leaks is handled by making events expire when they are no longer useful for the reasoning process.
- The set of rules is known and fixed at startup time.

3. PARTE

PARTE is a production rule system using a variant of the Rete algorithm to detect user-interaction patterns. To that end, it transforms a set of declarative *if-then* rules into a directed acyclic dependency graph, and uses this graph to match facts. For user interactions, this means that incoming events are processed by percolating through the graph constructed from interaction patterns, with which they are

matched to recognize high-level events. PARTE is designed to be scalable on parallel systems, as well as to satisfy the requirements and assumptions described in subsection 2.2.

This section first describes the architecture of the solution, how it is used for gesture recognition, and how it interacts with the main components in such an environment. Then, the parallel execution model of PARTE is described, detailing the solution strategy and implementation decisions that are essential to satisfy the posed requirements. Finally, a non-real-time version of PARTE is discussed, which is used for the evaluation.

3.1. *Architecture and Embedding into Gesture Recognition Context*

As outlined in subsection 2.1, inference engines such as PARTE are mediating between the raw input devices and high-level consumers such as application logic. For engineering reasons, such systems use tiered architectures to gradually enrich the semantics of the events. PARTE can be applied at multiple tiers in such an architecture. In such a scenario, PARTE would process incoming lower-level events from a set of event sources, based on a given set of rules which describe relevant patterns that need to be recognized in these event streams. Thus, PARTE is part of the middleware for building such applications.

Figure 2 depicts the architecture of PARTE with potential input sources on the left, and potential consumers at the right hand side. Rules, the templates describing the facts' structure, and the facts themselves are to be encoded as S-expressions and get converted to the internal representation by a set of parsers. A pool of threads is maintained by the engine, as well as a task queue on which the actors are scheduled. Finally, reentrant evaluator functions are provided to evaluate the test-expressions specified by the rules.

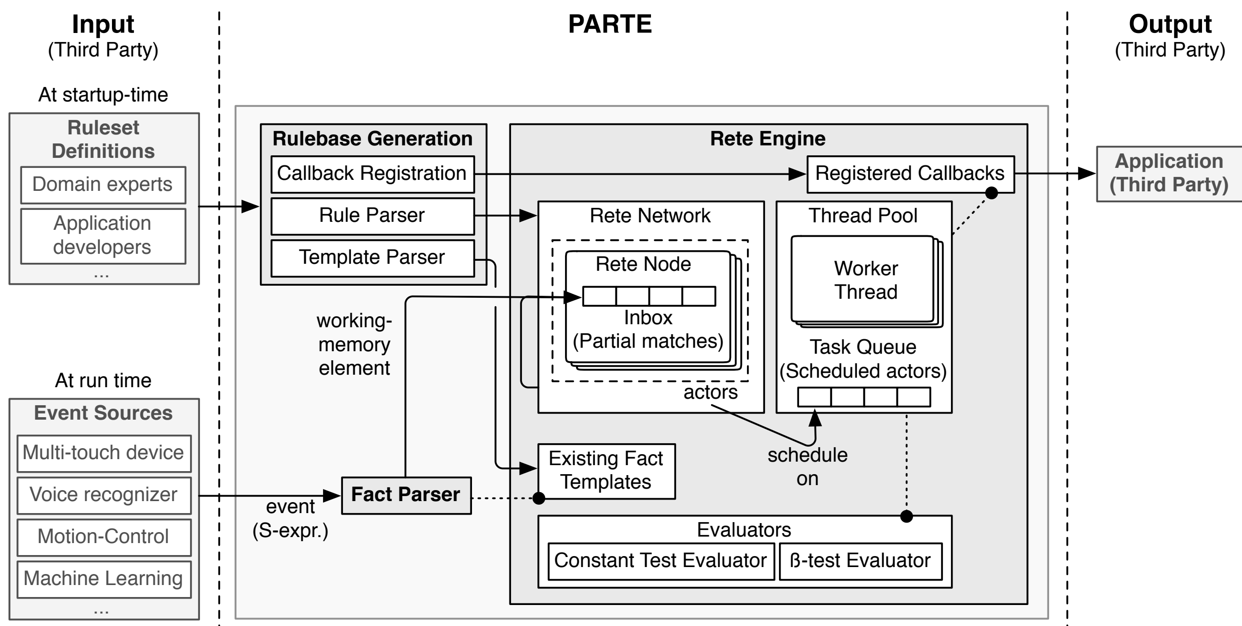


Figure 2: The architecture of PARTE

The Rete network itself is constructed from the set of rules inserted into PARTE at startup time. The rule parser converts the rules to an abstract syntax tree (AST). From the AST, it then builds the directed acyclic graph as prescribed by the Rete algorithm [5]. Node-reuse is not yet implemented in the parser, but the Rete nodes do support being shared by multiple subtrees where the Rete algorithm would allow it. After computing the Rete graph, PARTE computes the lexical addresses of variables by determining the index into the data structure which holds the values of tokens at runtime. As such, once the system is running, lookup of variables can be replaced with a constant-time indexed memory access. Finally, PARTE creates a set of actors and links them up to each other to constitute the Rete network.

To be used for gesture recognition, PARTE can handle three types of actions: *1*) assertions of new facts, which consists of constructing a fact based on some description and propagating it to the inboxes of the relevant entry nodes of the Rete network; *2*) calling foreign functions, passing in the values bound to variables[6]; and *3*) terminating the system. PARTE does not support retractions of facts, since in our event processing context, facts get removed from the fact base automatically when they expire. Since facts are only removed when removing them has no semantical impact, as described in subsection 2.2, conflict-resolution or other means of ordering actions are not required. Instead, actions are executed immediately as soon as a rule is triggered.

Note that this design is different from traditional Rete engines which use an additional data structure: the agenda. This agenda is commonly used in sequential Rete engines to represent the FIFO queue of actions that have to be performed by the inference engine as a reaction to triggering a rule. This is necessary to resolve conflicting rule activations of multiple matches, however, since this is not necessary in our context, PARTE omits the traditional agenda to avoid an otherwise globally shared data structure and that would likely be a bottleneck in a parallel system.

```
(defrule detectZShape
   ?hA <- (horizontal-drag)
   ?hB <- (horizontal-drag)
   ?diagonal <- (down-left-drag)

   (test (endMeetsStart ?hA ?diagonal))
   (test (endMeetsStart ?diagonal ?hB))
   (test (chronologically ?hA ?diagonal ?hB))
=>
  (reportZShapeCenteredOn
      (avg ?hA.startX ?hA.endX
           ?hB.startX ?hB.endX)
      (avg ?hA.y ?hB.y)))
```

Listing 1: A possible rule for gesture recognition

An Example Gesture. The S-expression in Listing 1 gives an example of a high-level motion gesture rule that can be processed by PARTE. The expression defines the rule `detectZShape`, which describes how two horizontal drags and a down-left drag can combine into a Z-shape. Line by line, the rule binds two events of type `horizontal-drag` to the variables `?hA` and `?hB`, and binds an event of type `down-left-drag` to the variable `?diagonal`. Then, it uses the user-defined functions `endMeetsStart` and `chronologically` to verify that the shapes follow each other both spatially and temporally. As a consequent to the recognition of the Z-shape, the rule specifies that the callback `reportZShapeCenteredOn` should be called, passing the average `x` and `y` slots of the shape's points. Figure 3 shows a Rete graph and the flow of the facts along the edges when the facts at the lower-left corner are asserted into the Rete network. Lists delimited by square brackets denote tokens, the units of communication between the nodes in the network.

3.2. Parallel Execution Model

For the description of the execution model, we use the metaphor of the actor model [9] to map each node of the Rete network to its own actor, executing independently. While such an approach does not enable us to utilize potential data parallelism for the matching inside a node, it enables a high degree of parallelism

[6]Since foreign functions are plain C functions, they *can* technically perform whatever I/O or other time-consuming and/or blocking operation they want, but *are presumed* not to do so. If a rule should require something which is not normally considered a good match to event processing, such as reading a file, the user-defined function should dispatch the job to a worker thread provided by the application hosting the PARTE engine, in a non-blocking way. That thread can then read the file and assert information back into the systems using facts.

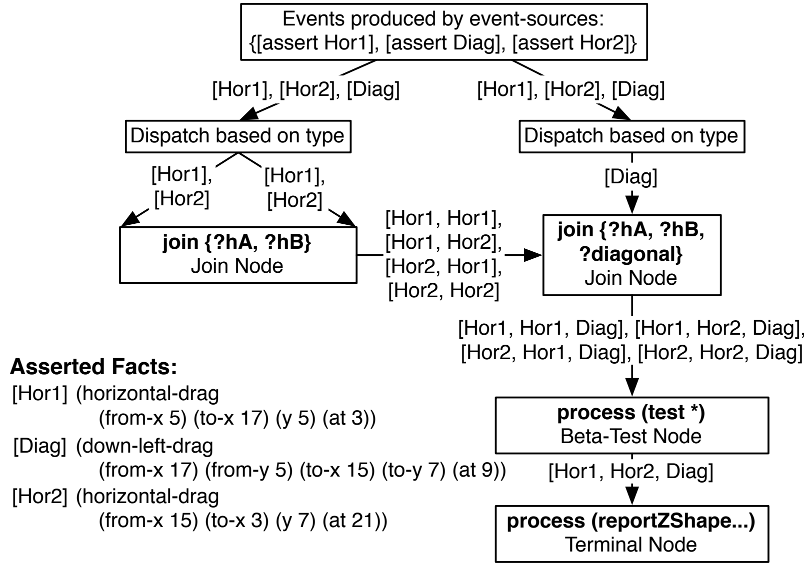


Figure 3: Flow of data through the Rete network specified by the rule in Listing 1

throughout the network. Even in situations where only parts of an actor network are used frequently, this approach enables pipeline parallelism, enabling scaling on multicore processors.

Furthermore, the directed acyclic graph (DAG) structure of a Rete network, and its structural properties in terms of edges between nodes provide an ideal foundation to apply non-blocking data structures to gain predictable upper bounds for execution time. We utilize these characteristics to provide the desired real-time properties.

Execution Model. As indicated above, the individual actor nodes of the Rete network are the parallel units of computation. The DAG of the Rete network thereby forms a task dependency graph for the match phase of the fact processing. This means that every actor node needs to wait only for information from their predecessors in the Rete graph, and send data only to their successors in the Rete graph. This entails that the same spatial and temporal efficiency that the Rete algorithm offers for matching facts to rules, also ensures low contention for shared resources: every node's communication channel is contended for by at most two predecessors and its own thread of control.

The Rete algorithm's approach of passing tokens between nodes corresponds directly to message-passing between actors. Especially, since the step of processing an incoming token can be seen as an atomic operation by the rest of the system, the processing does not require the notion of shared memory. In the implementation of PARTE, the nodes of the Rete network have an inbox, which is realized with a non-blocking queue, in which predecessors put the incoming tokens. A node dequeues tokens from its inbox one-by-one for processing. Based on the non-blocking nature of the inboxes, we avoid the potential for deadlocks and livelocks. In order to avoid frequent rescheduling of worker threads and actors, a node/actor will consume all available messages from its inbox before yielding the worker thread back to the scheduler.

To prevent starvation, every actor, i. e., every Rete node, is scheduled on a thread pool using a round-robin scheduler. Figure 4 shows a possible snapshot of a running PARTE system which contains only the rule specified in Listing 1. Two threads are allocated in the thread pool. A third one is used to receive the events coming from external event sources. Hence, in this case only two of the four actors can be active at the same time. The other two actors remain queued on the task queue. Note that the size of the system PARTE is running on and the number of worker threads it is using, needs to be adapted to the used rule set, i. e., to the structure of the network in order to ensure that the system is never overloaded and Rete nodes, i. e.actors are not starving.

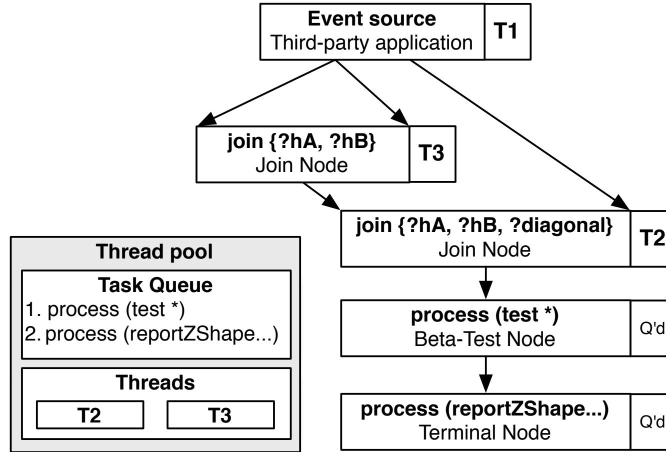


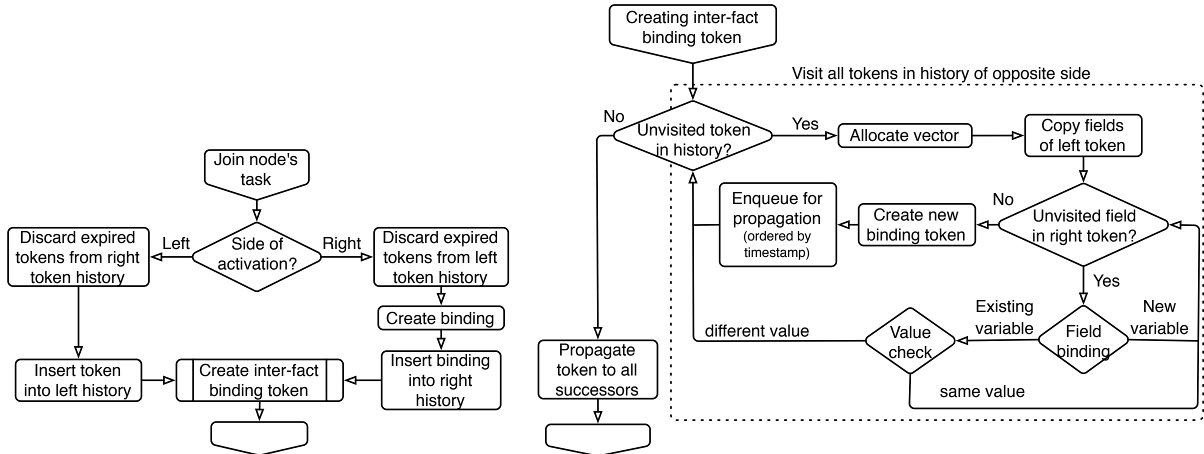
Figure 4: A potential runtime snapshot of a PARTE system detecting the pattern specified in Listing 1

Non-Blocking Data Structures. In order to achieve soft real-time properties, communication between actors representing Rete nodes needs to be non-blocking. Specifically, the operations to enqueue a message into the actor's inbox as well as dequeueing it needs to complete in a bounded number of steps. In our design, we achieve this by using non-blocking FIFO queues in combination with the guarantee that each Rete node has at most two predecessors adding messages to the inbox, and one message consumer, i. e., the node itself. Our implementation is based on the FIFO queue design by Harris [10]. The list offers lock-free inserts at the front and deletes at the back, and contention is localized to only the element that is inserted or removed, meaning that in lists with two elements or more, enqueuing and dequeuing can happen simultaneously without interfering with each other. Each node of the list consists of a pointer to the data and a next-pointer. The queue always contains at least one such node, with `NULL`-ed out data: the 'dummy' node.

To enqueue an element in the non-blocking queue, a new list node is created, and its data pointer filled in. The algorithm then enters a loop in which it tries to enqueue the newly created node. To this effect, it grabs the current tail-pointer of the queue, looks at its next-pointer, and if it is not `NULL`, helps the other thread which must have been responsible for adding a new node past the tail, by attempting to atomically compare-and-swap the next-pointer to the queue's tail slot. If the tail's next-pointer is `NULL`, then the algorithm attempts to compare-and-swap its own newly created node to the tail's next-pointer. If that succeeds, the algorithm breaks out of the loop; otherwise it keeps looping. After breaking out of the loop, the algorithm still has to compare-and-swap the newly created node to queue's tail-pointer, but if that fails (i. e., when another thread has overwritten the tail-pointer since), no correcting action has to be performed: the other thread will have set the correct tail-pointer. This loop to enqueue an element is bounded, and both local and global progress for worker threads is guaranteed, because the Rete network is constructed so that the number of producers and consumers is limited avoiding perpetual contention on the queues. Thus, PARTE's design of using lock-free inserts with a limited number of producers, contributes to the soft real-time guarantees.

Dequeuing works similarly, however, since all lists have only a single consumer, a simple compare-and-swap on the head-pointer to the head-pointer's next-pointer is sufficient. The case of the empty list is implicitly handled with the dummy node.

Memory Management. Since PARTE is implemented in C++, memory management becomes an issue that needs to be handled carefully. In our implementation, all tokens in the Rete network are handled by value and the Rete node receiving a token is responsible for freeing it when it expires. Expiration of events can be determined from data local to the actor in which the tokens are stored. Therefore the expiration does not require a global synchronization effort as is the case in systems where events are only removed after



(a) Steps to process incoming tokens of a join node (b) Steps to create infra-fact bindings, i. e., applying unification

Figure 5: Flow charts depicting parts of the matching algorithm

a request for retraction has percolated through the Rete network. PARTE makes use of the timestamps carried by tokens for two purposes: 1) to determine how long those tokens still have to be preserved, and 2) to implement a logical clock with which nodes can know what the oldest point in time is from which their predecessors still may have unpropagated events. By maintaining the invariant that tokens are always propagated in-order, the node actors can know whether new tokens can still correlate with stored tokens, and discard tokens for which this is not the case.

More complex is the situation of managing the list elements for the lock-free list implementation. Since they are inherently subject to race conditions, we use a solution proposed by Michael [11] that is similar to reference counting. All operations on list elements must maintain a set of *hazard pointers*. The hazard pointers are kept in a contiguous piece of memory, and their number depends on the number of worker threads as well as the use of the data structure. Each thread is associated with a subset of these pointers for its own use. Furthermore, in PARTE, we establish a fixed mapping between hazard pointers and the functions using them. This is possible since only operations on message queues require this form of safe memory management.

This means that before using list elements, a thread in PARTE will always stores pointers to these list elements into the thread's section of the hazard pointer array. When an element is deleted from a list, a memory reclamation operation is triggered, which iterates over all hazard pointers in the system to determine all list elements that could still be in use. With this conservative overestimation, it is safe to free all list elements that have not been referenced by a hazard pointer or are part of a list, because no thread is currently using them.

The amount of memory that is no longer in use, but not yet reclaimed, is bounded by a small constant per thread, and a reasonably small constant in total. The per-thread constant is the fixed number of hazard pointers. The overall constant is based on the fact that the overall number of threads is constant as well.

Sketch of the Rete Node Implementation. The implementation of the Rete nodes follows closely the original Rete design [5]. Therefore, we discuss here the algorithm used for join nodes to illustrate our solution strategy. Figure 5 gives a sketch of the algorithm. Figure 5a depicts the high-level process of handling the incoming tokens depending on whether they correspond to the left or right hand side of the join node. The right hand side of a join node receives only simple *alpha* tokens, while the left hand side receives *beta* tokens, which already contain variable bindings and potentially wrap multiple facts.

The first step a join node undertakes when it receives a token from either the left or the right hand side, is to discard expired tokens from the history of the other side. As explained earlier, this is based on the timestamp of the incoming token and on knowledge about their maximum lifetime, which can be determined

from the ruleset. A precondition is here that the number of items in the history of any join node is bounded by the rate of events. From this precondition follows that the number of execution steps is bounded for this cleanup step.

In case the join node is activated from the right hand side, it will subsequently create a binding (a beta token) satisfying the fact pattern using the values from the alpha token with which it was activated. In case of an activation from the left hand side, the token already contained such a binding. Regardless of the activation side, the next step consists of inserting the binding into the corresponding history.

This process is completed by the steps to create intra-fact bindings, i. e., unify facts as depicted in figure 5b. This unification process has two relevant loops. The outer loop iterates over all elements in the history, and thus is bounded. The second loop iterates over the fields of the facts and therefore is bounded as well.

The other node types are treated similarly to ensure the required soft real-time guarantees.

3.3. Non-real-time PARTE

Non-blocking data structures like the queues we use as inboxes for the actors and as task queue, come with important real-time properties. However, their absolute performance may be lower than that of comparable data structures which are not required to offer real-time guarantees. Furthermore, the need for real-time behavior forced us to utilize a scheduler which operates with very little information about the tasks it is scheduling. Most importantly, our scheduler is oblivious to whether the actor it schedules for execution actually has work in its queue.

Since in some cases raw performance is more important than predictable behavior, we created a version of PARTE which utilized a more efficient scheduling algorithm, albeit one which cannot offer real-time guarantees. This allows us to compare the performance of both, and quantify the loss in performance we incur from our real-time requirement.

The non-real-time version of PARTE is built on top of the same codebase, but differs in the following two aspects: Firstly, instead of non-blocking queues, blocking queues are used. The blocking queues are implemented using queues from the C++ standard library, protected by a lock and a condition variable. To enqueue an element into the blocking queues, a thread has to acquire the lock, push the element into the internal queue data structure, signal the condition variable that the queue is non-empty, and release the lock. To dequeue an element, a thread has to acquire the lock, check whether the queue is empty, wait on the condition variable to indicate that the queue is not empty anymore, pop the element from the internal queue data structure, and release the lock.

Secondly, the functions being executed by the worker threads have to be modified to work with the blocking queues. Instead of processing all items in the inbox, and picking up the next actor's task when the current's inbox is empty, the worker threads in the non-real-time version continuously try to dequeue elements to process, and block when the inbox is empty. As such, every actor is mapped one-to-one onto a thread, and the scheduling of the threads is left to the operating system's scheduler. We do not propose this one-to-one mapping as a general approach to actor-based systems. However, the design forms a useful basis for the comparison between real-time and non-real-time versions of our algorithm in section 5.

4. Conceptual Evaluation

In subsection 2.2, we outlined the requirements for a parallel inference engine used in the context of parallel gesture recognition. This section discusses how PARTE satisfies these requirements. First, we discuss the general requirements, how PARTE is optimized for complex event processing (CEP), and how it is embeddable into existing systems. The second part of the evaluation evaluates the soft real-time properties of PARTE by arguing that the general algorithmic properties and the boundedness of the evaluation process provide the desired guarantees.

4.1. Extensibility and Embeddability

PARTE is designed to be a middleware mediating between the low level of event sources and the application level. To that effect it is a self-contained system, managing its own memory and interacting with other systems via a simple API and callback methods. Applications using PARTE must provide a ruleset and register user functions and callbacks. Event sources simply need to inform PARTE of new events. The inference engine is continuously running and processes the incoming events as soon as they arrive, maximizing throughput. Through this low coupling between PARTE and the remainder of the system, PARTE is a reusable, easily embeddable software component. The notion of custom user-defined predicates and actions enables PARTE to process arbitrary events and produce output in whatever format the application requires.

4.2. Continuous Event Streams

Since the input devices are assumed to continuously produce events, PARTE was designed to handle expiration of facts automatically to avoid unboundedly growing fact bases. The sliding time window mechanism explained in section 3.2 causes the expiration of events which are no longer relevant. This removes not only the burden of manual memory management from the application-level; it also reduces synchronization overhead as *retract* messages need not be sent and processed separately. This results in a speedup compared to a version which would not use automatic expiration, as well as in improving the scalability.

4.3. Soft Real-Time

For the assessment of the real-time properties of PARTE, we rely on the algorithmic properties only. Thus, we disregard architectural issues such as microprocessor architectures [12] and operating system aspects [13]. Instead, we give an informal argument to demonstrate that the pattern matching is done in a bounded number of steps, which all complete in a bounded amount of time.

We therefore demonstrate the soft real-time properties of our system by showing that a predictable bound exists on *a*) the time every *turn* of an actor requires; on *b*) the time required for scheduling actors; on *c*) the time required for passing messages between actors; on *d*) the amount of actors; and on *e*) the amount of turns per actor that are required to detect a pattern.

This analysis relies on the requirements stated in subsection 2.2. Thus, we assume that the rule set of an application relies on tiering and is free of feedback cycles. Furthermore, all event sources of an application are required to work with an upper bound on the rate with which events are produced. Without such a bound, the operations performed by the Rete network would be unbounded as well. Note that the specification of an upper bound on the event rate together with a concrete set of rules would allow for an estimate of the computational power required from the system to avoid overloads. However, such performance modeling is outside of the scope of this paper.

Upper bound on actor operations. For this first requirement, more information about the inner workings of the actors is required. We introduced three types of actors in figure 3: join nodes, test nodes, and terminal nodes. In the case of terminal nodes, the requirement is trivially met. They perform a constant amount of work, performing a number of actions which cannot change at run-time.

For test nodes, the situation is barely different: the arithmetic expressions and (in-)equality tests they evaluate are fixed at compile-time, so an upper bound on their runtime can be computed.

For the join nodes, the argumentation is a little more involved, as discussed in section 3.2. The only variable factors in a join node's runtime, however, are the number of variables to unify and the number of fields that have to be bound to those variables. Both are constant given a concrete ruleset, and are therefore known at the time the Rete graph is being compiled. Furthermore, since we require an upper bound on the event rate, the number of tokens in a join node's history is bounded as well. Thus, for every ruleset there is an upper bound on the time a join node needs to process an incoming token (cf. figure 5a and 5b).

The scheduling and message passing requirements are closely related. Both the task queue and the actors' inboxes are implemented based on non-blocking FIFO queues [10]. Since the structure of the Rete algorithm limits contention on the actors' inboxes to two producers, which use an lock-free insert operation, and Rete

nodes cannot generate an arbitrary amount of tokens before having to wait for new incoming tokens, an upper bound on the time required to enqueue and dequeue exists.

The requirement for an upper bound on the number of actors is trivially met, as all actors are statically allocated at startup time, their number being constant and depending on the concrete rule set.

The last requirement is fulfilled thanks to tiering. By definition, the Rete DAG is acyclic, yet cyclic behavior can be achieved when the consequence of a rule includes the assertion of facts which may trigger that same rule again. By enforcing tiering, we prevent assertions from creating such a feedback loop. Hence, not only the width but also the depth of the loop-unrolled graph is bounded and known for a given ruleset. Since we require a known upper bound on the rate at which events can be asserted into the system, and communication happens via FIFO queues, the maximum number of turns required before all events currently in the system are processed can be computed.

To conclude, all operations of the Rete network have an upper bound for a given ruleset. Consequently, an overall upper bound on the amount of time PARTE requires to detect gestures can be computed. From this follows that PARTE offers bounded execution time and provides soft real-time guarantees.

4.4. Discussion

Our choice of actors as the unit of parallelization stems from the strong similarity between the passing of tokens between nodes in the Rete algorithm on the one hand, and message passing between actors on the other hand. The actor model provides a nice metaphor for the construction of a directed graph of interlinked nodes which share data only by explicitly passing it to their successors.

The main conceptual limitation in the proposed PARTE design is the absence of negation as a language feature. Since test-expressions are separated from the nodes that join two branches, negation-as-failure is not supported. The parallel execution model would require us to implement negation with additional communication between nodes and does not fit into the current model. However, gesture recognition systems can work without a notion of negation, but its addition would be worthwhile for other use cases. Furthermore, negation would reduced the number of rule activations significantly, and could thus improve performance.

5. Performance Evaluation

To evaluate the performance of PARTE, we follow the methodology proposed by Georges et al. [14]. The benchmarks were executed on a Ubuntu 12.10 server, kernel version 3.8.0, with four AMD Opteron 6376 processors at 2.3 GHz. Dynamic frequency scaling has been disabled to reduce measurement errors. The machine has 64 GB of memory with NUMA (non-uniform memory access) properties. NUMA is common for such server systems, but poses a challenge for software developers, as it can inhibit scalability.

CLIPS as well as PARTE and its non-real-time variant were compiled with maximum optimizations (-O3) using the GCC 4.7.2 compiler shipped with Ubuntu 12.10.

We use a set of 15 microbenchmarks and a set of multi-touch gesture recognition rules as application-level benchmarks to evaluate the performance characteristics of PARTE. The microbenchmarks were executed 60 times, and the application-level benchmarks were run 10 times each.

Each benchmark consists of a set of rules and a set of pre-generated events to be fed into the system. The microbenchmarks measure the performance of variable binding, different fact sizes, unification, and β-tests. Furthermore, they assess the performance of rules with complex tests, and tests that use computationally intensive user functions. For motion detection such tests are typically trying to find the spatial relations of a group of points and movements or use computationally-intensive classification approaches. Appendix A gives a characterization of each benchmark.

We first look into the efficiency of our system by comparing the runtime performance of PARTE running on a single thread with the runtime performance of CLIPS. CLIPS is an open-source and highly tuned sequential implementation of the Rete algorithm and forms the basis of multiple other production systems, such as PRAIS [15] and Lana [16]. After evaluating the efficiency, we assess the soft real-time properties observed in practice and compare it between our two PARTE variants. Finally, we evaluate PARTE's scalability by discussing its performance on up to 64 cores.

Beanplots. Part of the evaluation uses beanplots [17] as a visualization of the obtained measurements. Beanplots depict the distribution of measurements by highlighting regions with a dense population of measurements with a *wider bean*, while the range of outliers is indicate with a long tail of the bean. We chose beanplots over more common boxplots because beanplots show more data, where boxplots would hide relevant details. An additional advantage is that beanplots allow us to depict the distribution of two measurements side-by-side with so-called *split beans*, which allows for a direct visual comparison.

Performance Analysis. In order to attribute the observed performance characteristics to specific design and implementation details of PARTE, we experimented with different variations of PARTE. Since today's profiling and performance analysis tools are still limited, we used an indirect technique to identify bottlenecks.

Concretely, we doubled the amount of operations performed by the different relevant mechanisms individually and compared the results to our baseline. We did this experiment for the message passing implementation, the lock-free queues, and the scheduling. While the operations have been performed twice, the semantics remained unchanged. For the message passing this included, for instance, allocating two messages instead of one. This indirect approach enabled us to identify the most relevant bottlenecks, since a doubling of the performed operations caused significant performance loss.

5.1. Sequential Performance

To assess the sequential efficiency of our implementation, we compare CLIPS to the soft real-time and non-real-time version of PARTE. For this evaluation, we consider the execution time of all systems on a single core. Our goal is to demonstrate that PARTE, in its current unoptimized state has acceptable sequential performance characteristics in direct comparison. Thus, the average performance should be in the same order of magnitude.

Figure 6 depicts the results in the form of a beanplot [17] giving an overall overview, and in the form of a bar chart, which gives the details for all microbenchmarks. The beanplot depicts the overall distribution of all measurements for all benchmarks for the corresponding Rete implementation. For each benchmark, the results have been normalized to the average of the CLIPS results. The distribution of the CLIPS results are depicted in gray, while the results for PARTE are shown in black. Since the results are normalized to CLIPS, its results are centered around the expected 1.0 ratio of the runtime. The depicted distribution further indicates the measured outliers.

The beans for PARTE and PARTE its non-real-time variant (non-RT) make the performance difference of benchmarks visual. While some are on par with CLIPS, others indicate that the PARTE implementation has performance overheads and is not as optimized as CLIPS. However, the geometric mean[7] over all measurements indicates that PARTE is only about 2.1x slower than CLIPS, and the non-real-time version of PARTE (non-RT) is about 2.2x slower.

The bar chart in figure 6 details the performance aspects. In case of the *Simple Tests* benchmarks, the cost of reifying each node as an actor with *by-value* message passing and PARTE's round-robin scheduler with real-time guarantees is significantly higher than the actual test operations performed. The main overhead is caused by the scheduler. Since the design is constrained by the requirement for real-time guarantees, it is a conceptual overhead and cannot be directly avoided. A consequence of this design is that all variants of *Simple Tests* experience a significant slowdown compared to CLIPS. However, such pathological behavior as in these microbenchmarks is uncommon in applications. Application behavior will be closer to the *Complex Tests* benchmarks, which show on par or even slightly better performance. Here, the scheduling overhead is insignificant and the efficiency of the expression evaluator is the main aspect measured. The *Heavy Tests* also exhibit the expected on par performance, because in their case scheduling overhead is also negligible. The *Joining Tree* benchmarks, however, show again the significant impact of scheduling in cases where the work per Rete node is minimal, and thus, they exhibit significant slowdowns. The *Search* benchmark[8]

[7] Since we compare ratios to obtain intuitive results, the geometric mean has to be used (cf. Fleming and Wallace [18]).

[8] This benchmark is essentially a search problem because the Rete engine needs to find the specific top node, i. e., entry node of the Rete network that leads to the match of a rule based on the given input. If the correct entry node is selected early based on a changed search or scheduling strategy, the Rete engine can achieve superlinear speedups.

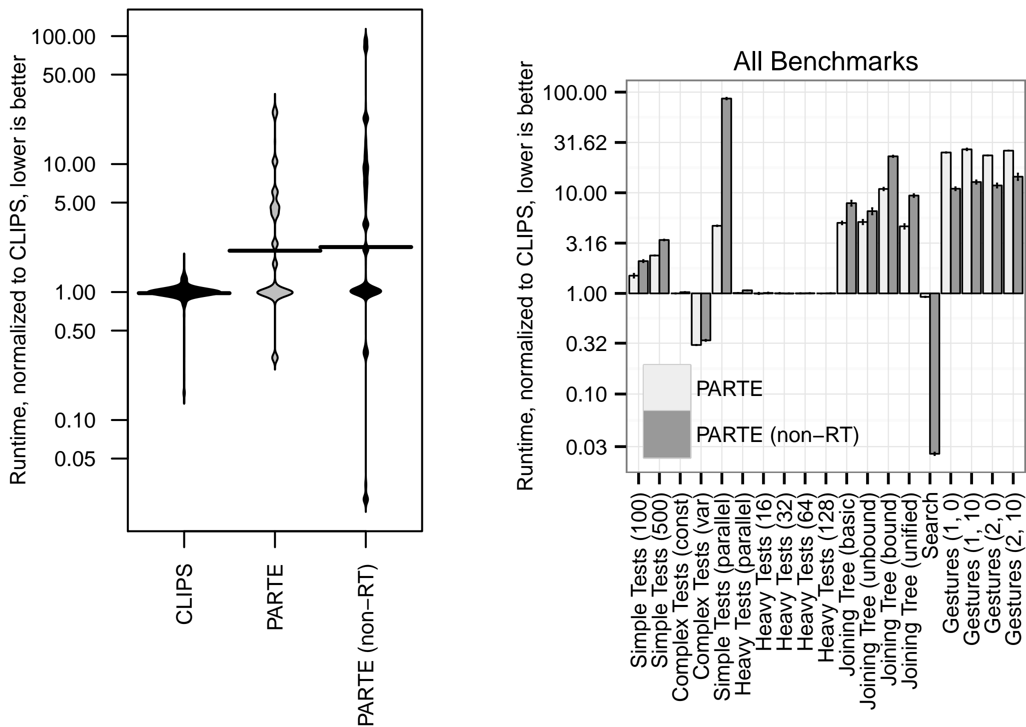


Figure 6: A beanplot comparing CLIPS to PARTE and PARTE (non-RT) executing on a single core. The overall slowdown of PARTE on all benchmarks is 2.1x, while PARTE (non-RT) is about 2.2x slower. Thus, we claim that PARTE exhibits in its unoptimized state an average performance that is of the same order of magnitude as that of CLIPS. The bar chart details the performance characteristics for each benchmark. It indicates that reifying each Rete node as an actor has a significant impact on PARTE's single core performance. For the gesture benchmarks this leads to an average slowdown of 25.3x for PARTE, while PARTE (non-RT) experiences only 12.4x slowdown, because it benefits from the changed scheduling logic. The *Complex Tests* benchmark indicates further that the expression evaluator is on par with CLIPS, which is an important requirement for complex Rete-based applications.

benefits from the changed order in which Rete nodes are processed. While CLIPS uses a strict depth-first approach, the PARTE scheduler does not enforce the depth-first order, since it is not necessary to avoid conflicts between rule activations. Thus, PARTE can match the final rule much earlier than in the sequential case, which demonstrates a benefit of PARTE's design for large rule sets with few active rules. A typical use case that benefits from this design would be to recognize a symbol gesture from a large set of symbols (cf. Appendix A.1.4).

The gesture recognition benchmarks show also a significant slowdown compared to CLIPS. PARTE is about 25.3x slower, while PARTE (non-RT) is about 12.4x slower. The main reason for the slowdown is the comparably simple tests performed in the multi-touch gestures. The benchmarks rely on comparably simple spatial operators, which have insignificant computational cost. Thus, the scheduling overhead has a significant impact on the sequential performance.

5.2. Soft Real-time Properties

In order to evaluate whether PARTE achieves the desired soft real-time properties, we compare its benchmark results with the results obtained for the non-real-time version. From a performance perspective, the design of PARTE has one conceptual bottleneck and that is the round-robin scheduling of Rete nodes using a non-blocking queue (cf. subsection 3.2). While this design gives predictable behavior and an upper bound on the execution time, it becomes a concurrency bottleneck (cf. subsection 3.3). By removing this bottleneck and relying solely on the operating system's scheduler in the non-real-time version of PARTE, we trade predictable execution characteristics for potential scalability.

From a user perspective it is important that the recognition rate of patterns is predictable, thus, a gesture should be recognized within a bounded amount of time. Outliers that take more time are undesirable. In terms of a soft real-time system, they increase the lateness, i. e., tardiness of the system.

To compare the predictability of performance characteristics of the two implementations, we compare the variations in the measured benchmark results. For this evaluation, we use the complete set of measurements utilizing 1 to 64 cores. Figure 7 visualizes the variation, i. e., the tardiness of the two PARTE implementations in terms of beanplots. We determine the variation by taking the mean for each benchmark configuration, i. e., for a specific benchmark, a specific number of cores, and one of the Rete engines. We normalize the results of this benchmark configuration with the mean and use the normalized values for the beanplot. To assess the tardiness of the systems, only the measurements that take more than the average runtime are relevant. Therefore, the beanplots only depict the distribution of the measurements larger than the mean.

The overview in the left half of figure 7 contains all results and indicates the tardiness. The beanplot on the right shows the results for each benchmark separately. The detailed view reveals that the benchmarks that are highly affected by scheduling compared to the amount of actual computation show more consistent results for the soft real-time version of PARTE. The number of measured results that are late on the non-real-time version is twice as high. We measured an average standard deviation of 4.7% for PARTE, whereas the average standard deviation for PARTE (non-RT) is 10.1%. The overall impact of parallel execution on the predictability is however still considerable. The average standard deviation measured for CLIPS is 1.5%.

To conclude, our design of a parallel Rete engine with soft real-time guarantees can provide significantly more consistent results and has an overall reduced tardiness compared to a parallel non-real-time Rete engine.

5.3. Scalability

In order to evaluate the scalability of PARTE, we discuss its performance for few-cores, i. e., up to 8 cores, and for large systems with up to 64 cores separately. The goal is to discuss the performance potential on common systems and up-coming mobile and embedded devices, separately form large-scale systems, which introduce for instance the complexity of non-uniform memory access (NUMA).

5.3.1. Scaling up to 8 Cores

Figure 8 and 9 indicate that PARTE is able to benefit linearly from additional cores for both the microbenchmarks and the application-level gesture recognition benchmarks. They show the results as speedup

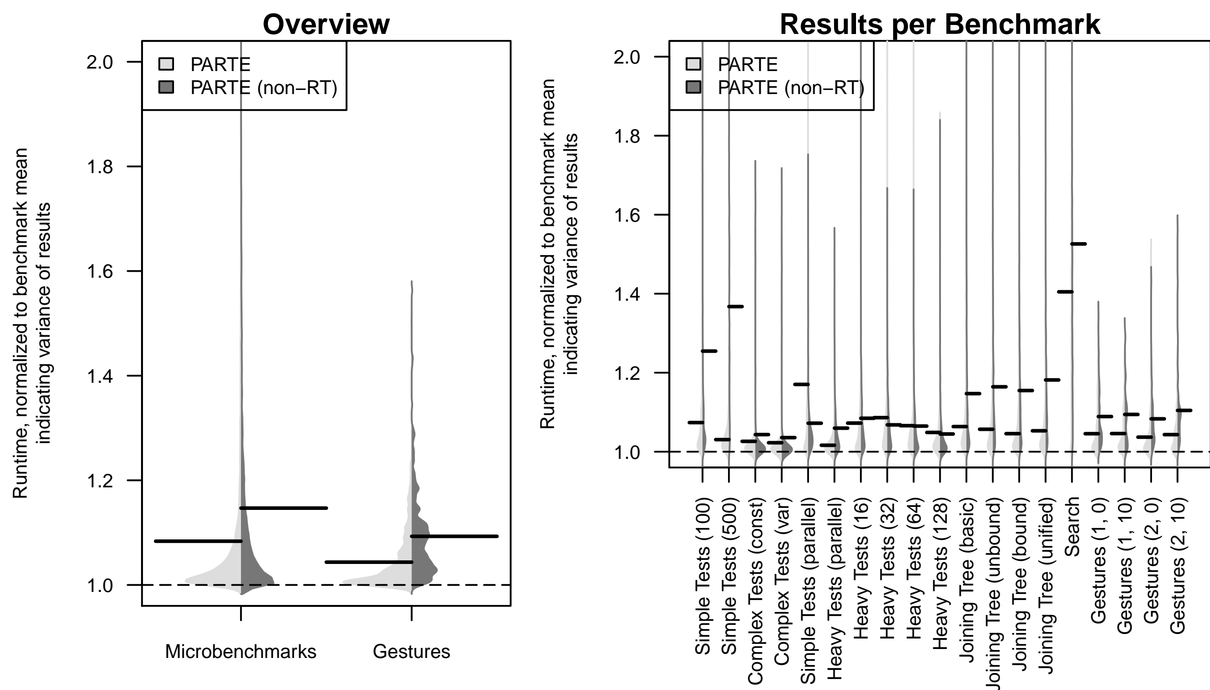


Figure 7: Beanplot comparing the measured variation of benchmark results for PARTE (left part of bean, light gray) and PARTE (non-RT) (right part of bean, dark gray) on 1 to 64 cores using split beans. The overview plot shows that the number of late results is significantly increased for the non-real-time version. The detailed plot shows the benefit of PARTE's soft real-time design for benchmarks with a large percentage of scheduling operations. These benchmarks exhibit a significantly reduced number of outliers, i. e., deadline misses. On the other hand, the non-real-time version exhibits a higher degree of outliers and provides therefore less predictable performance characteristics.

graphs. The dotted line indicates the ideal speedup—a speedup which increases linearly with the amount of processing cores. The remainder of this section details the observed performance and discusses it in the context of PARTE's design.

Microbenchmarks. Figure 8a and 8b depict the results for the *Simple Tests*. While these benchmarks are able to utilize the additional cores, the high overhead of the scheduling is apparent.

The *Complex Tests* in figure 8c and 8d show close to ideal scaling because the computational overhead of evaluating the expression makes the scheduling overhead negligible. The *Heavy Tests* in figure 8e do not scale as well, because the computational intensity is below that of the expression evaluation of the complex tests, and the results are still dominated by the scheduling overhead. Figure 8f to 8h show that this can be overcome by increasing the parallelizable workload.

The *Joining Tree* benchmarks in figure 8i to 8l show linear scaling. However the scaling factor depends significantly on the computational cost of the operation each node has to perform. With rising computational cost, the scaling factor comes closer to optimal.

The parallel version of the *Simple Test* in figure 8m shows better scaling than the pipelined version since the parallel execution avoids stalls on the predecessor caused by the communication delay and overhead. Similarly, the parallel version of the *Heavy Tests* in figure 8n shows perfect scaling by avoiding stalling on the predecessor in the pipeline.

For the *Search* benchmark in figure 8o, we see the expected superlinear speedup because the final node is scheduled much earlier than during sequential execution.

The non-real-time version of PARTE does not exhibit the same performance characteristics. The overhead of context thread switches and condition variables is too high in these microbenchmarks, and the scheduling data structure for the soft real-time PARTE is not yet a bottleneck. Furthermore, the indicated confidence intervals are overall larger than for the soft real-time version, showing that the performance is less predictable in the non-real-time version, which would lead to missed deadlines and undesirable delays in user-interface applications.

Gesture Recognition Benchmarks. The results of the *Gesture* benchmarks indicate a scalability similar to the microbenchmarks. However, since the problem is essentially a search problem of finding the rule to be activated from a large set of rules, we see a small superlinear speedup. Figure 9 depicts the results for the four variations of the benchmark. To simulate multi-touch input devices, the event stream contains additional noise, i. e., events unrelated to the actual gestures. However, the additional noise events do not inhibit PARTE's scalability.

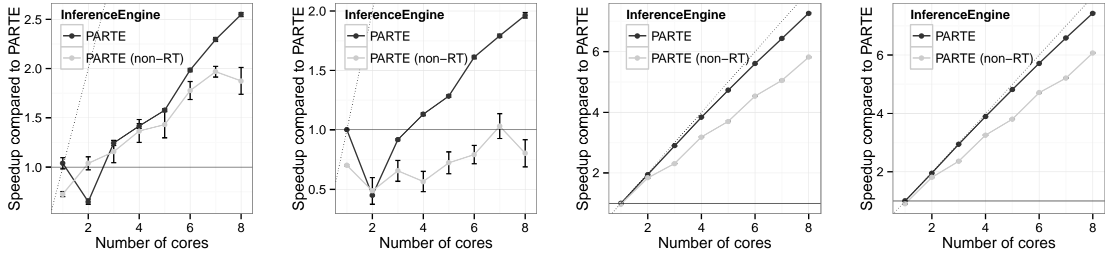
The second parameter of the benchmark is the number of concurrent event streams. Comparing figure 9a and 9c, we see that the scalability is not inhibited by the second event stream. However, the absolute scaling is lower when two event streams are used. Since the superlinear speedup only appears when an excess in computational resources is available, the speedup is decreased proportionally with the increased work load.

Note further that the non-real-time version of PARTE shows significant benefits over the real-time version in these benchmarks. The main reason is the changed scheduling strategy, which facilitates such search problems. However, while the absolute performance is improved, the non-real-time version also shows an increasing number of outliers, and thus has a significantly reduced predictability in performance.

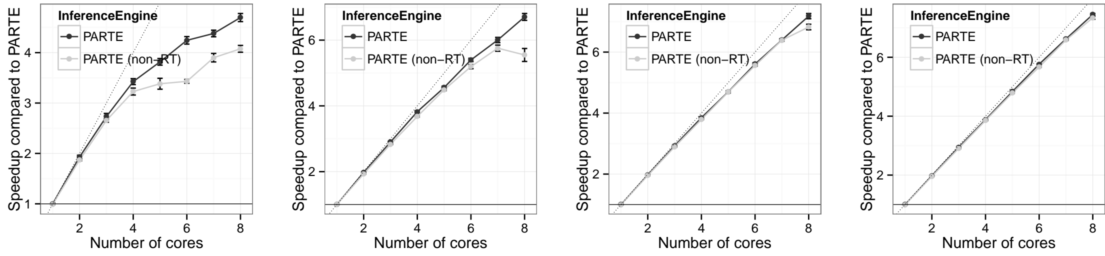
5.3.2. Scaling up to 64 Cores

In the previous section, we discussed the scaling of PARTE on up to 8 cores, since this is a common size for few-core systems, which do not yet experience the complexity of large-scale parallel systems. In this section, we discussed the scalability beyond such simple systems to up to 64 cores.

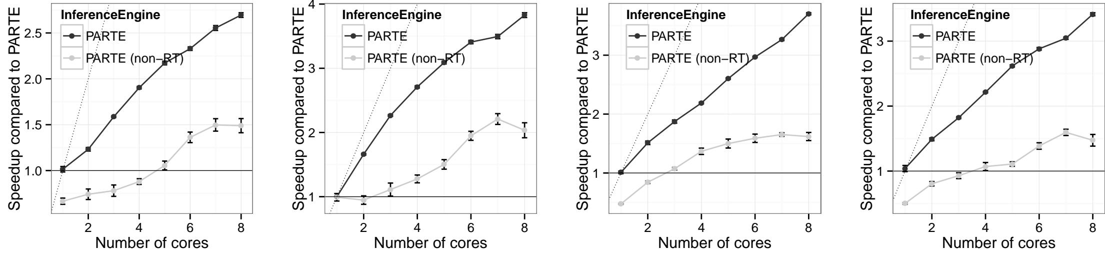
Figure 10 to 12 show the scaling of PARTE and PARTE (non-RT) on up to 64 cores. Overall the results indicate that the current design of PARTE scales only for a subset of the microbenchmarks, but can show superlinear speedups on the gesture recognition benchmarks.



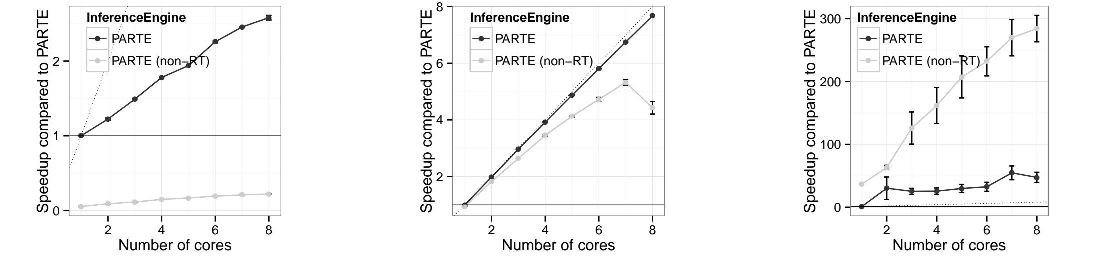
(a) Simple Tests (100) (b) Simple Tests (500) (c) Complex Tests (const) (d) Complex Tests (var)



(e) Heavy Tests (16) (f) Heavy Tests (32) (g) Heavy Tests (64) (h) Heavy Tests (128)

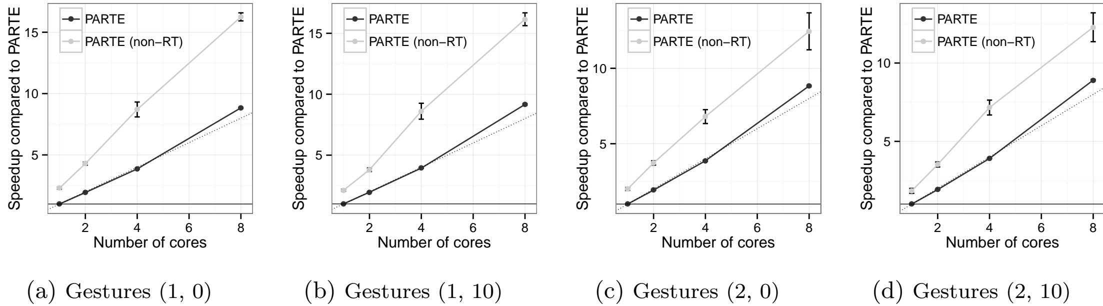


(i) Joining Tree (basic) (j) Joining Tree (unbound) (k) Joining Tree (bound) (l) Joining Tree (unified)



(m) Simple Tests (parallel) (n) Heavy Tests (parallel) (o) Search

Figure 8: Microbenchmarks: Scaling from 1 to 8 cores.



(a) Gestures (1, 0) (b) Gestures (1, 10) (c) Gestures (2, 0) (d) Gestures (2, 10)

Figure 9: Gesture Recognition Benchmarks: Scaling from 1 to 8 cores, with 1 or 2 event streams, and 0 or 10 noise events per relevant event.

Microbenchmarks. Figure 10a shows that PARTE scales up to 20 cores on the *Simple Tests*. Afterwards, the performance drops significantly. Main reason for this performance drop is scheduling overhead caused by the real-time design. While the non-real-time version experiences for instance pipeline stalls as well, it is able to utilize the additional computational resources more consistently, and does not experience the scheduling bottleneck for high core counts. However, while there is a bottleneck in the real-time version, the performance remains predictable, and the confidence interval remains consistently smaller than for the non-RT version.

Compared to the simple tests, the *Heavy Tests* scale much better. Figure 10b is a good example for the pipeline parallelism. Since most work in this benchmark is done in its 16 test nodes, it is able to scale up to 16 cores and then plateaus. However, the pipeline parallelism limits the scaling factor to 9.5x due to stalling on predecessors. For 128 heavy tests, the stalling becomes less relevant and we see a scaling factor of up to 30.5x. The *Heavy Tests (parallel)* benchmark, which contains nodes that are independent, and thus do not introduce pipeline stalls, shows linear scaling up to 52.5x despite doing less overall work than the 128 heavy tests.

The *Complex Tests (var)* benchmark shown in figure 11a exhibits near to perfect scaling for up to 20 cores due to its computational complexity, however then scaling is limited by the Rete engine and the pipeline parallelism, similar to the *Simple Tests (100)* benchmark.

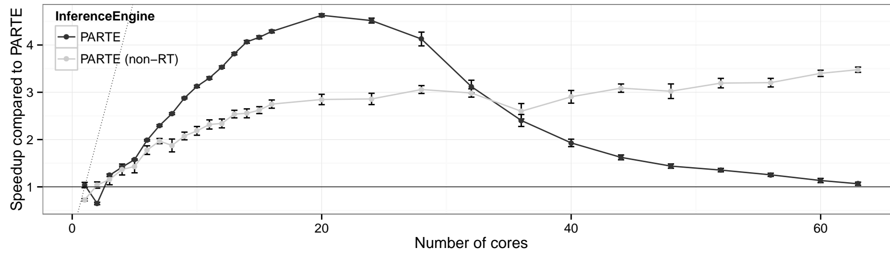
The *Joining Tree* benchmark in figure 11b suffers from the same bottlenecks, and the non-real-time version shows that the scheduling is again the biggest bottleneck. However, once more the soft real-time version of PARTE shows the more consistent and predictable results.

Finally, the *Search* benchmark can show up to 166.9x speedup for the soft real-time version. Note that for this benchmark the speedup is not expected to scale linearly with the number of cores, because at some point the maximum potential of the underlying scheduling strategy is reached.

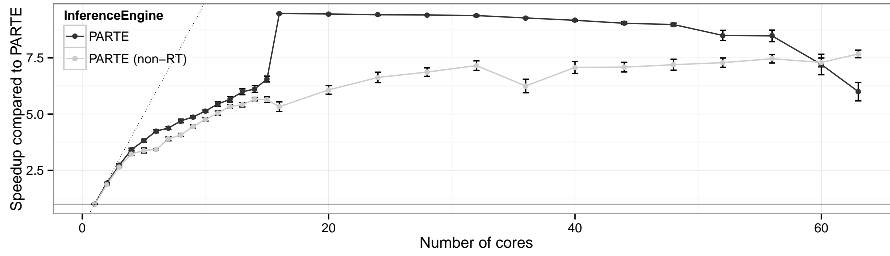
Gesture Recognition Benchmarks. As previously mentioned, the gesture recognition benchmarks are essentially search problems and benefit significantly from an abundance of parallelism. Figure 12 depicts the speedup potential. For the first two benchmarks with a single event stream, the Rete engine has sufficient remaining capacity to fully exploit the potential parallelism in the search problem. Figure 12a shows that the benchmark without additional noise events reaches a speedup of 556.0x. The benchmark with additional noise benefits even more and is up to 616.8x faster than the sequential execution.

Figure 12c and 12d show the performance when the workload is doubled and two concurrent event streams are fed into the system. Since the available parallelism did not double, the speedup is limited in these cases to 354.6x and 374.3x respectively.

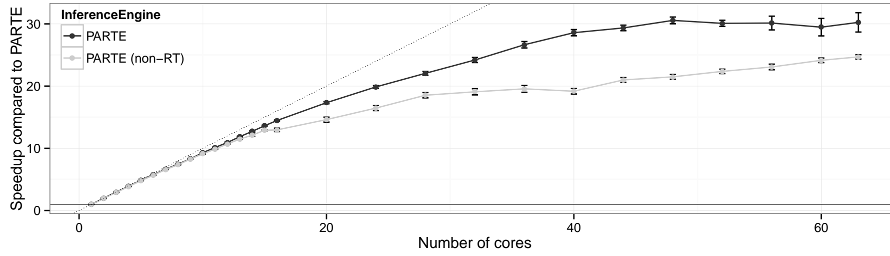
Finally, it remains to be noted that the non-real-time version performs better initially, but is eventually overtaken by the real-time version of PARTE. Thus, these benchmarks indicate that despite the theoretical drawbacks and bottlenecks PARTE can scale and provide the desired soft real-time guarantees as well as good performance.



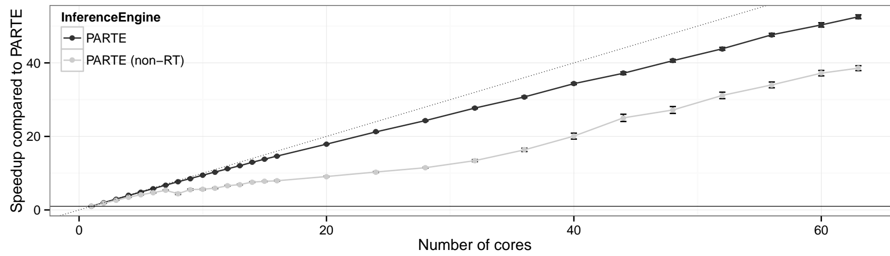
(a) Simple Tests (100)



(b) Heavy Tests (16)

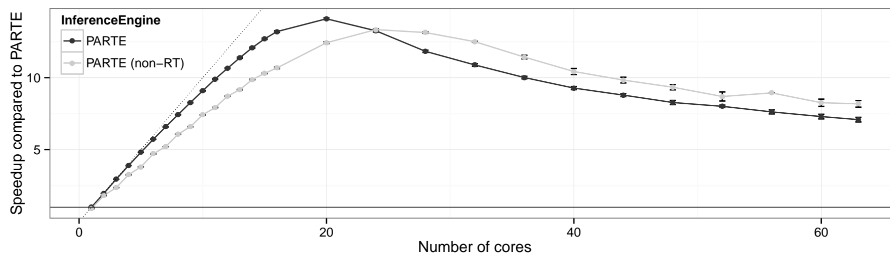


(c) Heavy Tests (128)

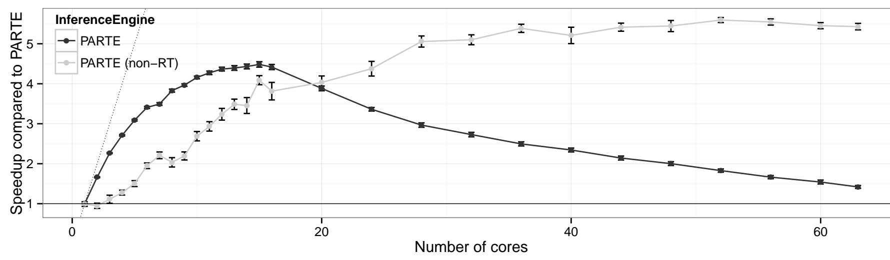


(d) Heavy Tests (parallel)

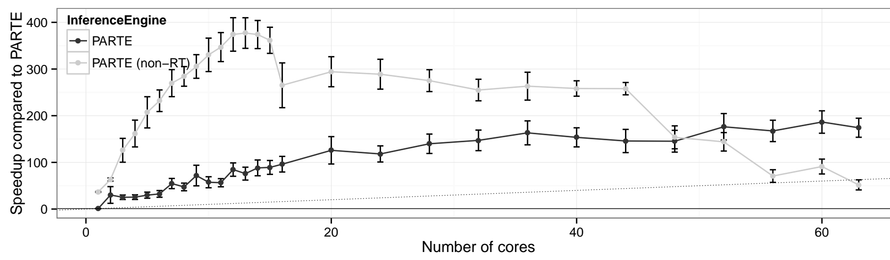
Figure 10: Microbenchmarks: Scaling from 1 to 64 cores.



(a) Complex Tests (var)

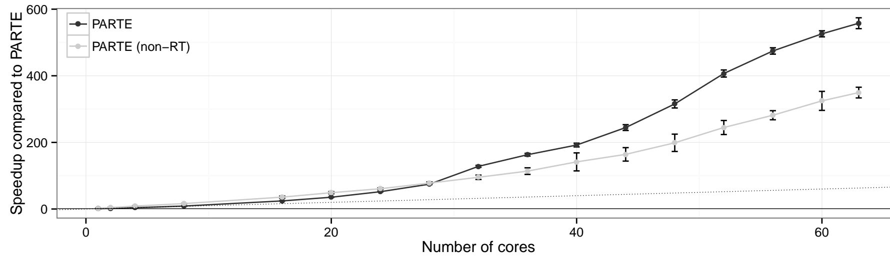


(b) Joining Tree (unbound)

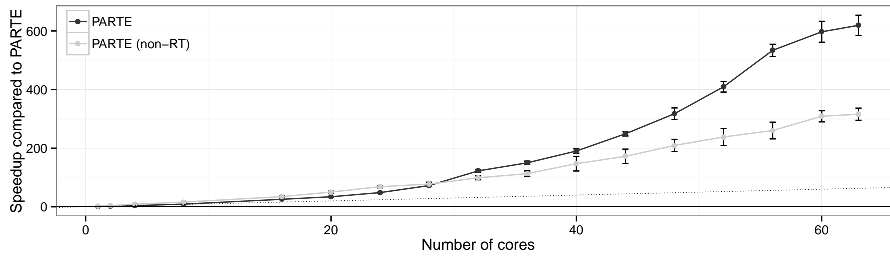


(c) Search

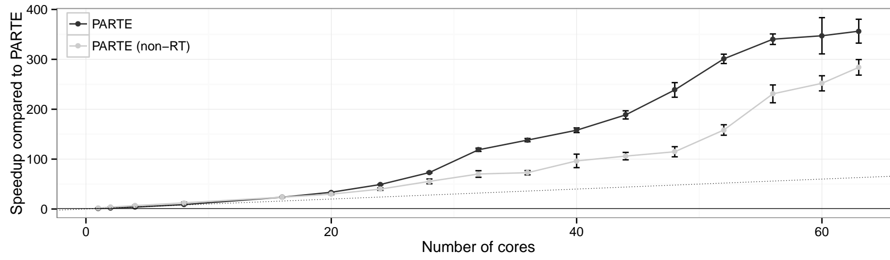
Figure 11: Microbenchmarks: Scaling from 1 to 64 cores.



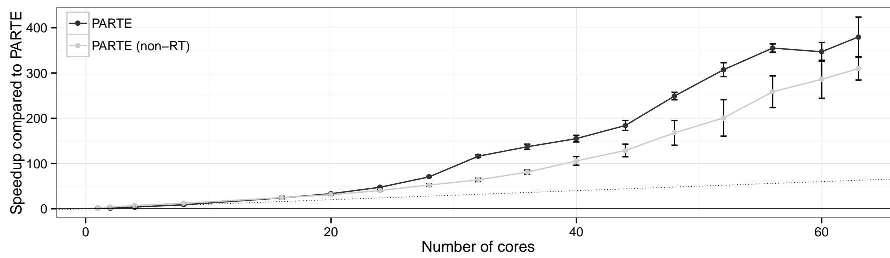
(a) Gesture Recognition, a single event stream, no noise



(b) Gesture Recognition, a single event stream, with noise



(c) Gesture Recognition, two event streams, no noise



(d) Gesture Recognition, two event streams, with noise

Figure 12: Gesture Recognition Benchmarks: Scaling from 1 to 64 cores.

5.4. Discussion

The performance evaluation indicates that PARTE scales for the microbenchmarks as well as the application benchmarks for up to 8 cores. Scaling on up to 64 cores depends on the benchmark and points out a bottleneck exists in the current scheduling mechanism.

However, PARTE's sequential performance compared to CLIPS requires improvement. Especially the gesture recognition benchmarks with a slowdown of about 25.3x would benefit from improved sequential performance. The main overhead here is the reification of nodes as actors and the corresponding scheduling overhead. One approach to reduce this overhead is to combine trivial nodes to avoid the overhead altogether. However, such coarsening of nodes could have an undesirable impact on the performance on parallel systems. For instance the superlinear speedup we observe for the gesture recognition benchmarks would most likely be reduced. Furthermore, computationally intensive nodes as for instance in the *Complex Tests* benchmarks could even benefit from being split into multiple nodes to expose more parallelism.

Another area that requires improvement is PARTE's scheduler for the actors, i. e., Rete nodes. Our performance analysis (cf. section 5) identified scheduling as the main bottleneck for a number of microbenchmarks. Furthermore, the results of the non-real-time version of PARTE give an additional indication for the impact of scheduling. Currently, PARTE's strategy focuses on correctness, simplicity, and an upper bound on the execution time to provide soft real-time guarantees. The main problem with the current scheduler is that actors are scheduled for execution independent of whether they have tokens in their inbox. An open challenge for PARTE is to find a scheduling strategy that does not require blocking synchronization, and reduces the scheduling overhead.

6. Related Work

The Rete algorithm is widely used, and has had many adaptations to both fit real-time requirements and to fit parallel processing.

6.1. Parallel Rete

Gupta et al. [19] measured that the matching phase is the most computationally expensive task in production systems, and takes up to 90% of the execution time. Consequently, most effort has been dedicated to parallelizing the match-phase.

One of the best known Rete derivatives focussing on parallelism is the TREAT algorithm by Miranker [20]. In TREAT, for every condition element, the matching facts are stored. This makes TREAT a state-saving algorithm, but less so than Rete, which in addition to those *alpha memories* stores the matching sets of facts for combinations of condition elements that appear in the rules, in *beta memories*. At the other end of the spectrum is the production system proposed by Oflazer [21], which stores the matching sets of facts for *every* combination of condition elements, regardless of whether they appear in the rules. Both TREAT and Oflazer's machine diverged from the traditional Rete algorithm to reduce the need for synchronization, thereby opening options for parallelism. Both approaches had the foreseeable drawbacks: TREAT spends a lot of time recomputing matches for entire patterns, and the combinatorial explosion made Oflazer's machine consume a lot of working memory, in addition to spending a large amount of time computing combinations of facts which may never get used. PARTE, which sticks to the traditional Rete algorithm, has none of these drawbacks. It does not decouple the different threads of execution by performing too little or too much work to be able to skip synchronizing, but instead focuses on reducing the overhead of the synchronization. In addition, it uses automatic expiration of facts, which halves the amount of inter-node communication that has to take place compared to Miranker's and Oflazer's system with manual retraction.

A different approach is taken by Aref and Tayyib [16], whose distributed Rete algorithm Lana is an optimistic algorithm, allowing the different processing elements to run with minimal synchronization, informing a single central *Master Fact List* of changes to the working memory, and backtracking when the updates of the multiple replicated Rete engines conflict. Unlike in PARTE, the different entities running in parallel in Lana are fixed, allowing less flexibility to redistribute workload among the available processing units, and by splitting up the Rete graph, common subgraphs cannot be shared by multiple rules, requiring Lana to

duplicate work where PARTE could reuse computations. Moreover, the optimistic approach adds a degree of nondeterminism to the run time which a real-time system like PARTE cannot risk to incur.

Yet other systems use hierarchical blackboard systems on which multiple agents concurrently work, and where every 'row' of nodes in the Rete network is reified as a different blackboard in the knowledge base's hierarchy. Examples of such systems are the Parallel Real-time Artificial Intelligence System (PRAIS) of Goldstein [15] and the Hierarchically Organized Parallel Expert System (HOPES) by Dai et al. [22]. Semantically similar, but not using the blackboard metaphor is for instance the parallel Rete system proposed by Gupta et al. [23]: they also acknowledged that having more than one token flowing through the graph at any one time could be opportune for the execution speed. In their paper, they propose to give every node one or more internal threads of control. Their approach is conceptually very close to ours, but is not designed for event processing. The goal was to create a parallel expert system that supports conflict-resolution strategies like sequential Rete implementations. Following from these design decisions, they had to omit all conceptual optimizations that depend on temporal reasoning as well as additional opportunities for parallelism in the evaluation of the rules' consequents. Furthermore, their approach depended on hardware task schedulers to enqueue and dequeue node activations in a timely manner.

In general, previous approaches did not use the abstraction of actors like PARTE does. Nodes were not considered as self-contained elements. Data-locality was achieved in an ad hoc manner, in Gupta's case expecting the presence of processor-local private memory in addition to the caches and main memory.

6.2. Real-Time Rete

Real-time execution characteristics can be achieved in two ways: *1*) by guaranteeing for every task that it completes in a known time frame, and scheduling them such that they all complete before their deadline; or *2*) by assigning priorities to tasks, and allowing the system to preempt lower-priority tasks to ensure higher-priority tasks complete in time.

PARTE takes this first approach, and ensures that every action taken by the inference engine completes in time – unless unexpected load is put on the system by entities other than PARTE. The Parallel Real-time Artificial Intelligence System (PRAIS) of Goldstein [15] instead takes the second approach, dropping the matching of lower-priority rules to allow more important rules to be matched in time. Despite being considered a real-time system, PRAIS can only offer best-effort guarantees, as it is a distributed system depending on TCP/IP for communication.

Sequential implementations such as the self-stabilizing OPS5 production system by Cheng and Fujii [24] do offer actual hard real-time guarantees, but their approach is not viable for our problem domain: it requires lots of effort in the generation of the ruleset to provide fault-tolerance on top of the real-time guarantees. Their system is aimed at situations where failure is catastrophic and must hence be avoided at all costs. PARTE does not pose such severe restrictions on the ruleset, and only requires *tiering*.

6.3. Multi-touch Software Frameworks

Richardson et al. [2] developed guidelines for the development of real-time multi-touch software applications. They start out with the very general guidelines of using operating systems and programming environments that support real-time applications. Utilizing a real-time operating system is a necessity for absolute real-time guarantees, especially in the case when PARTE is running on a system shared with other applications. Their suggestion to use real-time Java does however not apply to PARTE. Being implemented in C++, PARTE has full control over allocation and avoids therefore the non-determinism caused by a garbage collector. It also has direct access to the necessary platform mechanisms to for instance obtain time values and clocks, manage memory, and influence scheduling.

Richardson et al. further suggest to bound the number of simultaneous multi-touches and to avoid computationally expensive and long gestures. While these suggestions are sensible for a traditional approach of gesture recognition, with PARTE neither of these constraints are necessary. The use of a Rete network provides the capabilities to handle large numbers of gestures efficiently. The Rete network also handles complex, i. e., long gestures efficiently, because the chosen memory management strategy makes sure that a Rete node can manage state locally and discard expired facts automatically. This avoids the state management

issues classic approaches have with supporting long gestures. The parallel execution of the Rete network further makes sure that computationally expensive gestures are handled gracefully. Computationally expensive tests bind only the processor core, i. e., the work thread that handles the corresponding test node of the Rete network, while other parts of the network are processed in parallel. Thus, while the amount of computationally intensive tests that can be handled depends on the available computational resources, such tests do not negatively impact any of the unrelated gesture patterns and their recognition. As such, the overall system can still fulfill its real-time guarantees.

7. Conclusions

The presented PARTE inference engine implements a variation of the Rete algorithm using actor semantics for Rete nodes to achieve parallel execution. The system is designed for continuous gesture recognition which requires soft real-time execution guarantees and scalability on parallel systems. While PARTE utilizes the constraints of the domain to achieve these properties, it remains applicable to the broader domains of *Time Series Analysis* and *Complex Event Processing* because they have very similar properties with respect to event streams and performance requirements. This includes applications such as algorithmic stock trading [25–27] and monitoring network security [28, 29]. PARTE's key design choices are a tiered architecture, the use of lock-free queues, and an actor execution model to provide upper bound guarantees on the event matching in a Rete network.

PARTE is compatible with existing single-threaded inference engines such as NASA's CLIPS. As a replacement, PARTE provides the benefits of transparent parallelization of declarative rules as well as automatic event expiration. It can be used to replace CLIPS for instance in the core infrastructure of the multimodal Mudra framework [1]. Mudra itself has been used among other things to realize augmented reality effects for live music performances based on rule-based gesture definitions [30].

The single core performance evaluation based on microbenchmarks and multi-touch gesture recognition benchmarks shows that PARTE is on average 2.1x slower than CLIPS for microbenchmarks and gesture benchmarks combined. In its current unoptimized state, it has a significant performance overhead for Rete networks with many trivial nodes, which needs to be addressed in future work.

PARTE scales for microbenchmarks and gesture benchmarks on up to 8 cores. On 64-core systems, the soft real-time scheduling approach presents a bottleneck for some of the microbenchmarks. However, the gesture recognition benchmarks scale exceptionally well, because they can utilize the superlinear speedup of a parallel search problem. On 64 cores, they reach a speedup of up to 616.8x.

The evaluation of PARTE's soft real-time properties on these benchmarks indicates more consistent and predictable performance compared to PARTE's non-real-time version, making it more suitable for the intended use case of interactive gesture recognition.

Future work on PARTE needs to address the absence of support for negation rules. This would broaden the range of possible applications, and performance for gestures that are mutually exclusive. With support for negated rules, less superfluous matches need to be performed, resulting in better sequential performance and avoiding scheduling-related overhead and bottlenecks.

Overall, we conclude that PARTE is the first scalable soft real-time Rete engine that supports the domains of gesture recognition as well as complex event processing. The provided declarative language and sliding time window notion offer relevant software engineering properties for these domains. Furthermore, its scalability properties prepare it for the multi- and manycore era where performance improvements for sequential implementations will be limited.

Acknowledgments

Lode Hoste is supported by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium. Stefan Marr is supported by the MobiCra$^{\mathrm{NT}}$ project funded by INNOV$^{\mathrm{IRIS}}$.

Appendix A. Benchmark Characterization

This appendix details the benchmarks used for the performance evaluation of section 5. The benchmark implementations and test data is made available at `http://soft.vub.ac.be/~trenaux/PARTE/`.

Appendix A.1. Synthetic Benchmarks

These benchmarks are microbenchmarks evaluating the performance characteristics of specific implementation choices. The results of these benchmarks enable a comparison of different implementation strategies with regard to specific technical choices. However, they do not provide results that are generalizable and sufficient to compare the overall performance of complete systems.

Appendix A.1.1. Pipeline Parallelism

These microbenchmarks exercise different aspects of a Rete implementation. In case the Rete implementation supports parallelism, these benchmarks can benefit from pipeline parallelism. The Rete networks built in these benchmarks are a strictly linear chain of nodes, where the nodes can work in parallel on *different* facts. However, the network does not allow to process *the same fact* multiple times in parallel at any given point in time.

Simple Tests (100, 500) N simple tests are represented by separate Rete nodes. The test nodes form a linear dependency chain. Each test compares two constant numbers.

Such tests are for instance relevant to model typical gestures and their properties. Event ordering based on their timestamp is a common example, but orientation of objects, or application-specific data attributes such as type or color of an object are commonly implemented with simple, computationally trivial tests.

Goal: Evaluate the cost of inter-node communication and the benefit of pipeline parallelism.

Rules and Facts: The benchmark consists of a single rule that captures a fact identifier, then performs the tests, and finally checks the fact identifier for equality with the fact terminating the benchmark. 5000 facts are asserted and the last one will trigger the rule to complete the benchmark.

Expected Behavior: Most *sequential* Rete implementations will not impose significant cost for inter-node communication, because it does not need to be reified as such. Therefore, the benchmark is most likely measuring the effectiveness of data representation and the cost of comparing two numbers. A *parallel* Rete implementation will most likely need to communicate explicitly between Rete nodes either by passing some data token or by synchronizing access to shared mutable state. For these, the benchmarks measure the communication overhead which is expected to be the significant part of the cost. Compared to the sequential version, a parallel version might be able to do multiple tests in the dependency chain in parallel, benefiting from the theoretically possible pipeline parallelism. An execution of a parallel implementation on a single core represents one of the worst possible cases.

Variants: Number of constant comparisons N: 100, 500

Complex Tests (const, var) Similar to the Simple Tests benchmark, 100 test nodes are chained sequentially. However, the tests are in this case binary expressions nested nine levels deep, adding numbers.

This benchmark resembles rules that contain a high amount of application logic. Especially in a prototyping phase it is useful to directly encode complex spatial or temporal logic in the rules. However, as this benchmark shows, the interpretation of such logic is computationally intensive.

Goal: Evaluate the efficiency of the expression evaluator and compare the tradeoffs between inter-node communication and pipeline parallelism.

Rules and Facts: The benchmark is structured similar to the Simple Tests benchmark. It evaluates the 100 complex tests in a chain, and eventually a 101st test checks the fact identifier to terminate the benchmark after processing the last fact.

The benchmark is used in two variants: The first variant uses a simple constant inlined into the expressions. The second variant uses a variable, and thus requires a lookup cost to obtain the value for the addition.

Expected Behavior: A *sequential* Rete implementation will measure in addition to the aspects of the Simple Tests the efficiency of its expression evaluation. A *parallel* Rete implementation should show better scaling depending on the efficiency of its expression evaluator. If a highly efficient expression evaluators is used, the dominating overhead will be still the necessary inter-node communication.

Variants: Value of addition: a constant (const), or a variable requiring lookup (var)

Heavy Tests (16, 32, 64, 128) N tests perform computationally intensive operations, which are supposed to represent the computational cost template or machine learning based test nodes would have. For instance, Dynamic Time Warping [31] and Hidden Markov Models [32] are well known gesture recognition algorithms and can be used inside a rule to strengthen the accuracy of the classification. Other uses could be to implement complex logic that has a high interpretation overhead when it is implemented as done in the in the *Complex Tests*.

The test nodes form a linear dependency chain. Each test operates completely independently.

While the workload used in this benchmark is purely synthetic, machine learning-based tests for motion detection would require a similar amount of computation and are trivially parallelizable as well. Furthermore, the computationally intensity of this benchmark is lower than that of the Complex Tests.

Goal: Evaluate the best-case scenario for parallel execution by performing independent computationally expensive operations that are trivially parallelized.

Rules and Facts: The benchmark uses the same composition as the Simple Test benchmark. 500 facts are asserted and the last one will trigger the rule to complete the benchmark.

Expected Behavior: A *sequential* Rete implementation is expected to show only insignificant overhead over the pure runtime it takes to execute the tests. A *parallel* Rete implementation is expected to show ideal scaling with the number of available processing cores. The communication and synchronization overhead is expected to be minimal and insignificant.

Variants: Number of tests N: 16, 32, 64, 128

Appendix A.1.2. Independent Tasks

This section describes microbenchmarks that are designed to provide parallelism on the level of the Rete network. Thus, instead of resulting in a linear chain of nodes, the rules used in these benchmarks result in Rete networks that have multiple branches which allows the same fact to be processed multiple times at the same time in different parts of the network.

Simple Tests (parallel) Evaluating the impact of multiple independent rules next to each other. The main part of the benchmark consists of ten independent instances of the previously described version of the *Simple Tests (100)* benchmark.

This benchmark corresponds to standard applications that use multiple sets of independent rules to process incoming events recognizing different patterns.

Goal: The goal is to evaluate whether the additional parallelism can in any way offset the cost for inter-node communication.

Rules and Facts: A first rule matches the incoming fact and replicates it to the then independent instances of the Simple Tests rule, which corresponds to a linear chain of 100 simple test nodes. A final rule matches the completion of all ten independent rules. The benchmark asserts 5000 facts into the Rete engine and the last one will trigger the rule to complete the benchmark.

Expected Behavior: For a *sequential* implementation, this benchmark is expected to perform as the previous variations of Simple Tests. For a *parallel* implementation it is a worst case scenario that

measures the overhead of inter-node communication and synchronization. While the ten main parts of the Rete network are independent and do not require any communication, the need for communication is ten times as high as in the previous version of Simple Tests. Thus, it measures mostly the efficiency of communication. The additional parallelism in the network is not expected to lead to different execution characteristics, since the previous versions already benefited from pipeline parallelism.

Variants: It is a parallel variant of *Simple Tests (100)*

Heavy Tests (parallel) Evaluating the impact of multiple independent computationally intensive rules next to each other. The main part of the benchmark are ten independent instance of the *Heavy Tests (8)* benchmark.

This benchmark corresponds to complex motion recognition invocations, such as time-series alignment or machine learning algorithms.

Goal: The goal is to evaluate the best case of a computationally intensive test in such a highly parallel network.

Rules and Facts: The benchmark is structured the same as the *Simple Tests (parallel)* benchmark but instead of trivial tests uses eight computationally intensive tests for the ten independent chains of tests.

Expected Behavior: A *sequential* implementation should spent most of the runtime on the execution of the computationally expensive tests. All other operations should have a comparably negligible cost. A *parallel* implementation should be able to show perfect scaling with the number of available processing cores. The communication overhead should be negligible.

Variants: It is a parallel variant of *Heavy Tests (8)*

Appendix A.1.3. Parallel Tasks with Complex Joins

The microbenchmarks of this category evaluate the cost of different language features such as the presence of slots, binding slots to variables, and unifying sets of those variables. Furthermore, it uses parallelism between independent rules for performance, but also evaluates the impact of eventually having to join the independent execution to form a higher-level result (cf. subsection 2.1 for a motivation for tiering).

Joining Tree (basic, unbound, bound, unified) A network built from eight rules that contains independent and dependent parts matching sequences of facts to produce higher-level events.

This simulates a common scenario of independent low-level rules that are used on higher levels to match complex patterns. However, this synthetic benchmark has no computational complexity.

Goal: Assess performance and scalability for complex rule-sets with multi-tier interactions and measure the impact of unbound slots, bound slots, and unifying slots.

Rules and Facts: The rules constitute three independent chains of match and test nodes on the first tier. These feed higher-level facts into a second tier, which also consists of three independent chains of match and test nodes. The third level matches groups of facts from all three chains and eventually signals completion of the benchmark once all facts have been processed. The benchmark initially asserts 181 facts to trigger the benchmark completion exactly once.

The *unbound* variant uses fact definitions with five extra integer slots that are not used. The *bound* variant uses fact definitions with five extra integer slots that are used once, and thus bound but not unified. The *unified* variant uses fact definitions with five extra integer slots that are bound and unified.

Expected Behavior: A *sequential* implementation will measure purely the effects of the increasing complexity of the rules between the variants of the benchmark.

A *parallel* implementation should benefit from the available parallelism. However, the rules are trivial and do not contain computationally expensive operations. While the overall complexity increases, it is most likely not enough to offset the inherent overhead of the inter-node communication.

Variants: basic, unbound, bound, unified

Appendix A.1.4. Search Problems

This section introduces a microbenchmark that corresponds to a traditional search-problem, which can lead to superlinear speedups. Rete-based forward-chaining needs to process all nodes at the top of a network to find matches, while often only one of these entry nodes leads to a match. Parallelizing this search for this entry node can lead to speedups based on the fact that the search starts in parallel on multiple nodes, and thus, can select the matching node by chance much earlier.

Search Evaluating the impact of parallel execution of a search problem.

In large systems of rules, it is essential for good performance that independent rules can be processed in parallel and trigger their matches as soon as possible. Finding a match as early as possible in a Rete network is analogous to search problems in the field of artificial intelligence. An oracle could select a perfect order to traverse the network depending on the asserted fact, while classic Rete engines chose typically a traversal based on the network and independent of the fact.

Note, while this benchmark is synthetic, it is not artificial. Especially in the context of gesture recognition, the rule sets need to include the complete set of gestures, potentially in variations. We took the gesture set of Wobbrock et al. [33] and implemented rules to process the input. This leads to at least one rule for every symbol. To support the English alphabet, a rule set would therefore contain at least 52 rules to represent the lowercase and uppercase letters, of which only one rule at a time would be triggered while writing. Another example is the rule set for the six mid-air gestures describing the augmented reality effects presented by Hoste and Signer [30]. The gestures are described by 41 rules in total with minimal overlap between rules so that only one rule is triggered at a time.

Goal: Evaluate the effect of parallel execution on classic search problems that should be able to show superlinear speedups.

Rules and Facts: The Rete network is built from 50 rules that perform computationally expensive tests but never trigger their right-hand side. Rule 51 immediately matches the last asserted fact and terminates the benchmark. Overall 5000 facts are asserted into the network.

Expected Behavior: A *sequential* implementation is expected to exhibit worst-case behavior by executing all tests for all facts. A *parallel* implementation is expected to terminate much earlier and execute only a subset of all tests completely, because the final fact can be processed earlier and trigger the 51st rule immediately to complete the benchmark.

Appendix A.2. Application Benchmarks

In order to evaluate the performance as a gesture recognition engine, we rely on a data set of unistroke pen gestures made available by Wobbrock et al. [33]. The pen gestures have been recorded in a user study on an HP iPAQ Pocket PC and consists of a set of 16 different gestures for which ten subjects were asked to draw them in three different speeds. Thus, overall the data set contains 480 recordings of pen gestures with varying accuracy.

Sixteen gesture rules were defined by an expert developer based on ideas presented by Hoste et al. [1, 34]. These rules use *control points*, i. e., points by which the major characteristics of a gesture can be described. Each rule uses four of these control points to define a single gesture and allow combinations of events to be found in a continuous event stream. Through unification in the rule definition, the control points will only match data originating from the same finger.

In addition to creating the rules, we convert the data provided by Wobbrock et al. [33] into facts compatible with CLIPS and PARTE. The facts are fed into the system as concurrent streams to simulate multiple fingers gesturing at the same time.

Gestures (streams, noise) Evaluate the application performance based on existing gesture definitions.

The goal of the multi-touch gesture rules is to extract gesture candidates from a continuous stream of information. Since the begin and end event of such a multi-touch gesture are not known beforehand,

the problem is comparably computationally intensive and relies on the state-saving of the Rete engine to be done efficiently.

Goal: Evaluate the efficiency with which actual multi-touch gestures can be recognized.

Rules and Facts: The benchmark consists of 16 gestures rules. Each rule consisted of nine temporal and spatial operators. As the data consists of 2D coordinates, the temporal and spatial operators relate to the use of Simple Tests (cf. Appendix A.1.1).

The number of concurrent event streams is a parameter to the benchmark. It simulates the interaction of one or more fingers on an interactive surface.

A second parameter, *noise*, defines the number of synthetic touch events that are asserted in the system per every sample. In the domain of gesture recognition, noise represents irrelevant data and is an important factor of the accuracy of the recognition task.

Expected Behavior: A *sequential* Rete engine should perform comparably well in these benchmarks, because the used rules are have characteristics that are similar to other application use cases such as expert system, in that they consist of a number of conditions and unification operations. Thus, they essentially perform the operations common Rete engines have been optimized for.

A *parallel* Rete engine is most likely having an overhead when execute with a single core, compared to an optimized sequential implementation. However, it is expected to show superlinear speedups, because this type of pattern matching is essentially a search problem. As long as the number of parallel execution units is higher than the number of event streams, a parallel Rete engine should be able to gain significant benefits from the parallel execution.

Variants: Number of event streams S: 1, 2; number of noise events per relevant event N: 0, 10

References

[1] L. Hoste, B. Dumas, B. Signer, Mudra: A Unified Multimodal Interaction Framework, in: Proceedings of ICMI 2011, 13th International Conference on Multimodal Interaction, Alicante, Spain, 97–104, 2011.

[2] T. Richardson, L. Burd, S. Smith, Guidelines for supporting real-time multi-touch applications, Software: Practice and Experience ISSN 1097-024X.

[3] T. Hammond, R. Davis, LADDER, A Sketching Language for User Interface Developers, Computers & Graphics 29 (4) (2005) 518–532, ISSN 0097-8493.

[4] C. Scholliers, L. Hoste, B. Signer, W. De Meuter, Midas: A Declarative Multi-Touch Interaction Framework, in: Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction, TEI '11, ACM, New York, NY, USA, ISBN 978-1-4503-0478-8, 49–56, 2011.

[5] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence 19 (1) (1982) 17–37, ISSN 0004-3702.

[6] R. M. Akscyn, D. L. McCracken, E. A. Yoder, KMS: a distributed hypermedia system for managing knowledge in organizations, Communications of the ACM 31 (7) (1988) 820–835, ISSN 00010782.

[7] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, A. Blake, Real-time human pose recognition in parts from single depth images, in: CVPR, vol. 2, 7, 2011.

[8] R. B. Miller, Response time in man-computer conversational transactions, in: AFIPS '68 (Fall, part I) Proceedings of the December 9-11, 1968, fall joint computer conference, part I, ACM Press, New York, New York, USA, 267, 1968.

[9] G. Agha, Concurrent object-oriented programming, Commun. ACM 33 (9) (1990) 125–141, ISSN 0001-0782.

[10] T. L. Harris, A pragmatic implementation of non-blocking linked-lists, Springer, ISBN 3-540-42605-1, 2001.

[11] M. M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the twenty-first annual symposium on Principles of distributed computing - PODC '02, ACM Press, New York, New York, USA, ISBN 1581134851, 21, 2002.

[12] A. A. El-Haj Mahmoud, Hard-Real-Time Multithreading: A Combined Microarchitectural and Scheduling Approach, Ph.D. thesis, North Carolina State University, aAI3223132, 2006.

[13] J. W. S. Liu, Real-Time Systems, Prentice Hall, Upper Saddle River, NJ, ISBN 0130996513 9780130996510, 2000.

[14] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, in: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA '07, ACM, ISBN 978-1-59593-786-5, 57–76, 2007.

[15] D. Goldstein, Extensions to the Parallel Real-Time Artificial Intelligence System(PRAIS) for fault-tolerant heterogeneous cycle-stealing reasoning, in: Proceedings of the 2nd Annual CLIPS ('C' Language Integrated Production System) Conference, NASA. Johnson Space Center, Houston, 287–293, 1991.

[16] M. M. Aref, M. A. Tayyib, LanaMatch algorithm: a parallel version of the ReteMatch algorithm, Parallel Computing 24 (5-6) (1998) 763–775, ISSN 0167-8191.

[17] P. Kampstra, Beanplot: A Boxplot Alternative for Visual Comparison of Distributions, Journal of Statistical Software, Code Snippets 28 (1) (2008) 1–9, ISSN 1548-7660.

[18] P. J. Fleming, J. J. Wallace, How Not to Lie With Statistics: The Correct Way to Summarize Benchmark Results, Commun. ACM 29 (1986) 218–221, ISSN 0001-0782.

[19] A. P. Gupta, C. L. Forgy, D. Kalp, A. Newell, Parallel OPS5 on the Encore Multimax, Proceedings of the International Conference on Parallel Processing (1988) 271–280.

[20] D. P. Miranker, TREAT: a better match algorithm for AI production systems, in: Proceedings of the sixth National conference on Artificial intelligence - Volume 1, AAAI'87, AAAI Press, ISBN 0-934613-42-7, 42–47, 1987.

[21] K. Oflazer, Partitioning in ParalleI Processing of Production Systems, Ph.D. thesis, Carnegie Mellon University, 1985.

[22] H. Dai, T. J. Anderson, F. C. Monds, On the implementation issues of a parallel expert system, Information and Software Technology 34 (11) (1992) 739–755, ISSN 09505849.

[23] A. Gupta, C. Forgy, A. Newell, R. Wedig, Parallel algorithms and architectures for rule-based systems, in: Proceedings of the 13th annual international symposium on Computer architecture, ISCA '86, IEEE Computer Society Press, ISBN 0-8186-0719-X, ISSN 01635964, 28–37, 1986.

[24] A. Cheng, S. Fujii, Bounded-response-time self-stabilizing OPS5 production systems, in: Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, IEEE Comput. Soc, ISBN 0-7695-0574-0, 399–404, 2000.

[25] M. Tirea, I. Tandau, V. Negru, Multi-agent Stock Trading Algorithm Model, 2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing 0 (2011) 365–372.

[26] T. Laffey, P. Cox, J. Schmidt, S. Kao, J. Readk, Real-Time Knowledge-Based Systems, AI Magazine 9 (1), ISSN 0738-4602.

[27] H. Abdullah, M. Rinne, S. Törmä, E. Nuutila, Efficient Matching of SPARQL Subscriptions using Rete, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, ACM, New York, NY, USA, ISBN 978-1-4503-0857-1, 372–377, 2012.

[28] J. Moskal, C. Matheus, Detection of Suspicious Activity Using Different Rule Engines Comparison of BaseVISor, Jena and Jess Rule Engines, in: N. Bassiliades, G. Governatori, A. Paschke (Eds.), Rule Representation, Interchange and Reasoning on the Web, vol. 5321 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-88807-9, 73–80, 2008.

[29] C. Kloukinas, G. Spanoudakis, A Pattern-Driven Framework for Monitoring Security and Dependability, in: C. Lambrinoudakis, G. Pernul, A. Tjoa (Eds.), Trust, Privacy and Security in Digital Business, vol. 4657 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-74408-5, 210–218, 2007.

[30] L. Hoste, B. Signer, Expressive Control of Indirect Augmented Reality During Live Music Performances, in: Proceedings of NIME 2013, 13th International Conference on New Interfaces for Musical Expression, Daejeon + Seoul, Korea Republic, 2013.

[31] H. Sakoe, S. Chiba, Dynamic programming algorithm optimization for spoken word recognition, Acoustics, Speech and Signal Processing, IEEE Transactions on 26 (1) (1978) 43 – 49, ISSN 0096-3518.

[32] A. Wilson, A. Bobick, Parametric Hidden Markov Models for Gesture Recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence 21.

[33] J. O. Wobbrock, A. D. Wilson, Y. Li, Gestures Without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes, in: Proceedings of the 20th annual ACM symposium on User interface software and technology, UIST '07, ACM, New York, NY, USA, ISBN 978-1-59593-679-0, 159–168, 2007.

[34] L. Hoste, , B. De Rooms, B. Signer, Declarative Gesture Spotting Using Inferred and Refined Control Points, in: Proceedings of the 2nd International Conference on Pattern Recognition Applications and Methods, ICPRAM'13, Barcelona, Spain, 6, 2013.