Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code?

Sung-Shik Q. Jongmans, Farhad Arbab

Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code?

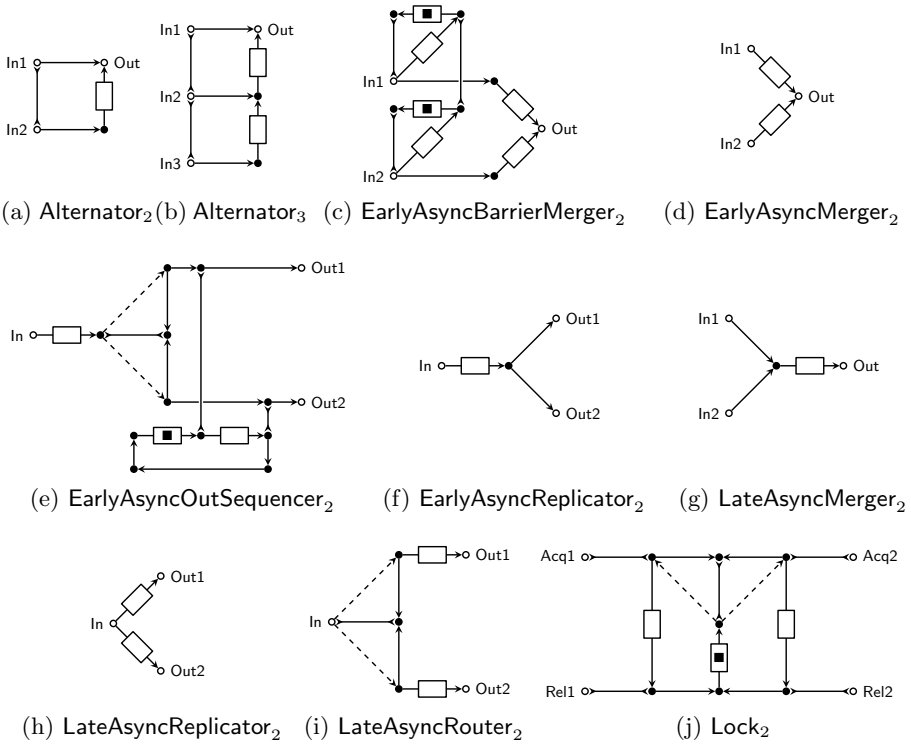
Sung-Shik T.Q. Jongmans and Farhad Arbab

Centrum Wiskunde & Informatica, Amsterdam, Netherlands

Abstract. High-level concurrency constructs and abstractions have several well-known software engineering advantages when it comes to programming concurrency protocols among threads in multicore applications. To also explore their complementary performance advantages, in ongoing work, we are developing compilation technology for a high-level coordination language, Reo, based on this language's formal automaton semantics. By now, as shown in our previous work, our tools are capable of generating code that can compete with carefully hand-crafted code, at least for some protocols. An important prerequisite to further advance this promising technology, now, is to gain a better understanding of how the significantly different compilation approaches that we developed so far, which vary in the amount of parallelism in their generated code, compare against each other. For instance, to better and more reliably tune our compilers, we must learn under which circumstances parallel protocol code, with high throughput but also high latency, outperforms sequential protocol code, with low latency but also low throughput.
In this paper, we report on an extensive performance comparison between these approaches for a substantial number of protocols, expressed in Reo. Because we have always formulated our compilation technology in terms of a general kind of communicating automaton (i.e., constraint automata), our findings apply not only to Reo but, in principle, to any language whose semantics can be defined in terms of such automata.

1 Introduction

Context. A promising application domain for coordination languages is programming protocols among threads in multicore applications. One reason for this is a classical software engineering advantage: coordination languages typically provide high-level constructs and abstractions that more easily compose into correct—with respect to programmers' intentions—protocol specifications than do conventional lower-level synchronization mechanisms (e.g., locks or semaphores). However, not only do coordination languages simplify programming protocols, but their high-level constructs and abstractions also leave more room for compilers to perform optimizations that conventional language compilers cannot apply. Eventually, sufficiently smart compilers for coordination languages should be capable of generating code (e.g., in Java or in C) that can compete with carefully hand-crafted code. Preliminary evidence for feasibility of this goal



(a) Alternator$_2$ (b) Alternator$_3$ (c) EarlyAsyncBarrierMerger$_2$ (d) EarlyAsyncMerger$_2$

(e) EarlyAsyncOutSequencer$_2$ (f) EarlyAsyncReplicator$_2$ (g) LateAsyncMerger$_2$

(h) LateAsyncReplicator$_2$ (i) LateAsyncRouter$_2$ (j) Lock$_2$

Fig. 1: Example connectors (ordered alphabetically)

appears elsewhere [13]. A crucial step toward adoption of coordination languages for multicore programming, then, is the development of such compilers.

To study the performance advantages of using coordination languages for multicore programming, in ongoing work, we are developing compilation technology for the coordination language Reo [1,2]. Reo facilitates compositional construction of protocol specifications manifested as *connectors*: channel-based mediums through which threads can communicate with each other. Figure 1 shows a number of example connectors in their usual graphical syntax. Briefly, a connector consists of one or more *channels*, through which data items flow, and a number of *nodes*, on which channel ends coincide. Reo features an open-ended set of channels, which means that programmers can define their own channels with custom semantics. Figure 1, for instance, includes standard synchronous channels (normal arrows) and asynchronous channels with a 1-capacity buffer (rectangle-decorated arrows), among others. Nodes, in contrast, have fixed semantics. Threads can perform blocking I/O operations—`put` and `get`—on the named *public nodes* of a connector, while a connector uses its anonymous *private nodes* only for internal routing. Section 2 provides a more detailed overview of Reo; Section 4 explains the behavior of the connectors in Figure 1.

Figure 2 shows one of our most promising achievements in developing compilation technology so far [13]. It shows the performance of three k-producer-single-consumer protocol implementations in C, for $k \in \{2^i \mid 2 \leq i \leq 9\}$: one naive hand-written implementation (continuous line), one optimized hand-written implementation (dashed line), and one implementation compiled from a Reo connector (dotted line). In every round of this protocol, every producer sends one data item to the consumer. Once the consumer has received a data item from every producer, in any order, it sends an acknowledgement to the producers, thereby signaling that the consumer is ready for the next round. (The Reo connector for this protocol, for $k = 2$, resembles $\mathsf{EarlyAsyncBarrierMerger}_2$ in Figure 1c.) This example shows that already our current compilation technology is capable of generating code that can compete with—and in this case even outperform—carefully hand-crafted code. Surely, our technology is not yet mature enough to achieve similarly positive results in *every* multicore application, for *every* connector. Nevertheless, this example offers preliminary evidence that programming protocols among threads using high-level constructs and abstractions can result in equally good—or better—performance as compared to conventional low-level, manual techniques.

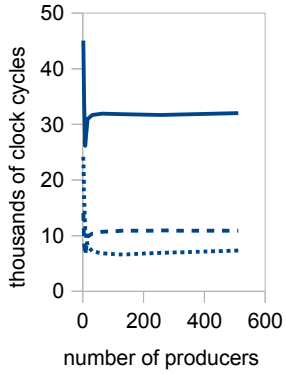


Fig. 2: Earlier results [13]

Problem. Despite our encouraging first results, a long road still lies ahead of us before we reach the point at which our tools can compile every connector into high-performance code, the following step of which we try to take in this paper.

In the Reo literature, three different approaches for compiling Reo connectors exist [11]. In the *distributed approach*, a compiler implements the behavior of each of the k constituents of a connector (i.e., its nodes and its channels) and runs these k implementations in parallel as a distributed system; in the *centralized approach*, a compiler computes the behavior of a connector as a whole, implements this behavior, and runs this implementation sequentially as a centralized system. The distributed approach has maximal parallelism, and it has the advantage of fast compilation at build-time and high throughput at run-time. However, this comes at the cost of higher latency at run-time (because of a necessary distributed consensus algorithm). In contrast, the centralized approach has maximal sequentiality, and it has the advantage of low latency at run-time. However, this comes at the cost of slower compilation and lower throughput. Moreover, centralized-approach compilers may generate an amount of code exponential in k, which may make their output prohibitively large and the time to produce it prohibitively long. Proença et al. observe that a partially-distributed, partially-centralized *hybrid approach*, where a compiler splits a connector into parts, implements those parts according to the centralized approach, and runs those implementations according to the distributed approach, is generally ideal [16,17]: a hybrid approach strikes a middle ground between latency and throughput at

run-time while achieving reasonably fast compilation at build-time.

We started developing a centralized-approach compiler and gradually moved to a hybrid-approach version, mainly motivated by the latter's advantages at build-time. Before this paper, however, we had only little understanding of the implications with respect to run-time performance. Moreover, in recent work [11], we found a case where hybrid-approach compilation actually took much longer than centralized-approach compilation. This made us realize that we *must* improve our understanding of the differences between the centralized approach and the hybrid approach to advance our compilation technology.

Contribution & Organization. In this paper, we compare centralized-approach compilation and execution with hybrid-approach compilation and execution. For this, we use nine different connector "families" (i.e., connectors parametric in the number of the coordinated threads), "members" of which Figure 1 shows. Our comparison reveals previously unknown strengths and weaknesses of the approaches under investigation. These new insights are imperative for the future development of our compilation technology and, consequently, for evidencing the performance merits of high-level constructs and abstractions for multicore programming, complementary to their classical software engineering advantages.

Although framed in the context of Reo, our technology works at the level of Reo's formal automaton semantics. This means that we have formulated and implemented our compilers in terms of a general kind of communicating automaton. Therefore, our findings apply to compilation technology not only for Reo but for any high-level model or language whose semantics one can define in terms of such automata (e.g., some process calculi). We expect this generality to make our work interesting to a larger audience, beyond the Reo community.

In Section 2, we discuss preliminaries on Reo and its automaton semantics. In Section 3, we present a centralized-approach and a hybrid-approach compiler for Reo, which we implemented from scratch (though conceptually based on earlier implementations). In Section 4, we explain our experimental setup. In Sections 5 and 6, we discuss our experimental results: in Section 5, we discuss results related to the *compilation* of our experimental connectors, while in Section 6, we discuss results related to their *execution*. Section 7 concludes this paper.

2 Preliminaries

Reo is a language for compositional construction of concurrency protocols, manifested as connectors [1,2]. Connectors consist of channels and nodes, organized in a graph-like structure. Every channel consists of two ends and a constraint that relates the timing and the content of the data-flows at those ends. A channel end has one of two types: *source ends* accept data (i.e., a source end of a channel connects to that channel's data source/producer), while *sink ends* dispense data (i.e., a sink end of a channel connects to that channel's data sink/consumer). Reo makes no other assumptions about channels and allows, for instance, channels with two source ends. Table 1 shows four common channels.

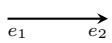
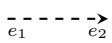
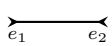
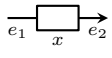
Syntax	*Semantics*
	Synchronously takes a data item d from its source end e_1 and writes d to its sink end e_2.
	Synchronously takes a data item d from its source end e_1 and nondeterministically either writes d to its sink end e_2 or loses d.
	Synchronously takes data items from both its source ends and loses them.
	Asynchronously $[$takes a data item d from its source end e_1 and stores d in a buffer $x]$, then $[$writes d to its sink end e_2 and clears $x]$.

Table 1: Graphical syntax and informal semantics of common channels

Channel ends coincide on nodes. Contrasting channels, every node behaves in the same way: repeatedly, it nondeterministically selects an available data item out of one of its coincident sink ends and replicates this data item into each of its coincident source ends. A node's nondeterministic selection and its subsequent replication constitute one atomic execution step; nodes cannot temporarily store, generate, or lose data items. Threads can perform blocking I/O operations on the *public nodes* of a connector: `put` operations enable threads to send data, while `get` operations enable threads to receive data. In Figure 1, we distinguish the white, named public nodes of a connector from its shaded, anonymous *private nodes*. Before a connector makes a global execution step, usually instigated by pending I/O operations, its channels and its nodes must have reached consensus about their behavior to guarantee mutual consistency of their local execution steps (e.g., a node should not replicate a data item into a channel with an already full buffer). Afterward, connector-wide data-flow emerges.

Through *composition*, programmers can construct arbitrarily complex connectors out of simpler ones. As Reo supports both synchronous and asynchronous channels, connector composition enables mixing synchronous and asynchronous communication within the same protocol.

Our compilers generate code for Reo connectors based on their *constraint automaton* (CA) semantics [4]. Constraint automata are a general formalism for modeling concurrent systems, better suited for modeling Reo connectors—and their composition in particular—than classical automata or traditional process calculi. For Reo, a CA specifies *when* during execution of a connector *which* data items flow *where* (i.e., through which channel ends). Structurally, every CA consists of finite sets of states and transitions, which model a connector's internal configurations and atomic execution steps. Every transition has a label that consists of two elements: (i) a set with the names of those channel ends that have synchronous data-flow (ii) and a logical formula that specifies which particular data items may flow through which of those ends. In such formulas, $d(e_1) = d(e_2)$ means that the same data item flows through e_1 and e_2. In practice, we associate every node and every channel with an elementary CA for its behavior. Figure 3

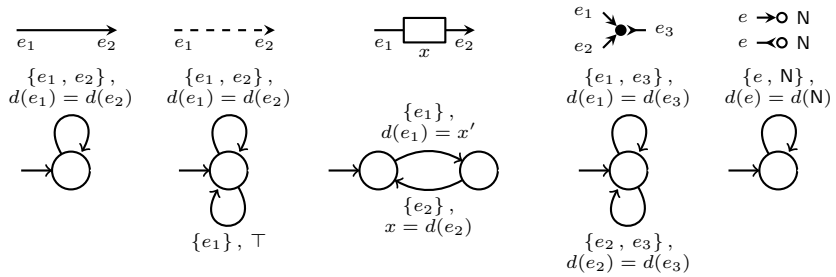
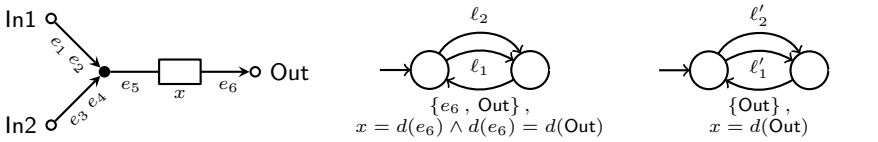


Fig. 3: Constraint automata for the channels in Table 1 (first three from the left), for a private node with two incoming and one outgoing channel (fourth from the left), and for two public nodes, each with either one incoming or one outgoing channel (fifth from the left). The latter CA is defined not only over the names of its coincident channel ends but also over its own name. (Threads use node names—not channel end names—to perform I/O operations on, and therefore, public node names must explicitly occur in their CA semantics.)



(a) $\mathsf{LateAsyncMerger}_2$ (b) Composition (c) Comp. and abstr.

$\ell_1 : \{\mathsf{In1}, e_1, e_2, e_5\}$, $d(e_1) = d(e_2) \wedge d(e_2) = d(e_5) \wedge d(e_5) = x'$
$\ell_2 : \{\mathsf{In2}, e_3, e_4, e_5\}$, $d(e_3) = d(e_4) \wedge d(e_4) = d(e_5) \wedge d(e_5) = x'$
$\ell'_1 : \{\mathsf{In1}\}$, $d(\mathsf{In1}) = x'$
$\ell'_2 : \{\mathsf{In2}\}$, $d(\mathsf{In2}) = x'$

(d) Transition labels

Fig. 4: Composition and abstraction of $\mathsf{LateAsyncMerger}_2$ in Figure 1g

shows example CAs. A *product operator* on CAs subsequently models connector composition: to obtain the "big" CA for a whole connector, one can incrementally form the product of the "small" CAs for its constituent nodes and channels. Afterward, one can abstract away private nodes' coincident channel ends with a *hide operator* on CAs [4], which also eliminates internal transitions involving only such ends. Figure 4 shows the composite CA of $\mathsf{LateAsyncMerger}_2$.

3 Compilers

Our compilers operate fully at the level of Reo's CA semantics. Our focus on Reo so far in this paper is therefore misleading: we use Reo's graphical, channel-based abstractions, just as *a*—not *the*—programmer-friendly syntax for exposing CA-based protocol programming. Different syntax alternatives for CAs may

work equally well or yield perhaps even more user-friendly languages. For instance, we know how to translate UML sequence/activity diagrams and BPMN to CAs [3,7,15]. Algebras of Bliudze and Sifakis [6], originally developed for BIP [5], also have a straightforward interpretation in terms of CAs, thereby offering an interesting alternative possible syntax. Due to their generality, CAs can thus serve as an intermediate format for compiling specifications in many different languages and models of concurrency, by reusing the core of our compilers. This makes the development of our compilation technology relevant beyond Reo.

For our performance comparison, based on earlier implementations [10,14], we developed two Reo/CA-to-Java compilers as mentioned already in Section 1: a centralized-approach one, henceforth referred to as $\texttt{Compiler}_{\text{centr}}$, and a hybrid-approach one, henceforth referred to as $\texttt{Compiler}_{\text{hybr}}$. (Both compilers are available on request.) Both compilers generate shared-memory Java code, geared toward multicore execution. On input of a connector, $\texttt{Compiler}_{\text{centr}}$ (i) first finds a small CA for every channel and every node that this connector consists of, (ii) then forms the product of all those CAs to get a big CA for the whole connector, abstracting away all internal details in the process, and (iii) finally generates one piece of sequential code for that big CA. At run-time, this piece of code logically has its own thread. (Physically, however, we can optimize this "protocol thread" away by letting "computation threads" perform its work.) Essentially, the construction of a big CA in this way corresponds to parallel expansion in process algebra [8]. $\texttt{Compiler}_{\text{hybr}}$ also first finds a set of small CAs, but in contrast to $\texttt{Compiler}_{\text{centr}}$, it does not form their product to get a big CA. Instead, it computes an m-size *partition* of this set. By doing so, $\texttt{Compiler}_{\text{hybr}}$ effectively splits a connector into a number of "regions" (i.e., connected subconnectors), each of which has a corresponding subset in the partition. After computing a partition, $\texttt{Compiler}_{\text{hybr}}$ forms products on a per-region basis, which results in m "medium" CAs, and generates a piece of sequential code for each of them. At run-time, every such piece of code has its own thread. These threads use shared-memory (plus concurrency protection) to synchronize their actions whenever necessary.

$\texttt{Compiler}_{\text{hybr}}$'s partitioning algorithm iterates over the set of small CAs and incrementally extends its computed partition (starting from an empty one) [9,14]. For every small CA α, the algorithm decides either to add $\{\alpha\}$ to the partition (as a new singleton subset) or to add α to one or more existing parts. (In the latter case, the algorithm subsequently merges all extended subsets into one new subset.) Jongmans et al. formulated the condition based on which the algorithm makes this decision generally, in terms of CAs and their transitions. In the context of Reo, however, this partitioning algorithm precisely coincides with the identification of *synchronous/asynchronous regions* of a connector [17] (each of which gets a corresponding subset in the partition). The asynchronous regions of a connector are its smallest connected subconnectors that have only asynchronous data-flow (e.g., the fourth channel in Table 1). By removing the asynchronous regions from a connector, its pairwise disconnected synchronous regions remain: connected subconnectors with synchronous data-flow. Intuitively, asynchronous regions decouple synchronous regions. Such decoupling enables synchronous re-

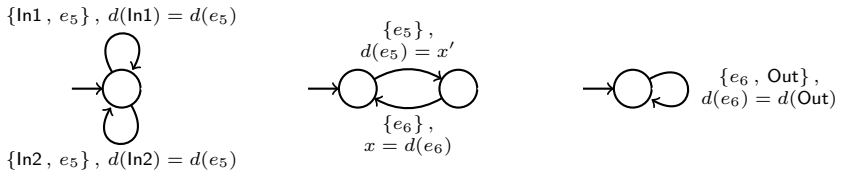


Fig. 5: Medium CAs that result from applying the partitioning algorithm to $\mathsf{LateAsyncMerger}_2$ (see also Figure 4. The middle CA represents the asynchronous channel in the middle (i.e., one asynchronous region). The leftmost CA represents the synchronous region left of the asynchronous channel (i.e., three nodes, two channels). It repeatedly makes a choice between its two inputs and passes the data item from that input into the asynchronous channel (i.e., into buffer x). The rightmost CA represents the synchronous region right of the asynchronous channel (i.e., only one node). It repeatedly passes a data item from the asynchronous channel (i.e., from buffer x) to its output.

gions to run independently of each other: communication between synchronous regions always proceeds in an asynchronous fashion, through a shared asynchronous region. Figure 5 shows the medium CAs that result from applying the previous partitioning algorithm to $\mathsf{LateAsyncMerger}_2$, composing CAs on a per-subset basis, and abstracting away private nodes (see also Figure 4). Note that a connector without asynchronous regions consists of one comprehensive synchronous region. For such connectors, $\texttt{Compiler}_{\text{hybr}}$ reduces to $\texttt{Compiler}_{\text{centr}}$.

Notably, a connector represents the logic behind—not the architecture of—the data-flow in a protocol. For instance, even though Lock_2 in Figure 1j, which represents a classical lock, consists of a mix of synchronous, asynchronous, and lossy channels, its compiler-generated code uses neither physical hardware channels nor virtual software channels to realize its desired behavior.

4 Experimental Setup

Practical details. To study under which circumstances code generated by $\texttt{Compiler}_{\text{hybr}}$ outperforms code generated by $\texttt{Compiler}_{\text{centr}}$, we performed a number of experiments. In every experiment, we compared the performance of centralized and hybrid implementations of a k-parametric connector family, for $k \in \{2\,, 4\,, 6\,, 8\,, 10\,, 12\,, 14\,, 16\,, 32\,, 48\,, 64\}$. Figure 1 shows the $k = 2$ members of the nine connector families that we investigated. (One can extend these $k = 2$ members to their $k > 2$ versions in a similar way as how we extended Figure 1a to Figure 1b.) We selected these families because each of them exhibits different behavior in terms of (a)synchrony, exclusion, nondeterminism, polarity, sequentiality, and parallelism, thereby aiming for a balanced comparison. In total, thus, we investigated 99 different connectors and twice as many Java implementations. We ran every implementation nine times on a machine with 24 cores (two Intel E5-2690V3 processors with twelve physical cores statically at 2.6 GHz in two

sockets, hyperthreading disabled) and averaged our measurements. In every run, we warmed up the JVM for thirty seconds before starting to measure the number of "rounds" that an implementation could finish in the subsequent four minutes. What constitutes one round differs per connector; see below.

Primarily, we wanted to study and measure the overhead of the synchronization algorithm between the protocol threads in the hybrid implementations (which increases their latency) relative to those implementations' increased parallelism (which increases their throughput). To focus our measurements on only that particular aspect, we needed to eliminate as much as possible all other, orthogonal sources of computation inside compiler-generated code. One notable such source is data processing: although both our compilers support compilation of *data-sensitive connectors*, whose behavior may depend on the particular data items that pass through them, we nevertheless compiled all connectors in a data-insensitive fashion. This ensured that no data processing occurred at run-time during our experiments, which would have constituted a substantial source of sequential, unoptimized computation, even though we already know of ways to significantly improve this. If we would have enabled data processing, its irrelevant—at least to this comparison—overhead would have polluted our measurements. Perhaps even worse, our results would become obsolete the moment we implement our upcoming data processing optimizations.

For convenience, we divided the connector families under study—except Lock—over two categories: k-producer-single-consumer and single-producer-k-consumer. Both of these categories consist of four families. The k-producer-single-consumer category contains LateAsyncMerger (cf. Figure 1g), EarlyAsyncMerger (cf. Figure 1d), EarlyAsyncBarrierMerger (cf. Figure 1c), and Alternator (cf. Figures 1a and 1b); the single-producer-k-consumer category contains LateAsyncReplicator (cf. Figure 1h), EarlyAsyncReplicator (cf. Figure 1f), LateAsyncRouter (cf. Figure 1i), and EarlyAsyncOutSequencer (cf., Figure 1e).

Connectors. Next, we explain the behavior of the connectors in Figure 1. We start with explaining the k-producer-single-consumer connector families. With $\mathsf{LateAsyncMerger}_k$ (cf. Figure 1g), whenever producer i `put`s a data item on its local node Ini, the connector stores this data item in its only buffer (unless this buffer is already filled by another producer, in which case the `put` suspends until the buffer becomes empty). The relieved producer can immediately continue, possibly before the consumer has completed a `get` for its data item (i.e., communication between a producer and the consumer transpires asynchronously). Whenever the consumer `get`s a data item from its local node Out, the connector empties the previously full buffer. The consumer `get`s data items in the order in which producers `put` them (i.e., communication between a producer and the consumer transpires transactionally, i.e., undisrupted by other producers). Every round consists of a `put` by a producer and a `get` by the consumer; in every round, two transitions fire.

With $\mathsf{EarlyAsyncMerger}_k$ (cf. Figure 1d), whenever a producer i `put`s a data item on its local node Ini, the connector stores this data item in its corresponding buffer. The relieved producer can immediately continue, possibly before the

consumer has completed a `get` for its data item (i.e., communication between a producer and the consumer transpires asynchronously). Whenever the consumer `get`s a data item from its local node Out, the connector empties one of the previously full buffers, selected nondeterministically. The consumer does not necessarily `get` data items in the order in which producers `put` them (i.e., communication between a producer and the consumer transpires not necessarily transactionally). Every round consists of a `put` by a producer and a `get` by the consumer; in every round, two transitions fire.

Connectors in the EarlyAsyncBarrierMerger family work in largely the same way as those in the EarlyAsyncMerger family, except that the former enforce a barrier on the producers: no producer can `put` its n-th data item until all other producers have `put` their $(n-1)$-th data items. The consumer may still `get` data items in an order different from the order in which the producers `put` them. Every round consists of a `put` by every producer and k `get`s by the consumer, one for every producer; in every round, $2k$ transitions fire.

With $\mathsf{Alternator}_k$ (cf. Figures 1a and 1b), whenever a producer i attempts to `put` a data item on its local node Ini, this operation suspends until both (1) the consumer attempts to `get` a data item from its local node Out, and (2) every other producer j attempts to `put` a data item on its local node Inj (i.e., the producers can `put` only synchronously). Once each of the producers and the consumer attempt to `put`/`get`, the consumer `get`s the data item sent by the top producer (i.e., communication between the top producer and the consumer transpires synchronously), while the connector stores the data items of the other producers in their corresponding buffers (i.e., communication between the other producers and the consumer transpires asynchronously). Afterward, the consumer `get`s the remaining buffered data items in the spatial top-to-bottom order of the producers. Every round consists of a `put` by every producer and k `get`s by the consumer, one for every producer; in every round, k transitions fire.

We proceed with explaining the single-producer-k-consumer connector families. With $\mathsf{EarlyAsyncReplicator}_k$ (cf. Figure 1f), whenever the producer `put`s a data item on its local node In, the connector stores this data item in its only buffer. The relieved producer can immediately continue, possibly before the consumers have completed `get`s for its data item (i.e., communication between the producers and a consumer transpires asynchronously). Whenever a consumer i attempts to `get` a data item from its local node Out$i$, this operation suspends until both (1) the buffer has become full, and (2) every other consumer attempts to `get` a data item (i.e., the consumers can `get` only synchronously). Once the buffer has become full and each of the consumers attempts to `get`, every consumer `get`s the data item in the buffer, while the connector empties that buffer. Every round consists of a `put` by the producer and a `get` by every consumer; in every round, two transitions fire.

With $\mathsf{LateAsyncReplicator}_k$ (cf. Figure 1h), whenever the producer `put`s a data item on its local node In, the connector stores a copy of this data item in each of its buffers. The relieved producer can immediately continue, possibly before the consumers have completed `get`s for its data item (i.e., communication

between the producers and a consumer transpires asynchronously). Whenever a consumer i gets a data item from its local node Out$i$, the connector empties its corresponding full buffer. Every round consists of a put by the producer and a get by every consumer; in every round, $k+1$ transitions fire.

With LateAsyncRouter$_k$ (cf. Figure 1i), whenever the producer puts a data item on its local node In, the connector stores this data item in exactly one of its buffers (instead of a copy in each of its buffers as LateAsyncReplicator$_k$ does), selected nondeterministically. The relieved producer can immediately continue, possibly before the consumer of the selected buffer has completed a get for its data item (i.e., communication between the producer and a consumer transpires asynchronously). Whenever a consumer i gets a data item from its local node Out$i$, the connector empties its corresponding full buffer. The consumers do not necessarily get data items in the order in which the connector stored those data items in its buffers. Every round consists of a put by the producer and a get by a consumer; in every round, two transitions fire.

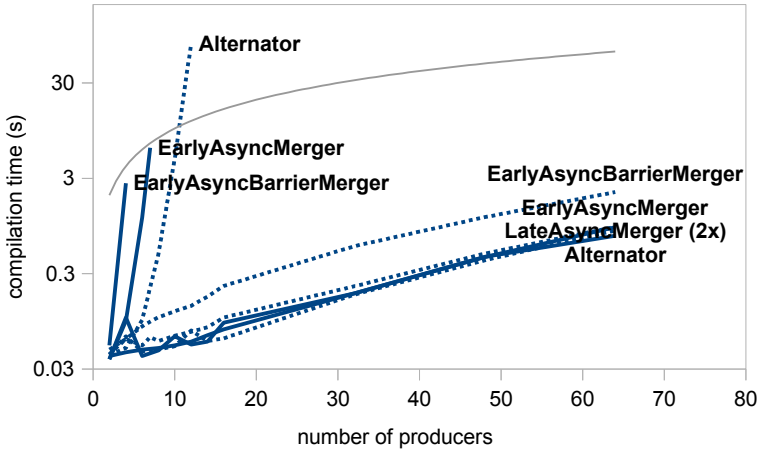
With EarlyAsyncOutSequencer$_k$ (cf. Figure 1e), whenever the producer puts a data item on its local node In, the connector stores this data item in its leftmost buffer. The relieved producer can immediately continue, possibly before a consumer has completed a get for its data item (i.e., communication between a producer and the consumers transpires asynchronously). The connector ensures that the consumers can get only in the top-to-bottom sequence. Whenever a consumer i gets a data item from its local node Out$i$, the connector empties its corresponding full buffer. Every round consists of k puts by the producer and a get by every consumer; in every round, $2k$ transitions fire.

Finally, Lock$_k$ represents a classical lock (cf. Figure 1j). To acquire the lock, a computation thread i puts an arbitrary data item (i.e., a signal) on its local node Acq$i$; to release the lock, this thread puts an arbitrary data item on its local node Rel$i$. A put on Acq$i$ suspends until every computation thread j that previously performed a put on Acq$j$ has performed its complementary put on Rel$j$ (i.e., the connector guarantees mutual exclusion). Every round consists of two puts by one of the k producers; in every round, two transitions fire.

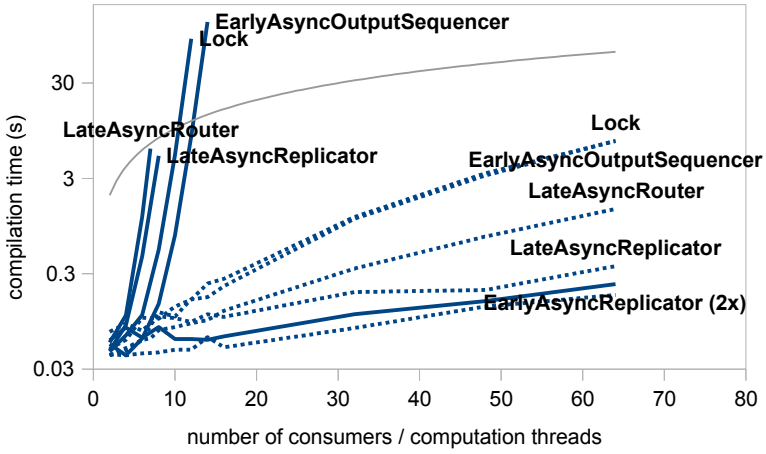
5 Experimental Results: Compilation

Measurements. We used $\texttt{Compiler}_{\text{hybr}}$ and $\texttt{Compiler}_{\text{centr}}$ to compile the connector families in Figure 1 for the aforementioned values of k with a transition limit of 8096 and a timeout after five minutes. We imposed a transition limit, because the Java compiler cannot conveniently handle Java code generated for CAs with so many transitions; we imposed a compilation timeout, because waiting for longer than five minutes to compile a single connector in practice seems unacceptable to us. Figure 6 shows the measured compilation times; see also [12].

For most connector families, $\texttt{Compiler}_{\text{hybr}}$ required substantially less time than $\texttt{Compiler}_{\text{centr}}$. In fact, for six of our nine connector families, $\texttt{Compiler}_{\text{centr}}$ failed to run to completion beyond certain (relatively low) values of k, as witnessed also by their very steep curves in Figure 6:



(a) k-producer-single-consumer



(b) single-producer-k-consumer and Lock_k

Fig. 6: Compilation times (continuous lines for $\texttt{Compiler}_{\text{centr}}$; dotted lines for $\texttt{Compiler}_{\text{hybr}}$; gray lines for proportional growth $x = y$, just as a reference)

- For $\mathsf{EarlyAsyncMerger}_{k>7}$, $\mathsf{LateAsyncReplicator}_{k>8}$ and $\mathsf{LateAsyncRouter}_{k>7}$, the transition number of their "big" CAs exceeded the limit (e.g., $\mathsf{EarlyAsyncMerger}_8$ has 23801 transitions, $\mathsf{LateAsyncReplicator}_9$ has 19172 transitions, and $\mathsf{LateAsyncRouter}_8$ has 23801 transitions) or the compiler timed out.
- For $\mathsf{EarlyAsyncBarrierMerger}_{k>4}$, $\mathsf{EarlyAsyncOutSequencer}_{k>14}$, and $\mathsf{Lock}_{k>12}$, the compiler timed out.

In contrast, $\texttt{Compiler}_{\text{hybr}}$ had no problems compiling these connector families

for all values of k under investigation. For $\mathsf{LateAsyncMerger}_k$ and $\mathsf{EarlyAsyncReplicator}_k$, our two compilers required a comparable amount of time for all values of k under investigation. Finally, only for $\mathsf{Alternator}_k$, $\texttt{Compiler}_{\text{hybr}}$ required substantially more time than $\texttt{Compiler}_{\text{centr}}$ does. In this case, $\texttt{Compiler}_{\text{hybr}}$ timed out for $k > 12$, while $\texttt{Compiler}_{\text{centr}}$ had no problems.

Discussion. In Section 1, we stated that hybrid-approach compilers have the advantage of "reasonably fast compilation at build-time" compared to centralized-approach compilers. The idea behind this statement is that the formation of a big CA in the centralized approach requires much computational resource, notably when state spaces or transition relations of such big CAs grow exponentially in k; hybrid-approach compilers usually avoid this, because hybrid-approach compilers do not compute big CAs. Intuitively, the medium CAs computed for a connector by hybrid-approach compilers are typically much smaller than its big CA. After all, each of those medium CAs consists of fewer small CAs than does this big CA. (The big CA consists of every small CA that also constitutes a medium CA.) Thus, in cases of exponential growth, medium CAs typically have a much smaller exponent than their corresponding big CA. A quick look at our measurements in Figure 6a seems to confirm this intuition: all six connector families for which $\texttt{Compiler}_{\text{centr}}$ eventually failed require exponentially more time as k increases. Beyond this quick look, however, there are peculiarities that need clarification.

A first, obvious peculiarity are the measurements for Alternator, which $\texttt{Compiler}_{\text{hybr}}$—instead of $\texttt{Compiler}_{\text{centr}}$—eventually fails for. Actually, we already made a preliminary *qualitative* analysis of this phenomenon in a recent workshop contribution [11]; our current *quantitative* results fully support the anecdotal analysis in that extended abstract. To save space—an in-depth explanation requires significantly more details of the partitioning algorithm used in hybrid-approach compilation—we only briefly summarize the cause of this phenomenon. Essentially, a hybrid-approach compiler cannot treat every private node of a connector as truly private: $\texttt{Compiler}_{\text{hybr}}$ cannot use the hide operator to abstract away those private nodes that mark the boundaries between a connector's regions. After all, protocol threads for neighboring regions synchronize their transitions through those nodes, which makes explicitly representing them in compiler-generated code essential. But because those private nodes must remain, also many internal transitions remain, potentially to the extent that they cause the transition relation of "medium" CAs formed for certain problematic regions to explode. This happens with Alternator. $\texttt{Compiler}_{\text{centr}}$, in contrast, can incrementally hide *all* private nodes to neutralize this source of explosion.

The second peculiarity concerns centralized-approach compilation. First, by analyzing the big CAs of the k-parametric connector families EarlyAsyncBarrierMerger, EarlyAsyncMerger, EarlyAsyncOutSequencer, LateAsyncReplicator, and LateAsyncRouter, we found that those CAs grow *exponentially* as k increases (due to the many ways in which their k independent transition can concurrently fire). This explains why $\texttt{Compiler}_{\text{centr}}$ requires exponentially more time as k increases to compile members of those families, as shown in Figure 6. Now, it seems not unreasonable to assume also the inverse: for k-parametric connector

families whose big CAs grow only *linearly* in k, $\texttt{Compiler}_{\text{centr}}$ should scale fine. Alternator, EarlyAsyncReplicator, and LateAsyncMerger, which satisfy its premise, seem to validate this assumption. Indeed, Figure 6 shows that $\texttt{Compiler}_{\text{centr}}$ has no problems with compiling members of those families. (The big CAs of the EarlyAsyncReplicator family even have a constant number of transitions.) However, this still leaves us with two families whose compilation behavior we have not yet accounted for: EarlyAsyncOutSequencer and Lock. Although the big CAs of both these k-parametric families grow only linearly in k, Figure 6 shows that $\texttt{Compiler}_{\text{centr}}$ nevertheless requires exponentially more time as k increases.

It turns out that even if big CAs grow only linearly in k, the "intermediate products" during their formation may "temporarily" grow exponentially. For instance, if we have three CAs α_1, α_2 and α_3, the intermediate product of α_1 and α_2 may grow exponentially in k, while the full product of α_1, α_2, and α_3 grows only linearly. This is easiest to explain for $\mathsf{EarlyAsyncOutSequencer}_k$ (cf. Figure 1e), in terms of its number of states. $\mathsf{EarlyAsyncOutSequencer}_k$ consists of a subconnector that, in turn, consists of a cycle of k buffered channels (of capacity 1). The first buffered channel initially contains a dummy data item ■ (i.e., its actual value does not matter); the other buffered channels initially contain nothing. As in the literature [1,2], we call this subconnector $\mathsf{Sequencer}_k$. Because no new data items can flow into $\mathsf{Sequencer}_k$, only ■ cycles through the buffers—ad infinitum—such that only one buffer holds a data item at any time. Consequently, the CA for $\mathsf{Sequencer}_k$ has only k states, each of which represents the presence of ■ in a different one of the k buffers. However, when $\texttt{Compiler}_{\text{centr}}$ compositionally computes this CA out of a number of smaller CAs by forming their product, it closes the cycle only with the very last application of the product operator: until that moment, the "cycle" still looks to the compiler as an open-ended chain of buffered channels. Because new data items can freely flow into it, such an open-ended chain can have a data item in any buffer at any time. Consequently, the CA for the largest chain (i.e., the chain of $k-1$ buffered channels, just before it becomes closed) has 2^{k-1} states. Only when $\texttt{Compiler}_{\text{centr}}$ forms the product of [the CA of the k-th buffered channel] and [the previously formed CA for the chain of $k-1$ buffered channels], the state space of 2^{k-1} states collapses into k states, as the compiler "finds out" that the open-ended chain is actually an input-closed cycle with exactly one data item. Clearly, because $\mathsf{Sequencer}_k$ constitutes $\mathsf{EarlyAsyncOutSequencer}_k$, also $\mathsf{EarlyAsyncOutSequencer}_k$ itself suffers from this problem. A similar argument applies to Lock_k.

Thus, even for k-parametric connector families whose big CAs grow only linearly in k, $\texttt{Compiler}_{\text{centr}}$ can have scalability issues because of exponential growth in intermediate products. $\texttt{Compiler}_{\text{hybr}}$ has no problems with the kind of cycle-based exponential growth discussed above because of how it deals with such cycles in its partitioning algorithm. Generally, however, we can imagine also $\texttt{Compiler}_{\text{hybr}}$ to have this problem for other sources of exponential growth.

Conclusion. For the four k-parametric connector families whose big CAs grow exponentially in k (i.e., EarlyAsyncBarrierMerger, EarlyAsyncMerger, LateAsyncReplicator, and LateAsyncRouter), hybrid compilation has clear advantages over

centralized compilation, as we already expected. For the two k-parametric connector families whose big CAs and intermediate products grow only linearly in k (i.e., LateAsyncMerger and EarlyAsyncReplicator), centralized-approach compilation and hybrid-approach compilation do not make much of a difference; here, run-time performance—investigated in the next section—becomes the key factor in deciding which approach to apply. For Alternator, centralized compilation has clear advantages over hybrid compilation. Finally, for the two k-parametric connector families whose intermediate products grow exponentially in k (i.e., EarlyAsyncOutSequencer and Lock), hybrid compilation seems to have clear advantages over centralized compilation as suggested by Figure 6b.

We find the latter conclusion slightly rash, though. After all, our previous analysis showed that the big CAs—the only CAs that we actually care about—for both EarlyAsyncOutSequencer and Lock grow only linearly in k. If we can develop technology that enables $\texttt{Compiler}_{\text{centr}}$ to avoid temporary exponential growth of intermediate products, $\texttt{Compiler}_{\text{centr}}$ should perform similar to $\texttt{Compiler}_{\text{hybr}}$.

One option is to equip $\texttt{Compiler}_{\text{centr}}$ with a novel static analysis technique to infer, *before* forming the full product, which states will have become unreachable *after* forming the full product. For instance, in the case of $\mathsf{EarlyAsyncOutSequencer}_k$ (or its subconnector $\mathsf{Sequencer}_k$), every state where two or more buffers contain a data item will have become unreachable in the full product but not so yet in the intermediate products. If $\texttt{Compiler}_{\text{centr}}$ can determine such "eventually unreachable states" from the start, it can already remove those states *while* forming the full product to keep the intermediate products as small as possible. This optimization requires significant theoretical work: not only must we formulate the analysis technique itself, but we must also prove that it preserves certain behavioral properties. It seems an interesting form of on-the-fly state space reduction, though, which may have applications also in model checking.

Another option is not really a solution to our problem but a way to avoid it. We observed that the $\mathsf{Sequencer}_k$ subconnector of $\mathsf{EarlyAsyncOutSequencer}_k$ causes its intermediate products to grow exponentially in k. For simplicity, let us therefore focus on this problematic $\mathsf{Sequencer}_k$. The obvious way to construct a connector with the behavior of $\mathsf{Sequencer}_k$ is by putting k buffered channels in a cycle, as we did before. An alternative way to construct such a connector, however, is by connecting a $\mathsf{Sequencer}_{0.5k}$ to another $\mathsf{Sequencer}_{0.5k}$ with a "glue subconnector". The details of this glue subconnector do not matter here: what matters is that in this alternative construction, $\texttt{Compiler}_{\text{centr}}$ can first form the products of the $\mathsf{Sequencer}_{0.5k}$ subconnectors to get two CAs with $0.5k$ states, and then form the products of those CAs and the two-state CA of the glue subconnector. The largest intermediate CA encountered by the compiler during this process has at most $\max(2^{0.5k}, 0.5k \cdot 0.5k \cdot 2)$ states. In contrast, the largest intermediate CA for the obviously constructed $\mathsf{Sequencer}_k$—the one with the cycle—has 2^{k-1} states. This analysis shows that *hierarchically* constructing $\mathsf{Sequencer}_k$ out of $\mathsf{Sequencer}_{l<k}$ subconnectors reduces its centralized-approach compilation complexity compared to its *flat* design. Generally, we should therefore encourage programmers to design connectors as hierarchically as possible.

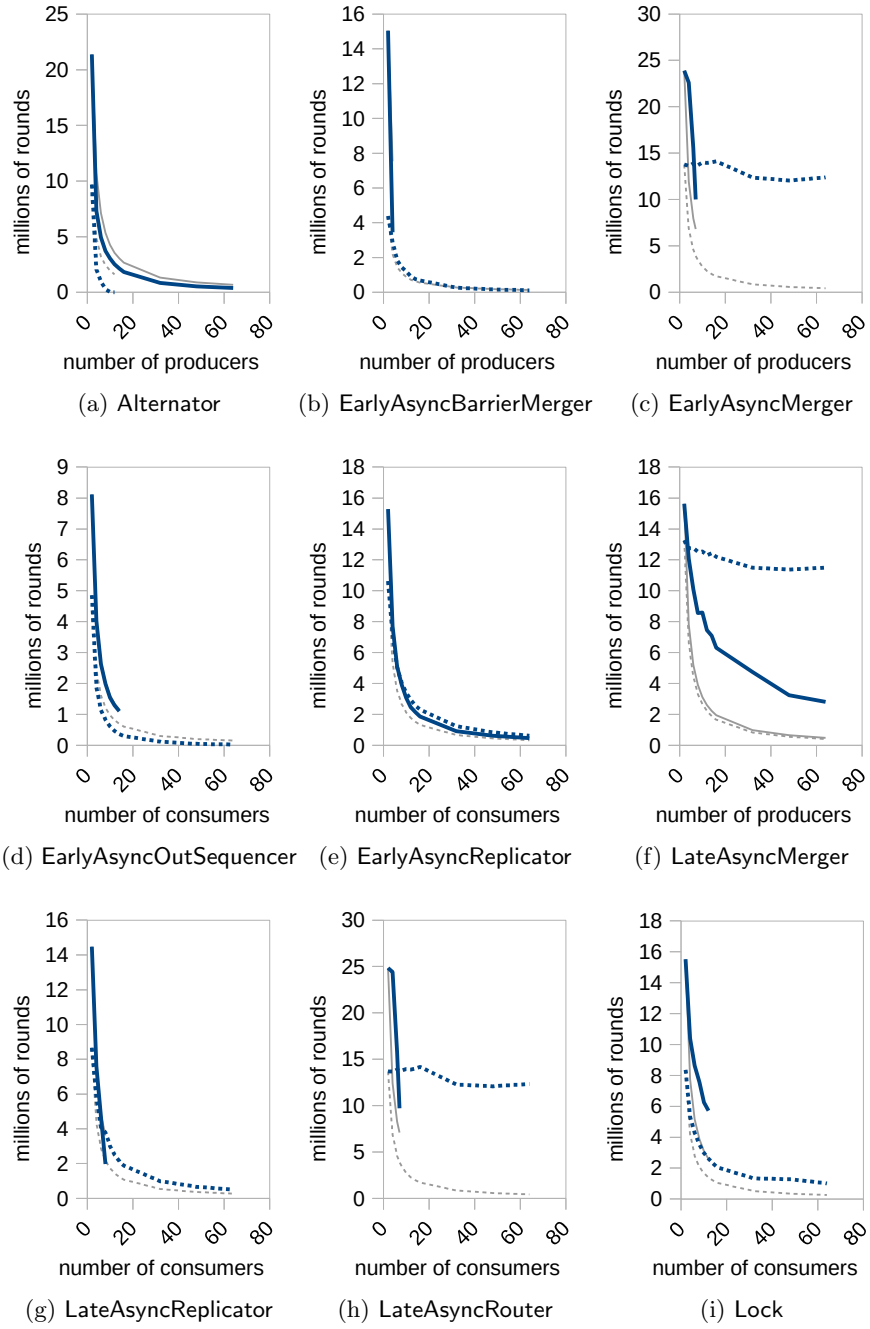
6 Experimental Results: Execution

Measurements. We ran every successfully compiled connector with "empty" computation threads: in every iteration of their infinite loop, a producer/consumer had no work and immediately performed a `put`/`get` on its own public node. As a result, we measured the performance of only the compiler-generated code. Figure 7 shows our measurements, in completed protocol rounds per four minutes. By dividing this number of rounds by 240, one gets the round-throughput, in rounds per second. By further dividing this number by the number of transitions per round, one gets the (transition-)throughput.

Figures 7a, 7b, 7c, and 7f show the performances in the k-producers-single-consumer category. For LateAsyncMerger, EarlyAsyncMerger, and EarlyAsyncBarrierMerger, their centralized implementations outperform their hybrid implementations in cases involving only few producers (up to/including four in the case of LateAsyncMerger and EarlyAsyncBarrierMerger; up to/including six in the case of EarlyAsyncMerger). In cases involving more producers, either the hybrid implementations outperform the centralized implementations, or $\texttt{Compiler}_{\text{centr}}$ failed to compile such that we cannot make a direct comparison. In those latter cases, however, it seems reasonable to assert, by extrapolation, that if compilation had succeeded, these generated centralized implementations would have performed worse than their corresponding hybrid counterparts. For Alternator, in contrast, its centralized implementations always outperform its hybrid implementations.

Figures 7d, 7e, 7g, and 7h show the performances in the single-producer-k-consumers category. The figures for LateAsyncReplicator and LateAsyncRouter are similar to those of LateAsyncMerger, EarlyAsyncMerger, and EarlyAsyncBarrierMerger that we saw before: with only few consumers, their centralized implementations outperform their hybrid implementations, while with more consumers, their hybrid implementations outperform their centralized implementations. For EarlyAsyncReplicator, the performance of its centralized and hybrid implementations is nearly the same. For EarlyAsyncOutSequencer, because $\texttt{Compiler}_{\text{centr}}$ failed to generate code for $k > 14$, the comparison remains inconclusive.

Discussion. For six of the nine connector families, the obtained results look as expected. For those families, we observe that with low values of k (i.e., little parallelism), their centralized implementations outperform their hybrid implementations. In those cases, the increased throughput of hybrid implementations as compared to their centralized counterparts cannot yet compensate for their increased latency. As k increases and more parallelism becomes available, however, hybrid implementations start to outperform centralized implementations. In those cases, increased throughput does seem to compensate for increased latency. This, however, is not the only reason why hybrid implementations outperform centralized implementations for larger values of k. More importantly, we found that the latency of not only hybrid implementations but also centralized implementations increases with k. In fact, the latency of centralized implementations increases much more dramatically. By analyzing the "big" CAs formed by $\texttt{Compiler}_{\text{centr}}$ for the families currently under discussion, we found that their



(a) Alternator (b) EarlyAsyncBarrierMerger (c) EarlyAsyncMerger

(d) EarlyAsyncOutSequencer (e) EarlyAsyncReplicator (f) LateAsyncMerger

(g) LateAsyncReplicator (h) LateAsyncRouter (i) Lock

Fig. 7: Performance, in rounds per four minutes (blue continuous/dotted lines for centralized/hybrid implementations; gray lines for inverse-proportional growth)

exponential growth (cf. Section 5) causes this steep increase in latency: the more transitions a CA has per state, the more time it takes for a thread to select and check any one of them at run-time. (EarlyAsyncReplicator constitutes a special case, where increased throughput and increased latency roughly balance out.)

Contrasting the families discussed in the previous paragraph, the results obtained for Alternator, EarlyAsyncOutSequencer, and Lock are more peculiar. In Section 5, we already briefly explained why $\texttt{Compiler}_{\text{centr}}$ succeeded in generating code for $\text{Alternator}_{k>12}$, while $\texttt{Compiler}_{\text{hybr}}$ failed. This, however, does not yet explain why centralized implementations of Alternator connectors outperform their hybrid implementations also at run-time. The reason becomes clear when we realize that Alternator_k essentially behaves sequentially: in every round, the producers start by synchronously `put`ting their data items (and the consumer synchronously `get`s the first data item), after which the consumer asynchronously `get`s the remaining $k-1$ data items in sequence. The centralized implementation of Alternator_k at run-time sequentially simulates one CA, which consists of k transitions between k states, that represents exactly this sequentiality. Its hybrid implementation, in contrast, at run-time has k parallel protocol threads and, as such, suffers from *overparallelization*: it uses parallelism—and incurs the overhead that parallelism involves—to implement intrinsically sequential behavior. Because also EarlyAsyncOutSequencer and Lock essentially behave sequentially, they suffer from the same problem. For these two families, however, this observation is even more imporant than for Alternator. After all, hybrid-approach compilation fails for $\text{Alternator}_{k>12}$, so for larger k, we must use centralized-approach compilation anyway. For $\text{EarlyAsyncOutSequencer}_{k>14}$ and $\text{Lock}_{k>12}$, in contrast, centralized-approach compilation fails, even though centralized implementations of those connectors are, by extrapolation, likely to perform better than their hybrid counterparts.

Centralized implementations consist of only one protocol thread, which can do only one thing at a time. If many computation threads each perform an I/O operation roughly simultaneously, depending on the connector, this may result in contention (i.e., every computation thread must wait until the protocol thread has time to process its I/O operation). To further study the effect of contention, we repeated our experiments with z-parametric producers/consumers that wait a random amount of time between 0 and $\lceil z$ times the previously measured round-latency$\rceil$ before they perform their `put`/`get`, for $z \in \{1\,,\,10\,,\,100\}$. [12] contains our measurements. The short conclusion is that as z increases, the performance of centralized implementations and hybrid implementations becomes more similar. We doubt whether this can be ascribed to less contention, though. Instead, we consider it more likely that the producers'/consumers' waiting times now dominate our measurements. Although perhaps not too surprising, we nevertheless consider this something that one should be aware of: the more work threads perform, the less important the choice between centralized/hybrid implementation becomes (with respect to run-time performance).

Conclusion. For six of the nine connector families, the obtained results are as we expected: their centralized implementations outperform their hybrid implemen-

tations for smaller values of k, while their hybrid implementations outperform their centralized implementations for larger values of k. As k increases and more parallelism becomes available, the higher throughput of hybrid implementations as compared to their centralized counterparts compensates for their higher latency, while the latency of centralized implementations dramatically increases.

Because Alternator, EarlyAsyncOutSequencer, and Lock essentially behave sequentially, their centralized implementations in fact outperform their hybrid implementations for all k. This is a strong incentive to improve our centralized-approach compilation technology (e.g., the optimization at the end of Section 5).

7 Conclusion

Better understanding the differences between centralized-approach compilation and hybrid-approach compilation is crucial to further advance our compilation technology, one of whose promising applications is programming protocols among threads in multicore applications. Initially, we wanted to investigate under which circumstances parallel protocol code generated by a hybrid-approach compiler, with high throughput but also high latency, outperforms sequential protocol code generated by a centralized-approach compiler, with low latency but also low throughput. Based on our comparison, the answer to this question is this:

- Except for cases with overparallelization, hybrid implementations of connectors with more than a few (e.g., at least ten to twelve) parallel computation threads perform at least as good as their centralized counterparts.

Our comparison taught us much more about centralized/hybrid-approach compilation, though. To summarize our other findings:

- Hybrid-approach compilation may suffer from exponentially sized CAs in cases where centralized-approach compilation works fine.
- Centralized-approach compilation may suffer from exponentially sized intermediate products in cases where hybrid-approach compilation works fine.
- Programmers should prefer hierarchically constructed connectors over flat constructed connectors to reduce compilation complexity.
- Hybrid implementations may overparallelize inherently sequential connectors, which leads to poor run-time performance.
- Centralized implementations may in fact have even higher latency than hybrid implementations.
- The more work threads perform, the less important the choice between centralized/hybrid approach becomes (with respect to run-time performance).

In future work, we want to follow up on these findings by developing new optimization techniques (such as the one sketched in Section 5). In particular, we should identify when hybrid-approach compilation should reduce to centralized-approach compilation and improve our partitioning algorithm accordingly.

Although we heavily used Reo/connector terminology in this paper as a narrative mechanism, we really have been talking about and investigating different

kinds of implementations of a general kind of communicating automaton (i.e., CAs). Because also other languages can have semantics in terms of such automata (e.g., Rebeca [18] and BIP [5]), our findings have applications beyond Reo.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. MSCS 14(3), 329–366 (2004)
2. Arbab, F.: Puff, The Magic Protocol. In: Talcott Festschrift, LNCS, vol. 7000, pp. 169–206. Springer (2011)
3. Arbab, F., Kokash, N., Meng, S.: Towards Using Reo for Compliance-Aware Business Process Modeling. In: ISoLA 2008, CCIS, vol. 17, pp. 108–123. Springer (2008)
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. SCP 61(2), 75–113 (2006)
5. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Components in BIP. In: SEFM 2006. pp. 3–12. IEEE (2006)
6. Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design 36(2), 167–194 (2010)
7. Changizi, B., Kokash, N., Arbab, F.: A Unified Toolset for Business Process Model Formalization. In: Preproceedings of FESCA 2010. pp. 147–156 (2010)
8. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press (2014)
9. Jongmans, S.S., Arbab, F.: Global Consensus through Local Synchronization. In: FOCLASA 2013, pp. 174–188. No. 393 in CCIS, Springer (2013)
10. Jongmans, S.S., Arbab, F.: Modularizing and Specifying Protocols among Threads. In: PLACES 2012, EPTCS, vol. 109, pp. 34–45. CoRR (2013)
11. Jongmans, S.S., Arbab, F.: Toward Sequentializing Overparallelized Protocol Code. In: ICE 2014, EPTCS, vol. 166, pp. 38–44. CoRR (2014)
12. Jongmans, S.S., Arbab, F.: Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code? (Technical Report). Tech. Rep. FM-1503, CWI (2015)
13. Jongmans, S.S., Halle, S., Arbab, F.: Automata-based Optimization of Interaction Protocols for Scalable Multicore Platforms. In: COORDINATION 2014, LNCS, vol. 8459, pp. 65–82. Springer (2014)
14. Jongmans, S.S., Santini, F., Arbab, F.: Partially-Distributed Coordination with Reo. In: PDP 2014. pp. 697–706. IEEE (2014)
15. Meng, S., Arbab, F., Baier, C.: Synthesis of Reo circuits from scenario-based interaction specifications. SCP 76(8), 651–680 (2011)
16. Proença, J., Clarke, D., de Vink, E., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: SAC 2012. pp. 1510–1515. ACM (2012)
17. Proença, J.: Synchronous Coordination of Distributed Components. Ph.D. thesis, Leiden University (2011)
18. Sirjani, M., Jaghoori, M.M., Baier, C., Arbab, F.: Compositional Semantics of an Actor-Based Language Using Constraint Automata. In: COORDINATION 2006, LNCS, vol. 4038, pp. 281–297. Springer (2006)