Automata-theoretic protocol programming

Jongmans, Sung-Shik Theodorus Quirinus

Chapter 8

Improved Compilation IV: Queue-Inference

In Chapter 6, I argued that the performance of compiler-generated code for Mergers, Routers, LateAsyncMergers, and EarlyAsyncMergers should stay (close to) constant in the number of producers/consumers. The experimental results in Chapter 7, however, show that code generated by Lykos does not achieve such scalability, not even with commandification: compiler-generated code for LateAsyncMergers and EarlyAsyncMergers benefits from commandification by 10–20%, but without improving scalability, while compiler-generated code for Mergers and Routers does not seem to benefit from commandification at all.

In this chapter, I present a technique, called *queue-inference*, that improves the scalability of compiler-generated code for protocols with nondeterministic choices, such as Merger, Router, LateAsyncMerger, and EarlyAsyncMerger. In Section 8.1, to develop an intuition for what queue-inference involves, I first explain how to manually apply this optimization technique. Subsequently, I formalize queue-inference in terms of constraint automata, thereby making this technique amenable to automation. In Section 8.2, I present an improved version of Lykos using queue-inference, including new experimental results on performance.

Although the improvement presented in this chapter eventually results in improved compiler-generated code, as in Chapters 5, 6 and 7, I define this improvement at the higher level of constraint automata instead of at the lower level of GPL code. Not only does this facilitate more elegant formal reasoning about correctness (compared to reasoning directly about GPL code), but it also eases the automatic application of this improvement by a FOCAML compiler. Moreover, it makes this improvement independent of GPLs—Java in this thesis—so that the same optimization automatically applies to, for instance, generated C code.

8.1 Theory

*(With Arbab and Halle, I previously published fragments of material in this section in a conference paper [**J**HA14a].)*

Manual Optimization

To understand why the compiler-generated code for Mergers, Routers, LateAsyncMergers, and EarlyAsyncMergers in Chapter 7 scales suboptimally, recall the following run-time behavior of worker and protocol units from Chapter 4: whenever a worker unit performs an I/O operation on the data structure for a port, it informs the protocol unit that shares access to this data structure about this event, after which this protocol unit starts a new round of event-handling. As shown in the simplified event-handler in Figure 4.6, this protocol unit subsequently loops over all transitions out of the current state in search of an enabled one and, once found, fires this enabled transition. For k outgoing transitions, then, this loop requires $\mathcal{O}(k)$ time; clearly, as the number of transitions increases, the average time taken by a protocol unit to complete one round of event-handling also increases. Because the number of transitions in Mergers, Routers, LateAsyncMergers, and EarlyAsyncMergers increases linearly in the number of producers/consumers, exactly the linear complexity of event-handling causes compiler-generated code for those constraint automata to scale suboptimally. This analysis, although formulated here for the Centralized Approach, similarly applies to the Hybrid Approach.

To find a solution for this scalability problem, or at the very least some inspiration, suppose that I provide EarlyAsyncMerger—a representative instance of this problem—as a specification to software engineers and ask them for a manual implementation. I actually did this little exercise with Sean Halle, then a colleague at CWI with many years of experience in parallel programming. Sean made two implementations. In his first implementation, every producer has its own variable for storing data-to-send. To receive, then, the consumer needs to iterate over all these variables in search of a nonempty one (cf., looping over all outgoing transitions in a constraint automaton by event-handlers). Of course, Sean also used locks (with condition variables) to synchronize the producers/consumer and avoid race conditions, but I skip those here. In Sean's first implementation, thus, once the consumer receives a datum, it actually knows which specific producer sent that datum (namely, the producer corresponding to the nonempty variable in which the consumer found that datum). This, however, overimplements my intention: the consumer does not really care about which *specific* producer it receives from so long as it receives from *some* producer. In other words, the producers may remain *indistinguishable* to the consumer. Sean's second implementation exploits this indistinguishability. Instead of using per-producer variables, in this second implementation, the producers offer their data-to-send into a *queue*. To receive, the consumer can simply poll a datum from the queue in $\mathcal{O}(1)$ time (ignoring, for simplicity, the overhead of synchronizing concurrent queue accesses). In this second im-

```
1  public interface Queue {
2    public boolean isEmpty();
3    public void offer(Port port);
4    public Port peek();
5    public Port poll();
6  }

7  public class QueueBasedOutputPortImpl extends OutputPortImpl {
8    public volatile Queue queue;
9
10   @Override
11   public void put(Object datum) throws InterruptedException {
12     buffer = datum;
13     status = IO.PENDING;
14     handler.register();
15     queue.offer(this);
16     resume();
17 } }

18 public class QueueBasedInputPortImpl extends InputPortImpl {
19   public volatile Queue queue;
20
21   @Override
22   public Object get() throws InterruptedException {
23     buffer = null;
24     status = IO.PENDING;
25     handler.register();
26     queue.offer(this);
27     return resume();
28 } }
```

Figure 8.1: Java run-time library extended with queues

plementation, thus, the consumer never knows from which specific producer it receives.

Although not directly a solution, Sean's second implementation formed a key inspiration for the optimization technique presented in this chapter: to improve scalability, compiler-generated code should leverage indistinguishability among workers by using queues. To clarify what exactly this means in the context of compiler-generated code for constraint automata, in the rest of this subsection, I explain—by example—how to perform this optimization technique by manually introducing queues in previous unoptimized compiler-generated code. In subsequent subsections, then, I formalize and automate this transformation at the higher level of constraint automata.

I take the compiler-generated code in Chapter 4 as my starting point. Although Lykos generated that code under the Centralized Approach, code generated under the Hybrid Approach requires similar modifications. First, Figure 8.1 shows my modifications to the run-time library. (This figure constitutes one of the rare exceptions in this thesis, where I qualify ports as "input" or "output" from the perspective of workers.) Implementations of interface `Queue` comprise queue data structures for `Ports`. Whenever a thread `peeks`

```
1  public class QueueBasedProtocol extends Protocol {
2    public Protocol(Port A, Port B, Port C) {
3      super(A, B, C);
4    }
5
6    @Override
7    public void initialize() {
8      ((QueueBasedOutputPortImpl) A).queue = automaton7.queue;
9      ((QueueBasedOutputPortImpl) B).queue = automaton7.queue;
10     super.initialize();
11 } }
```

```
1  class QueueBasedAutomaton7 extends Automaton7 {
2    final Queue queue = new Queue();
3  }
```

Figure 8.2: Classes `QueueBasedProtocol` and `QueueBasedAutomaton7`, manually derived from the automatically generated classes `Protocol` and `Automaton7` in Figures 4.15 and 4.16

such a data structure, it reads its first element without removing it; whenever a thread `poll`s, it not only reads but also removes. Classes `QueueBasedOutputPortImpl` and `QueueBasedInputPortImpl` extend classes `OutputPortImpl` and `InputPortImpl` in Figure 4.13 with `Queue` fields (set elsewhere, discussed shortly) and, notably, with invocations of method `offer` on lines 15 and 26. Thus, whenever a thread performs an I/O operation on a `Port`, after setting all the fields of that `Port`, it `offer`s this `Port` into the `queue`.

Figure 8.2 shows my manual modifications to the automatically generated classes `Protocol` and `Automaton7` in Figures 4.15 and 4.16. In method `initialize` of `QueueBasedProtocol`, the current thread sets the `queue`s in `Port`s `A` and `B`, in addition to everything it already had to do. The value assigned to `A.queue` and `B.queue` comes from an instance of `QueueBasedAutomaton7`, which differs from instances of `Automaton7` only in the added `Queue` field.

Finally, I also manually modified the automatically generated classes `Automaton7Transition1` and `Automaton7Transition2` in Figures 4.18 and 4.19. Figure 8.3 shows these modifications. Class `QueueBasedAutomaton7Transition1` has a new `Queue` field, `initialized` to the `queue` of the `Automaton7` in the `Protocol`. In method `checkSynchronizationConstraint`, instead of using the `Context` of the `Automaton7` as in method `checkSynchronizationConstraint` of the original `Automaton7Transition1`, the current thread checks if the `queue` contains a `Port`. After all, by my previous modifications to the runtime library, if a thread has performed an I/O operation on a `Port`, it must have offered that `Port` into the `queue`. In method `fire`, the current thread actually does the same as in method `fire` of the original `Automaton7Transition1`, except that it first needs to `poll` the `queue` to get the actual `Port` to operate on. To highlight their differences, I grayed out the similar parts in class `QueueBasedAutomaton7Transition2` with respect to class `QueueBasedAutomaton7Transition1`. These classes differ only in their name, which has the following

```
class QueueBasedAutomaton7Transition1 extends Automaton7Transition1 {
  Queue queue;

  public initialize(Protocol protocol) {
    this.queue = protocol.automaton7.queue;
    super.initialize(protocol);
  }

  @Override
  protected boolean checkSynchronizationConstraint() {
    return true && !queue.isEmpty();
  }

  @Override
  protected boolean fire() {
    boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
    if (canFire) {
      Port port = queue.poll();
      port.status = IO.COMPLETED;
      port.semaphore.release();
      target.reach();
    }
    return canFire;
} }

class QueueBasedAutomaton7Transition2 extends Automaton7Transition2 {
  Queue queue;

  public initialize(Protocol protocol) {
    this.queue = protocol.automaton7.queue;
    super.initialize(protocol);
  }

  @Override
  protected boolean checkSynchronizationConstraint() {
    return true && !queue.isEmpty();
  }

  @Override
  protected boolean fire() {
    boolean canFire = checkSynchronizationConstraint() && checkDataConstraint();
    if (canFire) {
      Port port = queue.poll();
      port.status = IO.COMPLETED;
      port.semaphore.release();
      target.reach();
    }
    return canFire;
} }
```

Figure 8.3: Classes `QueueBasedAutomaton7Transition1` and `QueueBasedAutomaton7Transition2`, manually derived from the automatically generated classes `Automaton7Transition1` and `Automaton7Transition2` in Figures 4.18 and 4.19

important consequence.

Reconsider class `HandlerForABC` in Figure 4.22 in the context of my previous modifications. Every instance of this `Handler` represents a comprehensive event-handler for `QueueBasedAutomaton7`, which any thread can call at any time in an attempt to `fire` any `Transition` (as described in more detail in Chapter 4). In method `call` of `HandlerForABC`, the current thread loops over an array of `Transitions` until it has successfully `fired` one. In the context of my previous modifications, this array consists of a `QueueBasedAutomaton7Transition1` and a `QueueBasedAutomaton7Transition2`. However, as highlighted in Figure 4.22, these classes consist of exactly the same code. Therefore, storing an instance of each of those classes in the array effectively amounts to letting the current thread try to `fire` the same `Transition` twice. To avoid this, I may modify `HandlerForABC` by removing the loop and letting the current thread invoke either method `fire` of `QueueBasedAutomaton7Transition1` or method `fire` of `QueueBasedAutomaton7Transition2`—but never both. By doing so, the current thread effectively tries to `fire` one of two `Transitions` at the same time.

With these modifications, essentially, I exploit the indistinguishability of producers to the consumer by making the ports on which those producers perform their I/O operations indistinguishable: without extra information about which producer has access to which specific port, afterward reconstructing which producer offered a port into a queue becomes impossible, notably upon polling that port from the queue—the ports have become indistinguishable. From an automata-theoretic perspective, my modifications to previous compiler-generated code essentially correspond to the notion of *combining* multiple transitions into a single transition (which, at run-time, requires only one check for enabledness in every round of event-handling by using queues). In the next subsections, I present a formalization of this automata-theoretic perspective, including a formalization of port indistinguishability.

Multiconstraint Automata

In this subsection and the next, I call constraint automata as defined in Chapter 2, Definition 19, *uniconstraint automata*. This new piece of terminology allows me to clearly distinguish uniconstraint automata from *multiconstraint automata*, which generalize uniconstraint automata and support combining multiple transitions into single transitions in a behavior-preserving way (which uniconstraint automata do not support). Multiconstraint automata differ from uniconstraint automata in only one aspect: while uniconstraint automata have *synchronization uniconstraints*—sets of ports as in Definition 19—in their transition labels, multiconstraint automata have *synchronization multiconstraints*. Henceforth, I call transitions in uniconstraint automata *unitransitions* and transitions in multiconstraint automata *multitransitions*. See the last paragraph of this subsection for related work.

Definition 60 (multiconstraint automata)**.** *A multiconstraint automaton is a tuple:*

$$(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))$$

where:

- $Q \subseteq \mathbb{Q}$ *(states)*
- $(P^{\text{all}}, P^{\text{in}}, P^{\text{out}}) \in 2^{\mathbb{P}} \times 2^{\mathbb{P}} \times 2^{\mathbb{P}}$ *such that:* *(ports)*

$$P^{\text{in}}, P^{\text{out}} \subseteq P^{\text{all}} \ \textbf{and} \ P^{\text{in}} \cap P^{\text{out}} = \emptyset$$

- $M \subseteq \mathbb{M}$ *(memory cells)*
- $\longrightarrow \subseteq Q \times 2^{2^{2^{P^{\text{all}}}}} \times \mathsf{Good}(P^{\text{all}} \cup {}^{\bullet}M \cup M^{\bullet}) \times Q$ *(multitransitions)*
- $(q^0, \mu^0) \in Q \times (M \to \mathbb{D})$ *(initial configuration)*

$\mathbb{A}\text{UTOM}^{+}$ *denotes the set of all multiconstraint automata, ranged over by* $\boldsymbol{e}$*.*

Every synchronization multiconstraint consists of $\big[$a set G of $\big[$sets E_i of $\big[$sets V_{ij} of ports$\big]\big]\big]$ and represents a nondeterministic choice among synchronization uniconstraints. As a first few examples,

$$G = \{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{D}\}}^{V_{12}}\}\}_{E_1} \quad \text{and} \quad G = \{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{C}\}}^{V_{12}}, \overbrace{\{\mathsf{E}\}}^{V_{13}}\}\}_{E_1}$$

represent the synchronization uniconstraints $\{\mathsf{A}, \mathsf{D}\}$ and $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$ (i.e., no nondeterministic choice). As a second few examples,

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{D}\}}^{V_{12}}\}}_{E_1}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{21}}, \overbrace{\{\mathsf{D}\}}^{V_{22}}\}}_{E_2}, \underbrace{\{\overbrace{\{\mathsf{C}\}}^{V_{31}}, \overbrace{\{\mathsf{D}\}}^{V_{32}}\}}_{E_3}\}$$

and

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{C}\}}^{V_{12}}, \overbrace{\{\mathsf{E}\}}^{V_{13}}\}}_{E_1}, \underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{21}}, \overbrace{\{\mathsf{D}\}}^{V_{22}}, \overbrace{\{\mathsf{E}\}}^{V_{23}}\}}_{E_2}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{31}}, \overbrace{\{\mathsf{C}\}}^{V_{32}}, \overbrace{\{\mathsf{E}\}}^{V_{33}}\}}_{E_3}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{41}}, \overbrace{\{\mathsf{D}\}}^{V_{42}}, \overbrace{\{\mathsf{E}\}}^{V_{43}}\}}_{E_4}\}$$

represent a nondeterministic choice among the three synchronization uniconstraints $\{\mathsf{A}, \mathsf{D}\}$, $\{\mathsf{B}, \mathsf{D}\}$, $\{\mathsf{C}, \mathsf{D}\}$ and a nondeterministic choice among the four synchronization uniconstraints $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$, $\{\mathsf{A}, \mathsf{D}, \mathsf{E}\}$, $\{\mathsf{B}, \mathsf{C}, \mathsf{E}\}$, $\{\mathsf{B}, \mathsf{D}, \mathsf{E}\}$. Thus, in the previous examples—and also in every other synchronization multiconstraint whose every V_{ij} contains exactly one port—G represents a disjunction (with disjuncts represented by E_i) of conjunctions (with conjuncts represented

by V_{ij}). Finally, to explain the more difficult meaning of synchronization multiconstraints with nonsingleton V_{ij}'s, as a third few examples,

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A},\mathsf{B},\mathsf{C}\}}^{V_{11}},\overbrace{\{\mathsf{D}\}}^{V_{12}}\}}_{E_1}\} \quad \text{and} \quad G = \{\underbrace{\{\overbrace{\{\mathsf{A},\mathsf{B}\}}^{V_{11}},\overbrace{\{\mathsf{C},\mathsf{D}\}}^{V_{12}},\overbrace{\{\mathsf{E}\}}^{V_{13}}\}}_{E_1}\}$$

represent *exactly* the same nondeterministic choices among synchronization uniconstraints as the previous two synchronization multiconstraints. Thus, while G represents a disjunction and while every $E_i \in G$ represents a conjunction, every $V_{ij} \in E_i$ represents a *uniqueness quantification*: to compute the nondeterministic choice among synchronization uniconstraints represented by G, for every E_i, collect all synchronization uniconstraints (i.e., sets of ports) constructed by selecting exactly one port from every V_{ij}.

Definition 61 (interpretation of synchronization multiconstraints)**.** $\|\cdot\| : 2^{2^{2^{\mathbb{P}}}} \cup 2^{2^{\mathbb{P}}} \to 2^{\mathbb{P}}$ *denotes the function defined by the following equation:*

$$\|E\| = \left\{ \{p_1,\ldots,p_n\} \,\middle|\, \begin{array}{l} E = \{V_1,\ldots,V_n\} \\ \textbf{and } p_1 \in V_1 \textbf{ and } \cdots \textbf{ and } p_n \in V_n \end{array} \right\}$$
$$\|G\| = \bigcup\{\|E\| \mid E \in G\}$$

Subsequently, I straightforwardly map every multiconstraint automaton to a uniconstraint automaton.

Definition 62 (interpretation of multiconstraint automata)**.** $\|\cdot\| : \mathbb{A}\text{UTOM}^+ \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$\|(Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\longrightarrow,(q^0,\mu^0))\| = (Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\|\longrightarrow\|,(q^0,\mu^0))$$

where $\|\longrightarrow\|$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{G,\phi} q' \textbf{ and } P \in \|G\|}{q \,\|\xrightarrow{P,\phi}\|\, q'} \tag{8.1}$$

Replacing every synchronization uniconstraint $\{p_1,\ldots,p_n\}$ in a uniconstraint automaton $\boldsymbol{a}$ with synchronization multiconstraint $\{\{\{p_1\},\ldots,\{p_n\}\}\}$ straightforwardly yields a multiconstraint automaton $\boldsymbol{e}$ such that $\boldsymbol{a} = \|\boldsymbol{e}\|$. Henceforth, I tacitly apply this behavior-preserving transformation from uniconstraint automata into multiconstraint automata whenever necessary. If a synchronization multiconstraint matches $\{\{p_1\},\ldots,\{p_n\}\}$ (e.g., every E_i in every synchronization multiconstraint resulting from the previous transformation from uniconstraint automata into multiconstraint automata), I call that E_i *simple*.

Definition 63 (simple choices)**.** $\mathsf{Simpl} : 2^{2^{2^{\mathbb{P}}}} \to 2^{2^{2^{\mathbb{P}}}}$ *denotes the function defined by the following equation:*

$$\mathsf{Simpl}(G) = \{\{\{p_1\}, \ldots, \{p_n\}\} \mid \{\{p_1\}, \ldots, \{p_n\}\} \in G\}$$

If a synchronization multiconstraint has only simple E_i's (i.e., if $\mathsf{Simpl}(G) = G$), I call also that synchronization multiconstraint "simple".

To fire a multitransition in a multiconstraint automaton at run-time, its corresponding protocol unit needs to check that multitransition's synchronization multiconstraint G. To perform such a check, this protocol unit must find some $E_i \in G$ such that for every $V_{ij} \in E_i$, *some* port $p \in V_{ij}$ exists such that its corresponding data structure has a pending I/O operation (i.e., the context of the protocol unit, constituted by its pending I/O operations, must satisfy at least one synchronization uniconstraint in the interpretation of G). If so, the protocol unit can effectuate an instance of interaction involving exactly one port out of every V_{ij} (i.e., corresponding to a synchronization uniconstraint in the interpretation of G). Thus, whenever a protocol unit fires a multitransition in a multiconstraint automaton, it effectively fires a unitransition in the corresponding uniconstraint automaton. Importantly, however, even if an external observer knows that a multitransition fired, without any additional information, this observer cannot possibly know to which particular unitransition this firing corresponds. After all, even if the firing multitransition has only one E_i, the observer cannot know which port out of every V_{ij} the protocol unit selected—the ports in every V_{ij} appear indistinguishable to the observer. Moreover, the protocol unit can efficiently select a port out of every V_{ij} by using a queue for that V_{ij}, as in my manually modified compiler-generated code in the previous subsection. This paragraph, then, establishes the connection between $\big[$queues and port indistinguishability in practice$\big]$ and $\big[$in theory$\big]$.

From a propositional logic perspective, a synchronization multiconstraint

$$G = \{E_1, \ldots, E_n\} = \{\{V_{11}, \ldots, V_{1n_1}\}, \ldots, \{V_{n1}, \ldots, V_{nn_n}\}\}$$

for a multiconstraint automaton with ports P^{all} corresponds to the formula

$$\begin{array}{c} \bigoplus V_{11} \cdots \bigoplus V_{1n_1} \prod \{\overline{p} \mid p \in P^{\text{all}} \textbf{ and } p \notin V_{11} \cup \cdots \cup V_{1n_1}\} \\ + \cdots + \\ \bigoplus V_{n1} \cdots \bigoplus V_{nn_n} \prod \{\overline{p} \mid p \in P^{\text{all}} \textbf{ and } p \notin V_{n1} \cup \cdots \cup V_{nn_n}\} \end{array}$$

where $+$ denotes disjunction, juxtaposition/ $\prod$ denotes conjunction, $\overline{\cdot}$ denotes negation, and $\bigoplus$ denotes uniqueness quantification. With Halle and Arbab, I presented (parts of) the material in this chapter entirely from this propositional logic perspective [JHA14a].

When indeed considered as propositional formulas, synchronization multiconstraints may seem similar to propositional *guards* on transitions in *guarded automata* [BCS09, BCS12], which consist of disjunction, conjunction and negation (but no uniqueness quantification). Bonsangue et al. use such guarded automata for modeling protocols, similar to constraint automata, but significantly

different in expressiveness: guarded automata support modeling *context-sensitive* protocols but not data-sensitive protocols, whereas constraint automata support modeling data-sensitive protocols but not context-sensitive protocols (at least not directly; see my work with Krause and Arbab for an indirect encoding [**J**KA11]). Although synchronization multiconstraints and guards formally may seem similar, they have different applications: guards specify the context in which a transition may—or may not—fire in the work of Bonsangue et al., whereas synchronization multiconstraints specify which ports may participate in a transition, irrespective of context. From a propositional logic perspective, synchronization multiconstraints bear similarities also with the various *connector algebras* of Bliudze and Sifakis [BS08, BS10] and Baranov and Bliudze [BB15], whose propositional formulas over ports—with a richer structure than just disjunction/conjunction/negation—induce sets of sets of ports, to compactly represent sets of synchronization uniconstraints ("interactions" in their terminology). The additional expressive power of these connector algebras, relative to synchronization multiconstraints, makes them suitable for compactly representing a wider range of synchronization patterns (their primary use case) but unsuitable for the kind of manipulation shortly introduced in Definition 67 (the primary use case of synchronization multiconstraints). The idea of labeling transitions with atomic propositions plays a key role also in the translation algorithm from LTL formulas to Büchi automata, in the context of model checking, by Giannakopoulou and Lerda [GL02]. Although atomic propositions significantly differ from propositional formulas in expressive power, interestingly, Giannakopoulou and Lerda have a comparable goal (merging "similar" states into single states to shrink automata) as I have (merging "similar" transitions into single transitions to generalize individual ports into queues of ports). The concept of merging states—but not transitions, to my knowledge—as a means of generalizing models has applications also in (stochastic) automaton/grammar inference [BO05, CO94, LPP98, SO93].

Operations on Multiconstraint Automata

Useful as multiconstraint automata in principle may seem for modeling port indistinguishability and queues, the straightforward transformation from uniconstraint automata into multiconstraint automata, just below Definition 62, does not unleash this full potential quite yet. After all, in multiconstraint automata, port indistinguishability and queues manifest as nonsingleton V_{ij}'s in synchronization multiconstraints, but the synchronization multiconstraints in the multiconstraint automata resulting from that transformation contain only singleton V_{ij}'s. Therefore, I introduce two behavior-preserving operations on such multiconstraint automata. Each of these operations changes the structure of its operand, ultimately to form nonsingleton V_{ij}'s, thereby revealing sets of indistinguishable ports in synchronization multiconstraints.

Recall that my manual modifications to compiler-generated code in the previous subsection essentially correspond to the notion of combining multiple transitions into a single transition. My first operation to change the structure

of multiconstraint automata, therefore, indeed combines multiple multitransitions into a single multitransition. Note that the structure of synchronization uniconstraints in uniconstraint automata does not support such an operation.

Definition 64 (combination)**.** $\mathsf{comb} : \mathbb{A}\text{UTOM}^+ \to \mathbb{A}\text{UTOM}^+$ *denotes the function defined by the following equation:*

$$\mathsf{comb}((Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))) = (Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_{\mathsf{comb}}, (q^0, \mu^0))$$

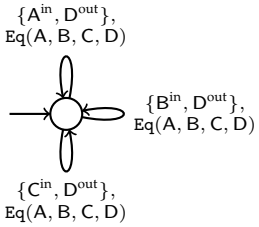
where $\longrightarrow_{\mathsf{comb}}$ *denotes the smallest relation induced by the following rule:*

$$\frac{G^{\text{all}} = \bigcup\{\hat{G} \mid q \xrightarrow{\hat{G},\phi} q'\}}{q \xrightarrow{G^{\text{all}},\phi}_{\mathsf{comb}} q'} \tag{8.2}$$

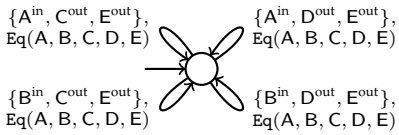
As formalized in Definition 64, to combine multiple multitransitions into a single multitransition, these multitransitions must satisfy three conditions: they must have the same source state, the same target state, and the same data constraint. Especially the latter condition may restrict the extent to which I can combine multitransitions in practice. For instance, in my manual modifications to compiler-generated code in the previous subsection, I effectively combined multitransitions with different—but somehow similar—data constraints (namely A = $x^\bullet$ and B = $x^\bullet$). For now, I simply assume equality of data constraints whenever necessary; I come back to this point in more detail in the next subsection, where I present a basic operation for making "similar" data constraints equal, called *homogenization*.

Figure 8.4 shows uniconstraint automata for two protocols (with homogenized, yet equivalent, data constraints as explained shortly) and their corresponding multiconstraint automata before and after combining their multitransitions; for now, ignore the fourth row with "manipulated synchronization multiconstraints". I introduced the Merger family already in Chapter 3; I introduce the RegulatedMergerRouter family here to demonstrate, shortly, that the optimization technique presented in this chapter supports inference of multiple queues. Figure 8.5 shows a circuit for the same member of the RegulatedMergerRouter$_{2,2}$ subfamily as in Figure 8.4; Figure 8.6 show a FOCAML definition for the entire RegulatedMergerRouter family and a main definition for the same member as in the previous figure. Every member of RegulatedMergerRouter$_{k,l}$ infinitely often atomically [accepts a datum d on one of its k input ports for producers, then offers d both on one of its l output ports for consumers and on another output port for the *regulator*] (where "input" and "output" qualify ports from the protocol perspective). This regulator forms a third party between the group of producers and the group of consumers, regulating (the pace of) flow between those two groups through its `get` operations.

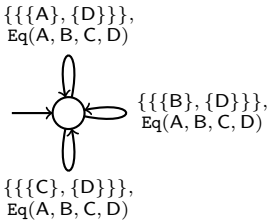
The following theorem states the correctness of Definition 64: combining its multitransitions preserves the interpretation of a multiconstraint automaton.



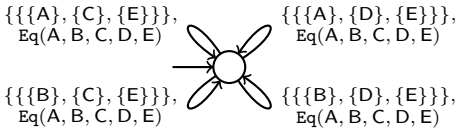
$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: uniconstraint automaton with homogenized data constraints



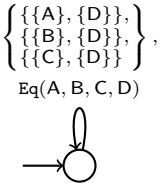
$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: uniconstraint automaton with homogenized data constraints



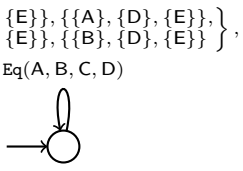
$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: multiconstraint automaton before combining multitransitions



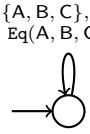
$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: multiconstraint automaton before combining multitransitions



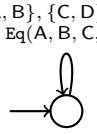
$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: multiconstraint automaton after combining multitransitions



$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: multiconstraint automaton after combining multitransitions



$\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$: multiconstraint automaton after combining multitransitions and after manipulating synchronization multiconstraints



$\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$: multiconstraint automaton after combining multitransitions and after manipulating synchronization multiconstraints

Figure 8.4: $\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$ and $\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A}, \mathsf{B}; \mathsf{C}, \mathsf{D}, \mathsf{E})$

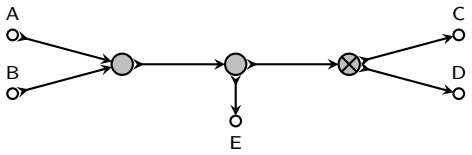


Figure 8.5: Circuit for a member of subfamily RegulatedMergerRouter$_{2,2}$

```
1  RegulatedMergerRouter(in[];out[],out_regulator) = {
2    { prod i:1..#in { Sync(in[i];P1[i]) } }
3      mult Merger(P1[1..#in];P2)
4        mult Replicator2(P2;P3,P6)
5          mult Sync(P3;P4)
6            mult Router(P4,P5[1..#out])
7              mult { prod i:1..#out { Sync(P5[i];out[i]) } }
8          mult Sync(P6;out_regulator)
9  }

10 main = { RegulatedMergerRouter([A,B];[C,D],E) }
```

Figure 8.6: FOCAML definition for family RegulatedMergerRouter and a main definition for a member of RegulatedMergerRouter$_{2,2}$

Theorem 22. $\|e\| = \|\mathsf{comb}(e)\|$

Once multiconstraint automata have combined multitransitions (i.e., after applying comb), for each of their synchronization multiconstraints, I can compute maximal sets of indistinguishable ports. I consider ports indistinguishable if they occur in exactly the same E_i's "modulo occurrences of those ports". To understand the latter phrase, consider the following example:

$$G = \{\underbrace{\{\overbrace{\{\mathsf{A}\}}^{V_{11}}, \overbrace{\{\mathsf{D}\}}^{V_{12}}\}}_{E_1}, \underbrace{\{\overbrace{\{\mathsf{B}\}}^{V_{21}}, \overbrace{\{\mathsf{D}\}}^{V_{22}}\}}_{E_2}, \underbrace{\{\overbrace{\{\mathsf{C}\}}^{V_{31}}, \overbrace{\{\mathsf{D}\}}^{V_{32}}\}}_{E_3}\}$$

Under this G, I can equate E_1, E_2, and E_3 to each other modulo the occurrence of ports A, B, and C, because by removing $\{\mathsf{A}\}$, $\{\mathsf{B}\}$, and $\{\mathsf{C}\}$ from E_1, E_2, and E_3, I get exactly the same *remainder* $\{\{\mathsf{D}\}\}$. Intuitively, this notion of port indistinguishability formalizes the idea that even if an external observer knows that the ports in the remainder of $E_i \in G$ participated in the firing of a G-labeled multitransition (e.g., D), this observer cannot know which port additionally participated in that firing (e.g., A, B, or C)—the ports that additionally may have participated remain indistinguishable to the observer. Let $\mathsf{Port}(G)$ denote the ports that occur in a synchronization multiconstraint G.

Definition 65 (remainders)**.** $\mathsf{Remaind} : 2^{2^{2^{\mathbb{P}}}} \times \mathbb{P} \to 2^{2^{2^{\mathbb{P}}}}$ *denotes the function defined by the following equation:*

$$\mathsf{Remaind}_G(p) = \{E \setminus \{V\} \mid p \in V \in E \in G\}$$

Definition 66 (indistinguishability)**.** $\mathsf{Indist} : 2^{2^{2^{\mathbb{P}}}} \to 2^{2^{\mathbb{P}}}$ *denotes the function defined by the following equation:*

$$\mathsf{Indist}(G) = \left\{ P \,\middle|\, \begin{array}{l} [P = \{p \mid p \in \mathsf{Port}(G) \ \textbf{and} \ \mathsf{Remaind}_G(p) = G'\} \ \textbf{for some} \ G'] \\ \textbf{and} \ P \neq \emptyset \end{array} \right\}$$

Lemma 19. $\mathsf{Indist}(G)$ *denotes a partition of* $\mathsf{Port}(G)$

For instance, let $G = \{\{\{\mathsf{A}\},\{\mathsf{D}\}\},\{\{\mathsf{B}\},\{\mathsf{D}\}\},\{\{\mathsf{C}\},\{\mathsf{D}\}\}\}$ (i.e., the synchronization multiconstraint in $\mathsf{Merger}_3(\mathsf{A},\mathsf{B},\mathsf{C};\mathsf{D})$ in Figure 8.4, third row). Then:

$$\begin{array}{ll} \mathsf{Remaind}_G(\mathsf{A}) & = \{\{\{\mathsf{D}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{B}) & = \{\{\{\mathsf{D}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{C}) & = \{\{\{\mathsf{D}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{D}) & = \{\{\{\mathsf{A}\}\},\{\{\mathsf{B}\}\},\{\{\mathsf{C}\}\}\} \\ \mathsf{Indist}(G) & = \{\{\mathsf{A},\mathsf{B},\mathsf{C}\},\{\mathsf{D}\}\} \end{array}$$

As a more complex example, let

$$G = \{\{\{\mathsf{A}\},\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{A}\},\{\mathsf{D}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{D}\},\{\mathsf{E}\}\}\}$$

(i.e., the synchronization multiconstraint in $\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A},\mathsf{B};\mathsf{C},\mathsf{D},\mathsf{E})$ in Figure 8.4, third row). Then:

$$\begin{array}{ll} \mathsf{Remaind}_G(\mathsf{A}) & = \{\{\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{D}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{B}) & = \{\{\{\mathsf{C}\},\{\mathsf{E}\}\},\{\{\mathsf{D}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{C}) & = \{\{\{\mathsf{A}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{D}) & = \{\{\{\mathsf{A}\},\{\mathsf{E}\}\},\{\{\mathsf{B}\},\{\mathsf{E}\}\}\} \\ \mathsf{Remaind}_G(\mathsf{E}) & = \{\{\{\mathsf{A}\},\{\mathsf{C}\}\},\{\{\mathsf{A}\},\{\mathsf{D}\}\},\{\{\mathsf{B}\},\{\mathsf{C}\}\},\{\{\mathsf{B}\},\{\mathsf{D}\}\}\} \\ \mathsf{Indist}(G) & = \{\{\mathsf{A},\mathsf{B}\},\{\mathsf{C},\mathsf{D}\},\{\mathsf{E}\}\} \end{array}$$

In this more complex example, thus, I get two nonsingleton V_{ij}'s, each of which corresponds to a different queue.

Using the previous formalization of port indistinguishability, I define an operation for manipulating simple synchronization multiconstraints such that afterward, every V_{ij} corresponds to a set of indistinguishable ports. The restriction to simple synchronization multiconstraint does not affect the applicability of this manipulation (in the current context), because a FOCAML compiler manipulates only multiconstraint automata derived from uniconstraint automata using the transformation below Definition 60; such multiconstraint automata have only simple synchronization multiconstraints.

Definition 67 (manipulation of synchronization multiconstraints)**.** $\wr\cdot\int : 2^{2^{2^{\mathbb{P}}}} \to 2^{2^{2^{\mathbb{P}}}}$ *denotes the function defined by the following equation:*

$$\wr G\int = \begin{cases} \left\{ \{P_1,\ldots,P_n\} \;\middle|\; \begin{array}{l} \{p_1,\ldots,p_n\} \in \|G\| \\ \textbf{and}\ \ p_1 \in P_1 \in \mathsf{Indist}(G) \\ \textbf{and}\ \ \cdots \\ \textbf{and}\ \ p_n \in P_n \in \mathsf{Indist}(G) \end{array} \right\} & \textbf{if}\ \ G = \mathsf{Simpl}(G) \\ G & \textbf{otherwise} \end{cases}$$

The following lemma states the correctness of Definition 67: manipulating a synchronization multiconstraint preserves the interpretation of that synchronization multiconstraint.

Lemma 20. $\|G\| = \|\wr G\int\|$

Next, I straightforwardly extend manipulation of synchronization multiconstraints to multiconstraint automata.

Definition 68 (manipulation of multiconstraint automata)**.** $\wr\cdot\int : \mathbb{A}\text{UTOM}^{+} \to \mathbb{A}\text{UTOM}^{+}$ *denotes the function defined by the following equation:*

$$\begin{array}{c} \wr(Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\longrightarrow,(q^0,\mu^0))\int = \\ (Q,(P^{\text{all}},P^{\text{in}},P^{\text{out}}),M,\wr\longrightarrow\int,(q^0,\mu^0)) \end{array}$$

where $\wr\longrightarrow\int$ *denotes the smallest relation induced by the following rule:*

$$\frac{q \xrightarrow{G,\phi} q'}{q \wr\xrightarrow{\wr G\int,\phi}\int q'} \tag{8.3}$$

By manipulating the multiconstraint automata on the third row in Figure 8.4 according to Definition 68, I get the multiconstraint automata on the fourth row in the same figure. These resulting multiconstraint automata have synchronization multiconstraints with nonsingleton V_{ij}'s—as desired—and thereby make the indistinguishability of the ports in those V_{ij}'s structurally explicit. Manipulation of synchronization multiconstraints in multiconstraint automata thus enables a FOCAML compiler to automatically identify indistinguishable ports to infer queues and generate queue-optimized code. Note that for $\mathsf{RegulatedMergerRouter}_{2,2}(\mathsf{A},\mathsf{B};\mathsf{C},\mathsf{D},\mathsf{E})$ in Figure 8.4, doing so yields two queues instead of just one.

The following theorem states the correctness of Definition 68: manipulating its synchronization multiconstraints preserves the interpretation of a multiconstraint automaton.

Theorem 23. $\|\boldsymbol{e}\| = \|\wr\boldsymbol{e}\int\|$

Figure 8.7 shows the compilation approach resulting from queue-inference as just formalized (in which I leave the straightforward step to transform uniconstraint automata $\boldsymbol{d}_1,\ldots,\boldsymbol{d}_m$ into multiconstraint automata implicit).

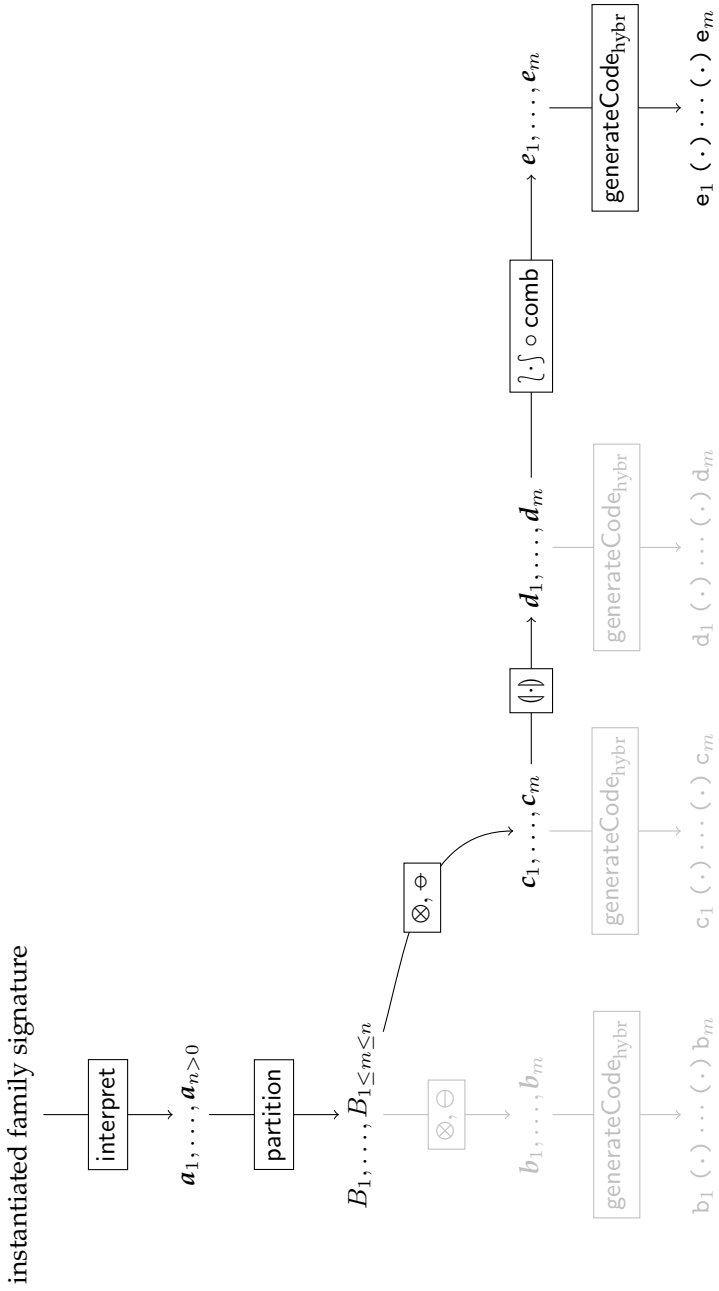


Figure 8.7: Hybrid compilation approach with syntactic subtraction, commandification, and queue-inference

$$\frac{\left[p_1, p_2 \in P \cap \mathrm{Dom}(\sigma) \ \textbf{implies} \ \sigma \models p_1 = p_2\right] \ \textbf{for all} \ p_1, p_2}{\sigma \models \mathtt{Eq}(P)} \tag{8.4}$$

Figure 8.8: Addendum to Definition 69

Homogenization

To combine multiple multitransitions into a single multitransition, by Definition 64 of comb, these multitransitions must have exactly the same data constraint. So far, I simply assumed that this condition holds true, but in fact, all practical cases where queue-inference may improve performance that I know of violate this condition. For instance, Merger_3 in Figure 8.4, first row, actually has A = D, B = D, and C = D as its data constraints. This makes their corresponding multitransitions not amenable for combination. To solve this problem, I must *homogenize* "similar" data constraints before trying to combine their corresponding multitransitions, the result of which Figure 8.4 already exemplifies. In this subsection, I present one approach to such homogenization. Because I apply homogenization only to uniconstraint automata (i.e., before translating them into multiconstraint automata), henceforth, I simply write "constraint automaton", "transition", and "synchronization constraint" (instead of "uniconstraint automaton", "unitransition", and "synchronization uniconstraint").

Essentially, my basic approach to homogenization presented in the rest of this section replaces conjunctions of $p_1 = p_2$ data equalities with a new kind of data atom, $\mathtt{Eq}(P)$. Informally, $\mathtt{Eq}(P)$ means that every port in P has either the same value or no value. Let $\mathbb{DC}^{\mathtt{Eq}}$ denote $\mathbb{DC}$ extended with $\mathtt{Eq}(P)$ data atoms, and let $\mathbb{DC}^{\mathtt{Eq}}_{\exists,\wedge}$ denote the corresponding set of normal data constraints. I define a new entailment relation for $\mathbb{DC}^{\mathtt{Eq}}$.

Definition 69 (entailment with Eq)**.** $\models \subseteq \mathbb{ASSIGNM} \times \mathbb{DC}^{\mathtt{Eq}}$ *denotes the smallest relation induced by the rules in Definition 16 and Figure 8.8.*

Thus, in the rest of this section, I overload $\models$ from the previous Definition 16 with the current Definition 69. Similarly, let $\Rightarrow$ and $\equiv$ denote the implication and the equivalence relation derived from $\models$ in Definition 69 (instead of from $\models$ in Definition 16) in the usual way for first-order logic [Rau10a]. Finally, in the rest of this section, let every data constraint in every constraint automaton come from $\mathbb{DC}^{\mathtt{Eq}}$ instead of from $\mathbb{DC}$. Shortly, I explain the significant semantic difference between $\mathtt{Eq}(P)$ and enumerations of data terms for equating the ports in P in more detail.

To first more informally explain the process of homogenization, suppose that I have a transition (q, P, ϕ, q') in a constraint automaton with ports P^{all} (i.e., $P \subseteq P^{\mathrm{all}}$). Also, suppose that ϕ contains no data variables for memory cells (i.e., $\mathsf{Free}(\phi) \subseteq P$). Finally, suppose that ϕ consist of sufficiently many

data equalities to equate—either directly or transitively—all ports in P to each other (i.e., $P \subseteq \mathsf{Free}(\phi)$). Under these assumptions, clearly, I can safely replace ϕ with $\mathtt{Eq}(P)$. Less than clearly, however, I can also safely replace ϕ with $\mathtt{Eq}(P^{\text{all}})$. After all, whenever transition (q, P, ϕ, q') fires, its corresponding data assignment σ has exactly the ports in P in its domain and no other ports (i.e., $\mathrm{Dom}(\sigma) = P$). Consequently, the condition $p_1, p_2 \in P^{\text{all}} \cap \mathrm{Dom}(\sigma)$ in the premise of Rule 8.4 in Figure 8.8 (instantiated for P^{all}) reduces to $p_1, p_2 \in P^{\text{all}} \cap P$, which in turn reduces to $p_1, p_2 \in P$. In other words, when I evaluate $\mathtt{Eq}(P^{\text{all}})$ in the context of transition (q, P, ϕ, q'), its semantics causes Eq to ignore all ports in $P^{\text{all}} \setminus P$. In those cases, thus, $\mathtt{Eq}(P^{\text{all}})$ reduces to $\mathtt{Eq}(P)$. To homogenize data constraints in a constraint automaton, then, I substitute $\mathtt{Eq}(P^{\text{all}})$ for every data constraint with similar properties as the previous ϕ. For instance, in $\mathsf{Merger}_3(\mathsf{A}, \mathsf{B}, \mathsf{C}; \mathsf{D})$, with transitions $(q, \{\mathsf{A}, \mathsf{D}\}, \mathsf{A} = \mathsf{D}, q)$, $(q, \{\mathsf{B}, \mathsf{D}\}, \mathsf{B} = \mathsf{D}, q)$, and $(q, \{\mathsf{C}, \mathsf{D}\}, \mathsf{C} = \mathsf{D}, q)$, every data constraint has the required properties, and therefore, homogenization replaces each of them with $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$. Subsequently, I can translate the resulting homogenized constraint automaton into a multiconstraint automaton as explained in the previous subsection and exemplified in Figure 8.4.

To formalize homogenization, I restrict myself to normalized constraint automata (without loss of generality). To determine, then, whether a normal data constraint φ has sufficiently many data equalities to equate all ports in a synchronization constraint P to each other, let function EqTerm construct the edge relation of a graph, whose every vertex corresponds to a data term in φ, and whose every edge corresponds to a data equality in φ.

Definition 70 (equated data terms)**.** $\mathsf{EqTerm} : \mathbb{DC}^{\mathtt{Eq}}_{\exists,\wedge} \to 2^{\mathbb{TERM} \times \mathbb{TERM}}$ *denotes the function defined by the following equations:*

$$
\begin{aligned}
\mathsf{EqTerm}(a) &= \begin{cases} \{(t_1, t_2), (t_2, t_1)\} & \textbf{if } a = t_1 = t_2 \\ \emptyset & \textbf{otherwise} \end{cases} \\
\mathsf{EqTerm}(\neg a) &= \emptyset \\
\mathsf{EqTerm}(\exists x.\varphi) &= \mathsf{EqTerm}(\varphi) \\
\mathsf{EqTerm}(\ell_1, \ldots, \ell_k) &= \mathsf{EqTerm}(\ell_1) \cup \cdots \cup \mathsf{EqTerm}(\ell_k)
\end{aligned}
$$

For a normal data constraint φ to have sufficiently many data equalities to equate all ports in P to each other, then, a path from every $p_1 \in P$ to every other $p_2 \in P$ must exist in $\mathsf{EqTerm}(\varphi)$. Equivalently, the transitive closure of $\mathsf{EqTerm}(\varphi)$, denoted by $\mathsf{EqTerm}(\varphi)^+$, must contain the pair (p_1, p_2) for all $p_1, p_2 \in P$. Furthermore, as previously explained, φ must not contain memory cell variables, and for simplicity, I also forbid the occurrence of data functions and data relations. Let $\mathsf{Term}(\varphi)$ denote the set of data terms that occur in φ, and let $\mathsf{OnlyEq}(\varphi)$ hold true iff φ contains only data equalities (and no data relations).

Definition 71 (total data equalities over sets of ports)**.** $\mathsf{Eq} : 2^{\mathbb{P}} \to 2^{\mathbb{DC}^{\mathsf{Eq}}_{\exists,\wedge}}$ *denotes the function defined by the following equation:*

$$\mathsf{Eq}(P) = \left\{ \varphi \;\middle|\; \begin{array}{l} \{(p_1, p_2) \mid p_1, p_2 \in P\} \subseteq \mathsf{EqTerm}(\varphi)^+ \\ \textbf{and } \mathsf{Term}(\varphi) = \mathsf{Free}(\varphi) = P \\ \textbf{and } \mathsf{OnlyEq}(\varphi) \end{array} \right\}$$

I define homogenization in terms of Eq and $\mathtt{Eq}$.

Definition 72 (homogenization)**.** $\langle\cdot\rangle : \mathbb{A}\mathrm{UTOM} \to \mathbb{A}\mathrm{UTOM}$ *denotes the function defined by the following equation:*

$$\begin{array}{c} \langle (Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \longrightarrow, (q^0, \mu^0)) \rangle = \\ (Q, (P^{\mathrm{all}}, P^{\mathrm{in}}, P^{\mathrm{out}}), M, \langle\longrightarrow\rangle, (q^0, \mu^0)) \end{array}$$

where $\langle\longrightarrow\rangle$ *denotes the smallest relation induced by the following rules:*

$$\frac{q \xrightarrow{P,\phi} q' \;\textbf{ and }\; \phi \in \mathsf{Eq}(P)}{q \;\langle\xrightarrow{P,\mathsf{Eq}(P^{\mathrm{all}})}\rangle\; q'} \quad (8.5) \qquad \frac{q \xrightarrow{P,\phi} q' \;\textbf{ and }\; \phi \notin \mathsf{Eq}(P)}{q \;\langle\xrightarrow{P,\phi}\rangle\; q'} \quad (8.6)$$

Problematically, if I substitute $\mathtt{Eq}(P^{\mathrm{all}})$ for ϕ in a transition (q, P, ϕ, q'), I do not preserve the semantics of ϕ in the sense of $\equiv$ (i.e., the equivalence relation on data constraints derived from $\models$): although satisfaction of $\mathtt{Eq}(P^{\mathrm{all}})$ implies satisfaction of ϕ (i.e., $\mathtt{Eq}(P^{\mathrm{all}}) \Rightarrow \phi$), satisfaction of ϕ does not imply satisfaction of $\mathtt{Eq}(P^{\mathrm{all}})$ (i.e., $\phi \not\Rightarrow \mathtt{Eq}(P^{\mathrm{all}})$). For instance, whereas data assignment $\{\mathsf{A} \mapsto 0, \mathsf{B} \mapsto 0, \mathsf{C} \mapsto 0, \mathsf{D} \mapsto 0\}$ satisfies both $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$ and $\mathsf{A} = \mathsf{D}$, data assignment $\{\mathsf{A} \mapsto 0, \mathsf{B} \mapsto 1, \mathsf{D} \mapsto 0\}$ satisfies $\mathsf{A} = \mathsf{D}$ but not $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$. To prove the correctness of Definition 72, therefore, I need to develop some more technical machinery, based on a *tighter* notion of implication/equivalence for data constraints and their corresponding notion of behavioral preorder/congruence for constraint automata.

Definition 73 (tight implication)**.** $\Rightarrow^{\mathsf{t}} \subseteq 2^{\mathbb{P}} \times \mathbb{DC}^{\mathsf{Eq}} \times \mathbb{DC}^{\mathsf{Eq}}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\begin{bmatrix} P = \mathbb{P} \cap \mathrm{Dom}(\sigma) \\ \textbf{and } \sigma \models \phi_1 \end{bmatrix} \textbf{ implies } \sigma \models \phi_2\right] \textbf{ for all } \sigma}{\phi_1 \Rightarrow^{\mathsf{t}}_P \phi_2} \quad (8.7)$$

Definition 74 (tight equivalence)**.** $\equiv^{\mathsf{t}} \subseteq 2^{\mathbb{P}} \times \mathbb{DC}^{\mathsf{Eq}} \times \mathbb{DC}^{\mathsf{Eq}}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\phi_1 \Rightarrow^{\mathsf{t}}_P \phi_2 \;\textbf{ and }\; \phi_2 \Rightarrow^{\mathsf{t}}_P \phi_1}{\phi_1 \equiv^{\mathsf{t}}_P \phi_2} \quad (8.8)$$

Revisiting my previous example, $\mathtt{Eq}(\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\})$ tightly implies $\mathsf{A} = \mathsf{D}$ under any set of ports (because implication subsumes tight implication), while $\mathsf{A} = \mathsf{D}$

tightly implies $\mathsf{Eq}(\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D}\})$ under $\{\mathsf{A},\mathsf{D}\}$. Thus, $\mathsf{Eq}(\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D}\})$ and $\mathsf{A} = \mathsf{D}$ have tightly equivalent semantics under $\{\mathsf{A},\mathsf{D}\}$. The following lemma generalizes this example: it states that whenever a data constraint contains sufficiently many data equalities to equate all ports in P to each other, this data constraint and data constraint $\mathsf{Eq}(P^{\text{all}})$ have tightly equivalent semantics under P.

Lemma 21. $\left[\phi \in \mathsf{Eq}(P) \text{ and } P \subseteq P^{\text{all}}\right]$ **implies** $\phi \equiv^{\text{t}}_{P} \mathsf{Eq}(P^{\text{all}})$

Using tight implication, I can formulate a tighter version of the behavioral preorder and the behavioral congruence in Definitions 24 and 25. Under these tighter versions—and as the only difference with respect to Definitions 24 and 25—for one constraint automaton to "tightly simulate" a transition of another constraint automaton, the data constraint of the simulated transition must tightly imply the data constraint on the simulating transition under the synchronization constraint of the simulated transition.

Definition 75 (tight behavioral preorder). $\preceq^{\text{t}} \subseteq 2^{\mathbb{Q}\times\mathbb{Q}} \times \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\begin{array}{l} R \subseteq Q_1 \times Q_2 \text{ and } q_1^0 \; R \; q_2^0 \\ \text{and } \left[\begin{array}{c} \left[\, \left[\begin{array}{l} q_1 \xrightarrow{P,\phi_1}_1 q_1' \\ \text{and } q_1 \; R \; q_2 \end{array}\right] \text{ implies } \phi_1 \Rightarrow^{\text{t}}_{P} \bigvee \left\{ \phi_2 \;\middle|\; \begin{array}{l} q_2 \xrightarrow{P,\phi_2}_2 q_2' \\ \text{and } q_1' \; R \; q_2' \end{array} \right\} \right] \\ \text{for all } q_1, q_1', q_2, P, \phi_1 \end{array}\right] \end{array}}{\begin{array}{c} (Q_1, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_1, (q_1^0, \mu^0)) \\ \preceq^{\text{t}}_{R} (Q_2, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow_2, (q_2^0, \mu^0)) \end{array}} \tag{8.9}$$

Definition 76 (tight behavioral congruence). $\simeq^{\text{t}} \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\boldsymbol{a}_1 \preceq^{\text{t}}_{R} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_2 \preceq^{\text{t}}_{R^{-1}} \boldsymbol{a}_1\right] \text{ for some } R}{\boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2} \tag{8.10}$$

Behavioral congruence subsumes tight behavioral congruence. Moreover, the following theorems state that tight behavioral congruence implies behavioral equivalence and that tight behavioral congruence constitutes a congruence under the multiplications previously defined in this thesis.

Theorem 24. $\boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2$ **implies** $\boldsymbol{a}_1 \approx \boldsymbol{a}_2$

Theorem 25. $\left[\begin{array}{l} \boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3, \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \text{and } \boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_4 \end{array}\right]$ **implies** $\boldsymbol{a}_1 \circledast_* \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_2 \circledast_* \boldsymbol{a}_4$

Theorem 26. $\left[\begin{array}{l} \boldsymbol{a}_1 \otimes \boldsymbol{a}_3, \boldsymbol{a}_2 \otimes \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \text{and } \boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_4 \end{array}\right]$ **implies** $\boldsymbol{a}_1 \otimes \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_2 \otimes \boldsymbol{a}_4$

Theorem 27. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_3, \boldsymbol{a}_2 \odot \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq^{\text{t}} \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \odot \boldsymbol{a}_3 \simeq^{\text{t}} \boldsymbol{a}_2 \odot \boldsymbol{a}_4$

Using all this tight machinery, I can finally prove the correctness of Definition 72 of $\langle\cdot\rangle$: homogenizing a constraint automaton preserves its behavior.

Theorem 28. $\boldsymbol{a} \simeq^{\text{t}} \langle \boldsymbol{a} \rangle$

The use of tight behavioral congruence instead of behavioral congruence in the previous theorem remains inconsequential, because only behavioral equivalence truly matters in the end (i.e., accepted interaction languages), which Theorem 24 guarantees for tightly behaviorally congruent constraint automata. Furthermore, because $\simeq^{\text{t}}$ denotes a congruence under $\odot$ by Theorem 27, I can safely replace every constraint automaton with its homogenized version in an l-multiplication expression (i.e., $\boldsymbol{d}_1 \odot \cdots \odot \boldsymbol{d}_m \simeq^{\text{t}} \langle \boldsymbol{d}_1 \rangle \odot \cdots \odot \langle \boldsymbol{d}_m \rangle$ in Figure 8.7).

Homogenization as presented above has limitations, notably as it does not support data functions and data relations. The development of a more general approach to homogenization of data constraints seems an interesting—and challenging—piece of future work. One possible approach to such generalized homogenization consists of the introduction of a "metaquantifier" whose "metavariables" range over data variables (cf. $\exists$, whose data variables range over data). Consider, for instance, transitions $(q, \{\mathsf{A}, \mathsf{C}\}, \texttt{incr}(\mathsf{A}) = \mathsf{C}, q')$ and $(q, \{\mathsf{B}, \mathsf{C}\}, \texttt{incr}(\mathsf{B}) = \mathsf{C}, q')$. Homogenization as defined in Definition 72 leaves these transitions untouched, thereby inhibiting their combination later on. The metaquantifier-based homogenization that I imagine, in contrast, replaces the data constraints in these transitions with $\mathcal{M}[\xi : \{\mathsf{A}, \mathsf{B}\}].\texttt{incr}(\xi) = \mathsf{C}$, where $\mathcal{M}$ denotes the metaquantifier, where ξ denotes a metavariable that ranges over $\{\mathsf{A}, \mathsf{B}\}$, and where $\texttt{incr}(\xi) = \mathsf{C}$ denotes a *template* (i.e., a data constraint with metavariables at places where normally data variables occur). Metaquantification, then, has the following semantics:

$$\sigma \models \mathcal{M}[\xi : X].\theta \textbf{ iff } \left[\left[x \in X \textbf{ and } \sigma \models \theta\{x/\xi\}\right] \textbf{ for some } x\right]$$

where θ denotes a template and where $\theta\{x/\xi\}$ denotes the syntactic substitution of data variable x for metavariable ξ. The main challenge with this approach lies in the inference of metaquantifications, for which I have not found an elegant solution yet.

8.2 Practice

(I have not yet submitted material in this section for publication.)

Compiler

I extended Lykos with the ability to apply queue-inference as in Figure 8.7, controllable through flag `INFER_QUEUES`. When raised, after having computed

product automata (and after subtraction of internal ports, either semantically or syntactically depending on the status of the SUBTRACT_SYNTACTICALLY flag), Lykos homogenizes data constraints, translates constraint automata into multiconstraint automata, merges transitions with homogenized data constraints, and modifies the synchronization multiconstraints on the resulting multitransitions to infer indistinguishable ports, all as described in Section 8.1.

To avoid unnecessary queue overhead, Lykos does not inject queues for singleton sets of indistinguishable ports or for sets with ports of mixed polarity. Also, Lykos injects queues only for ports that occupy the same set of indistinguishable ports in every multitransition. For instance, if one multitransition has a synchronization multiconstraint with $V_{ij} = \{\mathsf{A}, \mathsf{B}\}$, while another multitransition has a synchronization multiconstraint with $V_{ij} = \{\mathsf{A}, \mathsf{C}\}$, Lykos injects a queue neither for A and B nor for A and C. Otherwise, whenever a thread performs an I/O operation on the data structure for A, it has to offer this data structure into both queues, which subsequently requires additional machinery to keep the queues consistent (e.g., whenever another thread polls the data structure for A from one of the queues, it must remove this data structure also from the other queue). Not injecting queues in these cases avoids this kind of overhead. More advanced schemes for injecting queues may exist, though, that alleviate or completely eliminate such overhead. One possible such scheme, for instance, comprises the organization of queues in *conjunctions*, or more generally, in BDD-like structures, by combining unitransitions into (more advanced than in this chapter) multitransitions to further reduce the cost of checking their enabledness. Investigating such schemes seems an interesting opportunity for future work.

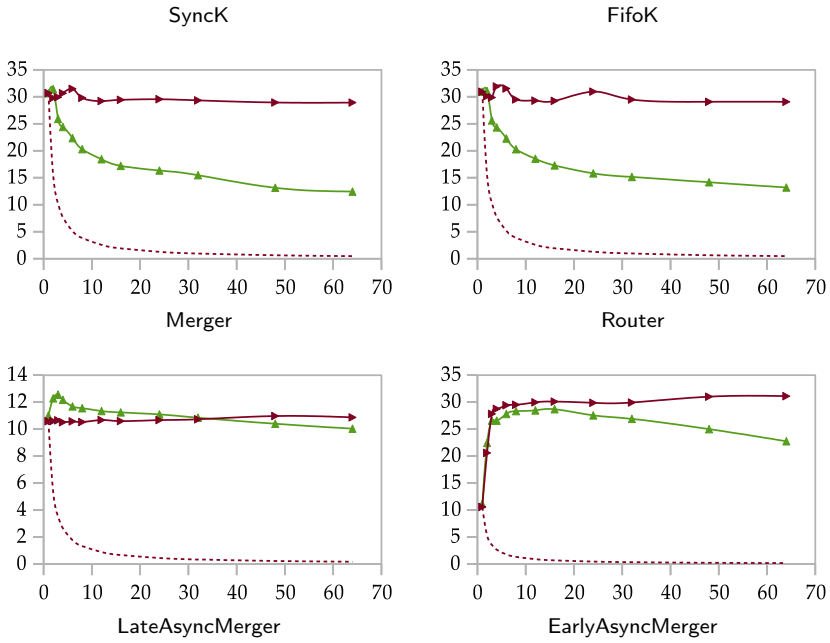
In contrast to Lykos's internals, the run-time library requires no significant modifications, beside the addition of a queue data structure. I implemented this data structure as a concurrent circular buffer, using a semaphore to control concurrent accesses. At run-time, every instance of this data structure has a capacity n, where n equals the size of the set of indistinguishable ports to which this instance corresponds. Code generated with queue-inference differs somewhat from code generated without queue-inference, primarily to account for queues, as already explained in Section 8.1.

Experiments I: Protocols

I repeated the same experiments as in Chapter 7 (and Chapters 4, 5, and 6), generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the INFER_QUEUES-flag raised, but otherwise under the same conditions as in Chapter 7. Figure 8.9 shows the per-family experimental results, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. The purple lines represent the new results; the green lines represent the results from Chapter 6. For SyncK, FifoK, OddFibonacci, and Chess, Lykos generated exactly the same code as in Chapter 7—members of these families have no indistinguishable ports to optimize.

—same as in Figure 7.10—

—same as in Figure 7.10—



—same as in Figure 7.10—

—same as in Figure 7.10—

OddFibonacci

Chess

Figure 8.9: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

In contrast, for all members of Merger, Router, LateAsyncMerger, and EarlyAsyncMerger, queue-inference has the desirable effect that the performance or their generated code becomes constant in the number of producers/consumers.

Figure 8.10 shows per-family speedup charts corresponding to the measurements in Figure 8.9; the dotted lines represent equal performance. Only for LateAsyncMergers and smaller values of k, code generated without queue-inference outperforms code generated with queue-inference. In these cases, the overhead of managing queues outweighs their benefits. In all other cases, code generated with queue-inference outperforms code generated without queue-inference, by an increasing margin in k, up to a speedup of 133% for Mergers, of 120% for Routers, of 8% for LateAsyncMergers, and of 37% for EarlyAsyncMergers.

Experiments II: Programs

I repeated the same experiments as in Chapter 7 (and Chapters 4, 5 and 6), generating code for the NPB benchmarks with the `INFER_QUEUES`-flag raised, but otherwise under the same conditions as in Chapter 7.

Figures 8.11–8.18 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. In the absence of dotted purple lines, the dotted green lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks without queue-inference; the solid purple/green lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represent the Java versions by Frumkin et al. For the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions, Lykos generated exactly the same code as in Chapter 7—members of these families have no indistinguishable ports to optimize—and therefore, I have no new results for these versions (i.e., Figures 8.11–8.18 have no dotted purple lines).

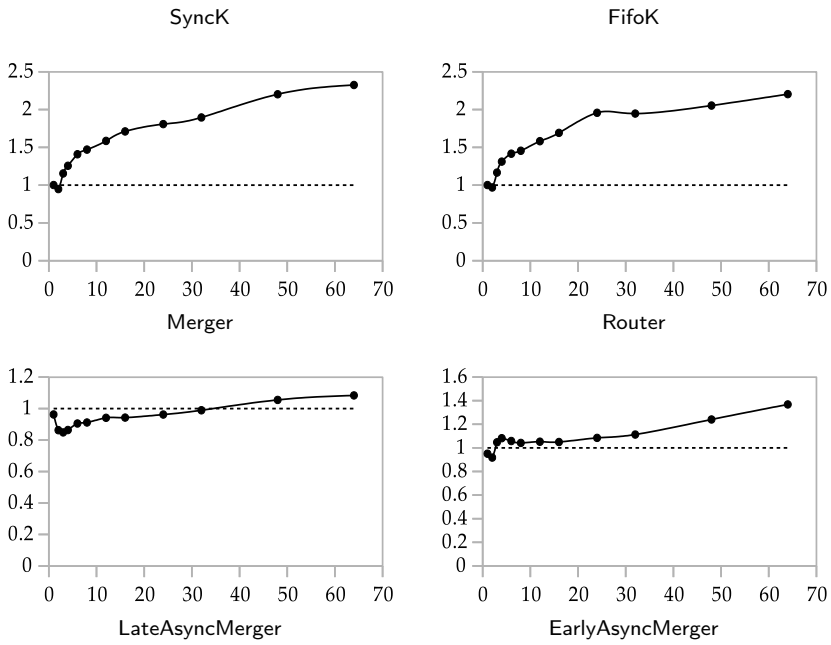
I make the following main observations about these experimental results:

- Overall, the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks with queue-inference outperform the FOCAML-to-Java-compiled versions without queue-inference (purple lines versus green lines).

- Overall, the Java versions of the NPB kernel benchmarks by Frumkin et al. and the FOCAML-to-Java-compiled versions with queue-inference have roughly similar performance: in some cases the former outperform the latter, while in other cases, the latter outperform the former (e.g., in NPB-FT for smaller values of k; in NPB-MG and NPB-CG for larger values of k).

- In cases such as for NPB-MG and NPB-CG, whose FOCAML-to-Java-compiled versions outperform their Java versions for larger values of k, those

—same as in Figure 7.11— —same as in Figure 7.11—



—same as in Figure 7.11— —same as in Figure 7.11—

OddFibonacci Chess

Figure 8.10: Speedup (relative to compiler-generated code in Chapter 6) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

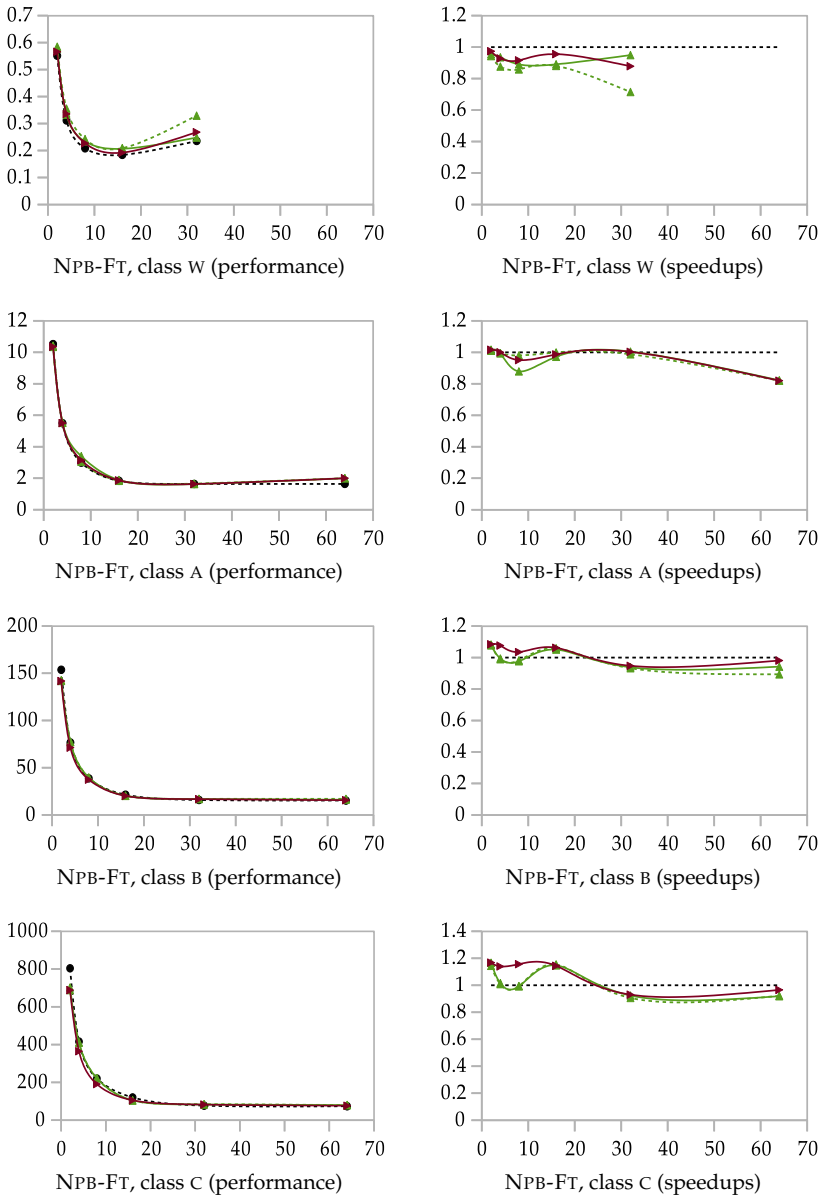


Figure 8.11: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

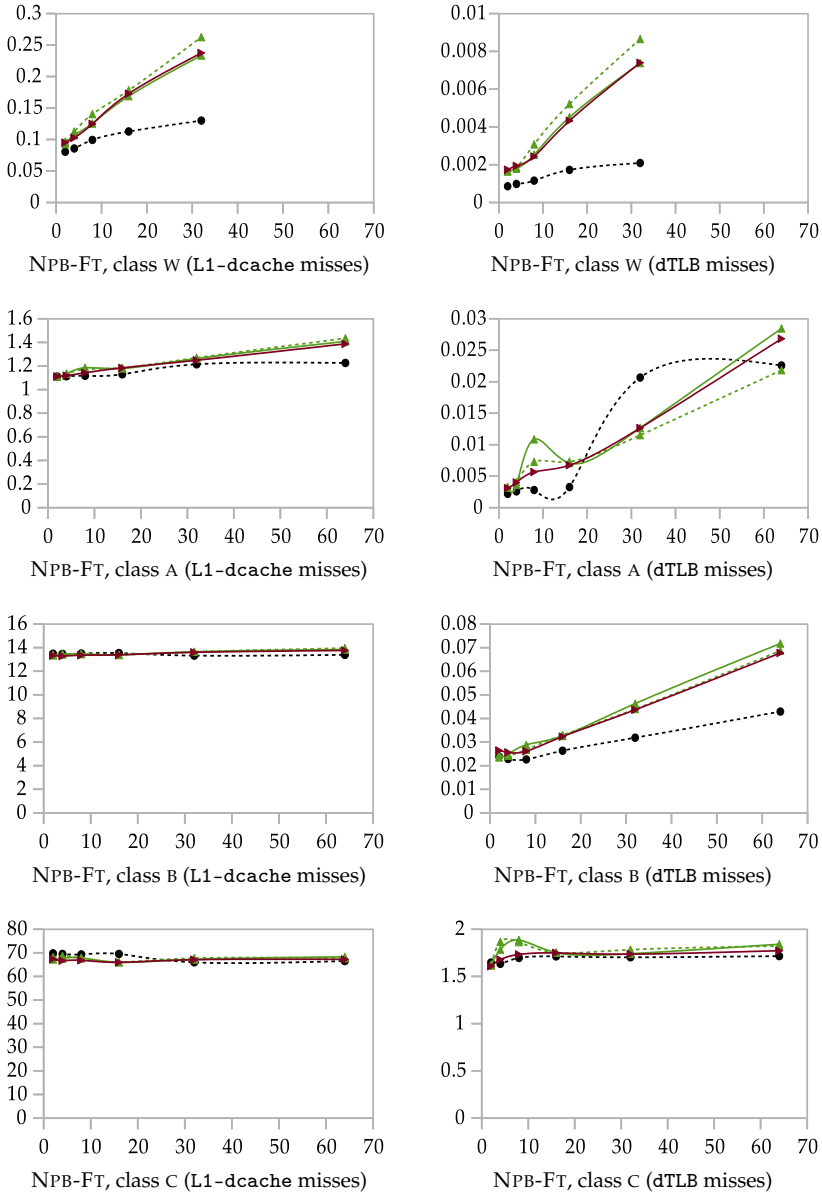


Figure 8.12: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

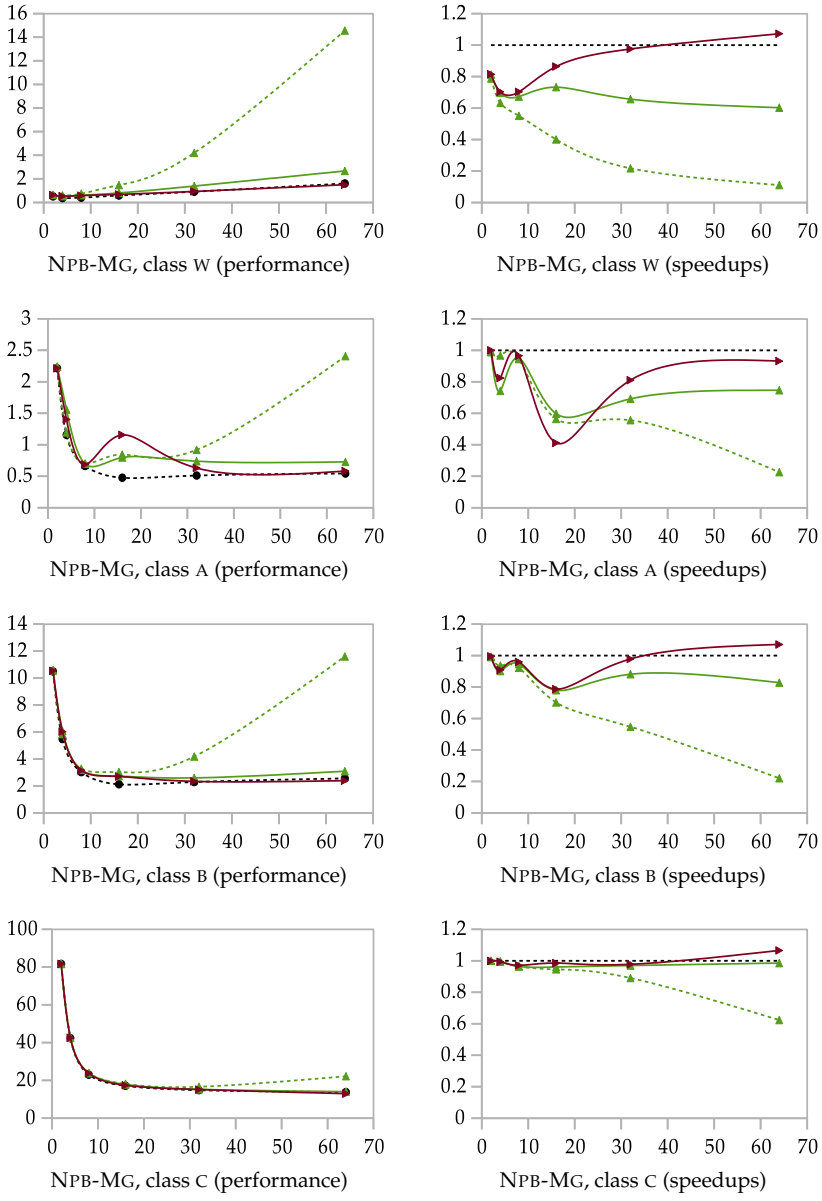


Figure 8.13: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

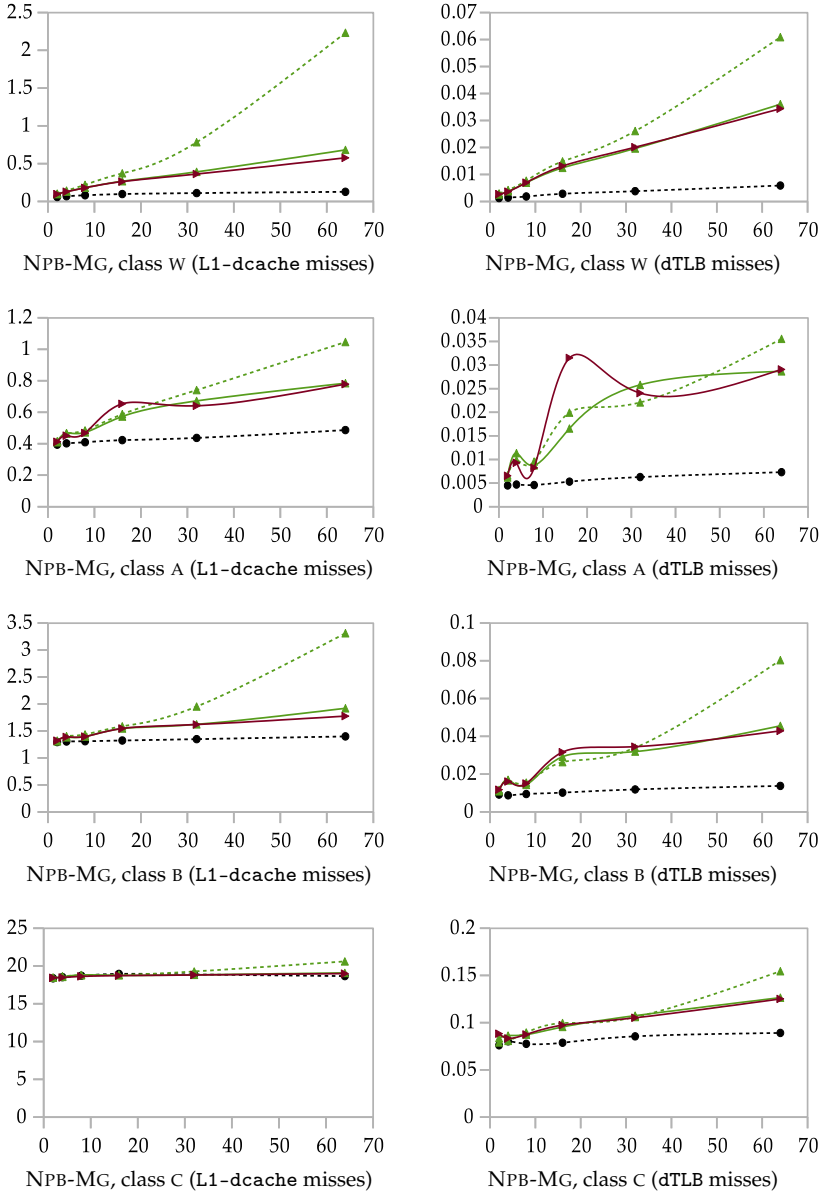


Figure 8.14: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

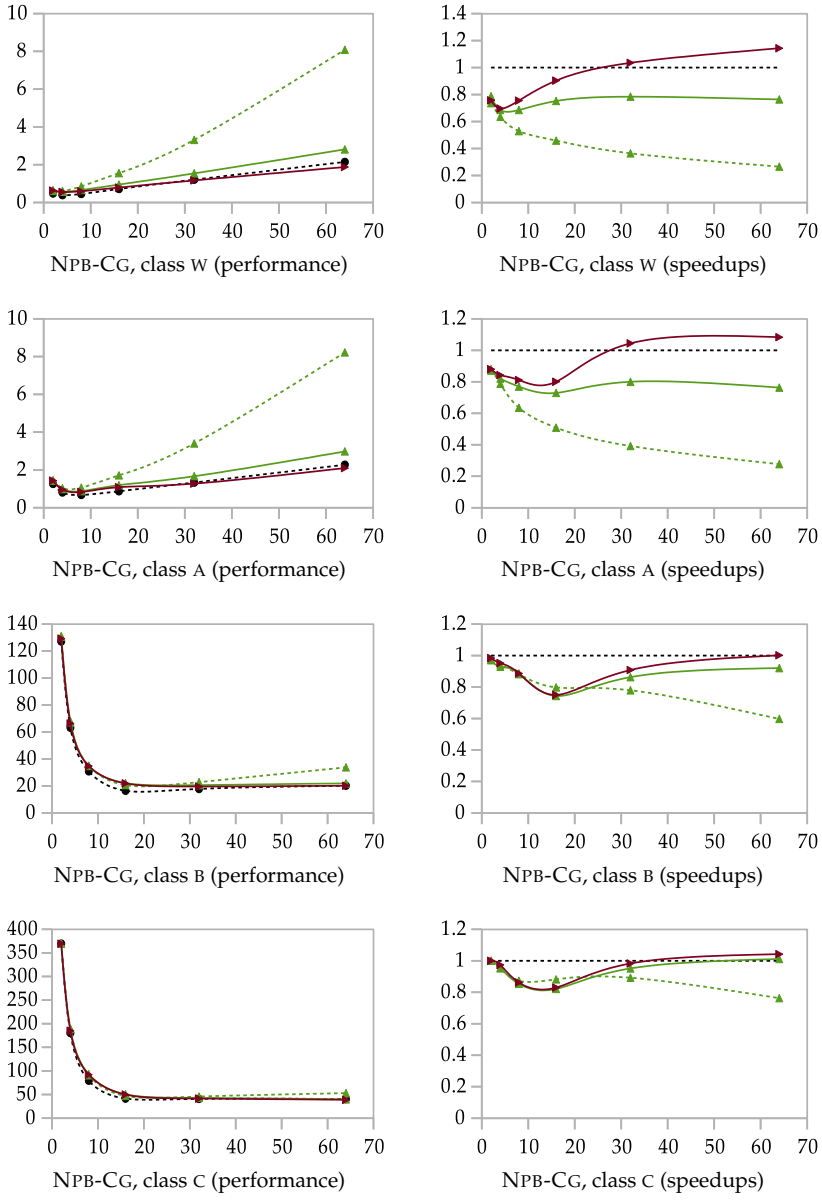


Figure 8.15: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

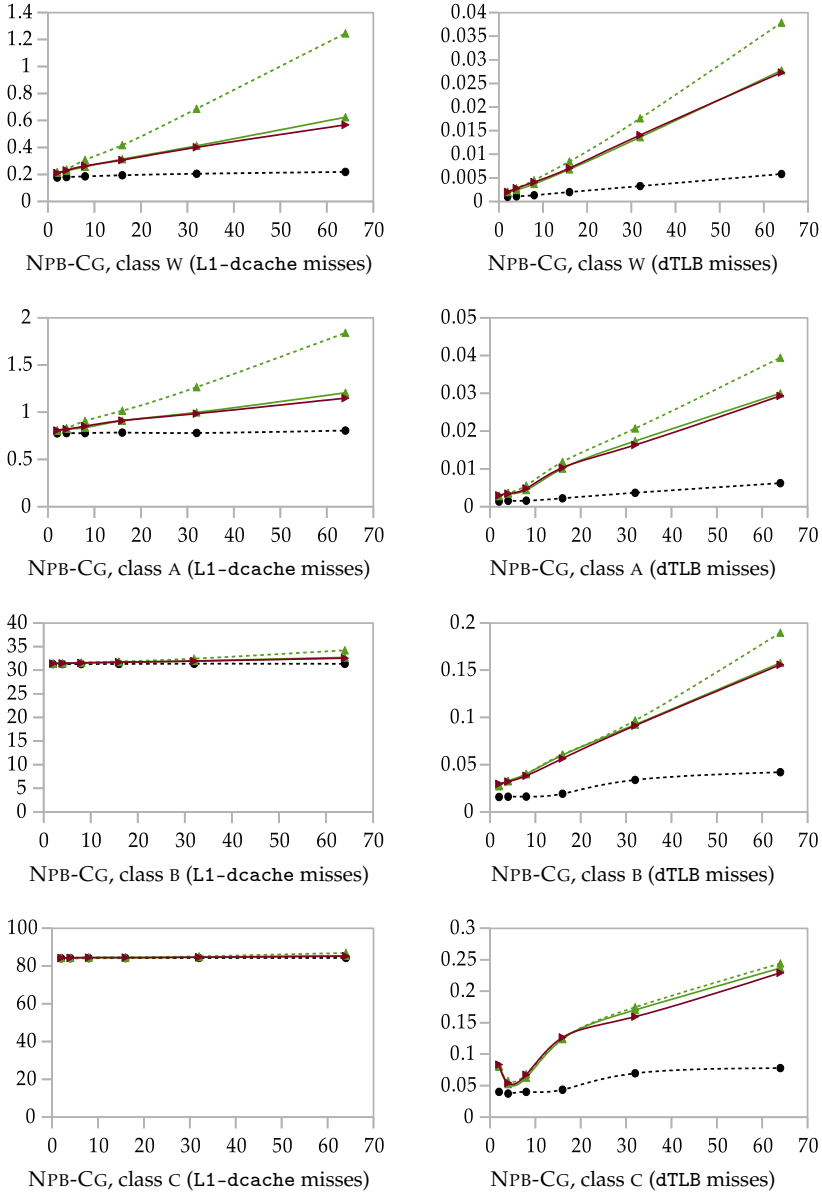


Figure 8.16: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

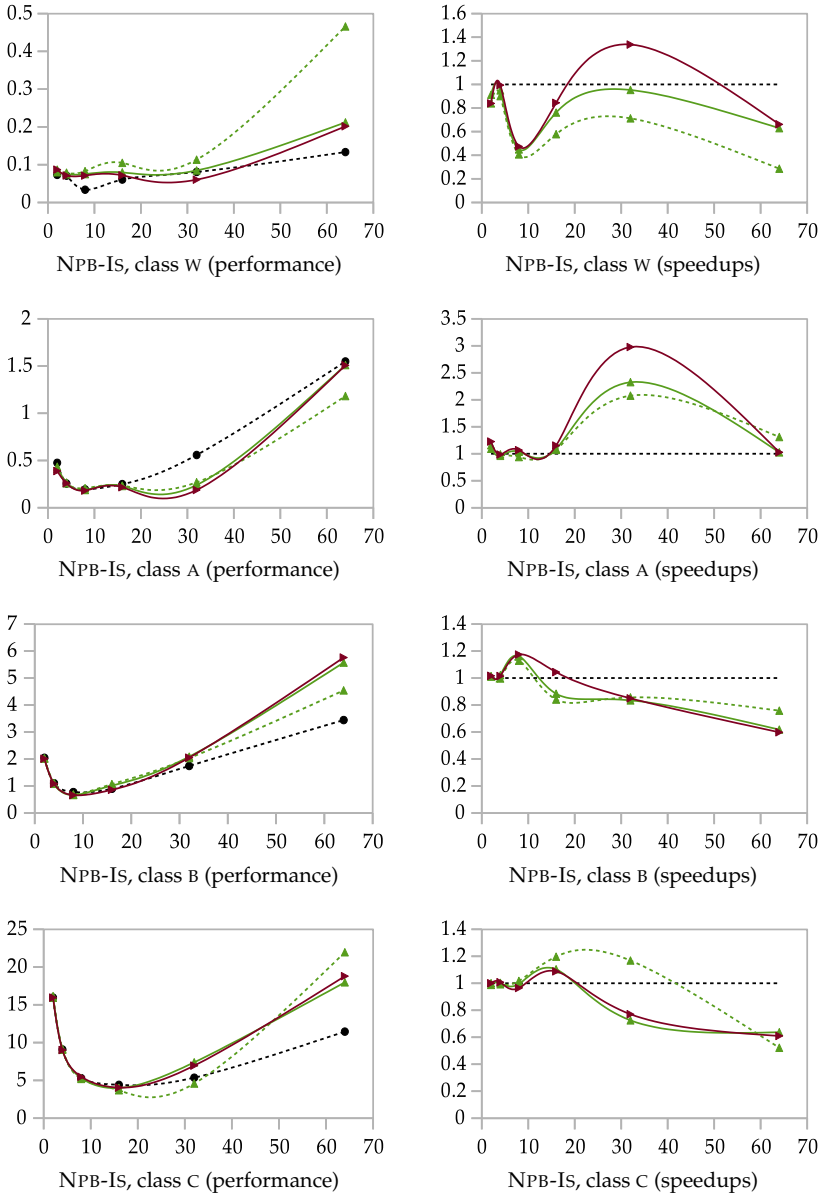


Figure 8.17: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

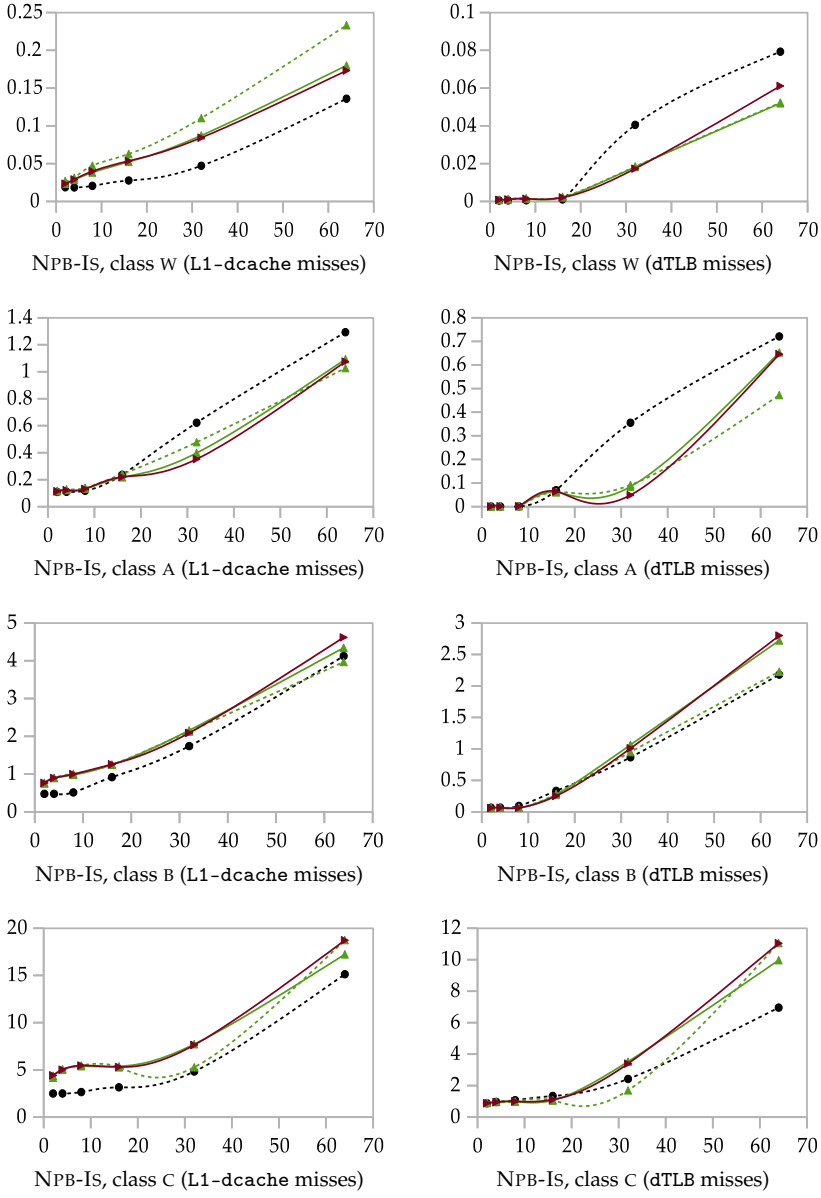


Figure 8.18: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

former versions actually incur substantially more cache misses. Assuming that these cache misses occur uniformly over the entire run time (recall from Chapter 5 that I had no choice but to measure cache misses from start to end instead of only during the interval of my time measurements, which make the numbers of cache misses reported here only an approximation), this suggests that the Java virtual machine needs to execute substantially fewer instructions for those FOCAML-to-Java-compiled versions than for those Java versions.

- An indication of improved performance and scalability, the FOCAML-to-Java-compiled versions of NPB-MG and NPB-CG outperform their Java versions not only in the larger problem size classes but also in the smaller problem size class W (improved performance), for larger values of k (improved scalability).

- As in Chapters 5, 6, and 7, differences in numbers of cache misses explain the perhaps confusing results for NPB-IS.

Figures 8.19–8.24 show performance charts for the FOCAML-to-Java-compiled versions of the NPB application benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The lines have the same meaning as in the figures with experimental results for the NPB kernel benchmarks. Recall from Figure 5.12 that NPB-BT and NPB-LU do not support more than 22 and 31 slaves, for which reason I have no measurements beyond $k = 16$ in class W for those benchmarks. In the same figure, note that NPB-BT, NPB-SP, and NPB-LU support at most 62 workers in class A. For that reason, as in Chapter 7, I compiled the FOCAML versions of those benchmarks for $k = 62$ instead of $k = 64$. Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks.

This concludes the NPB experiments that I report on in this thesis. To summarize my main findings, the series of experiments in Chapters 5, 6, 7, and 8 show that: (i) without any improvements, the FOCAML-to-Java-compiled versions perform substantially worse than the Java versions by Frumkin et al., (ii) the performance of these FOCAML-to-Java-compiled versions improves with every new improvement I introduce, (iii) with all these improvements in place, FOCAML-to-Java-compiled versions perform roughly as well—sometimes slightly worse, sometimes slightly better—than the Java versions by Frumkin et al., and (iv) memory and cache usage comprises an important future point of attention, as I state also in Chapter 9.

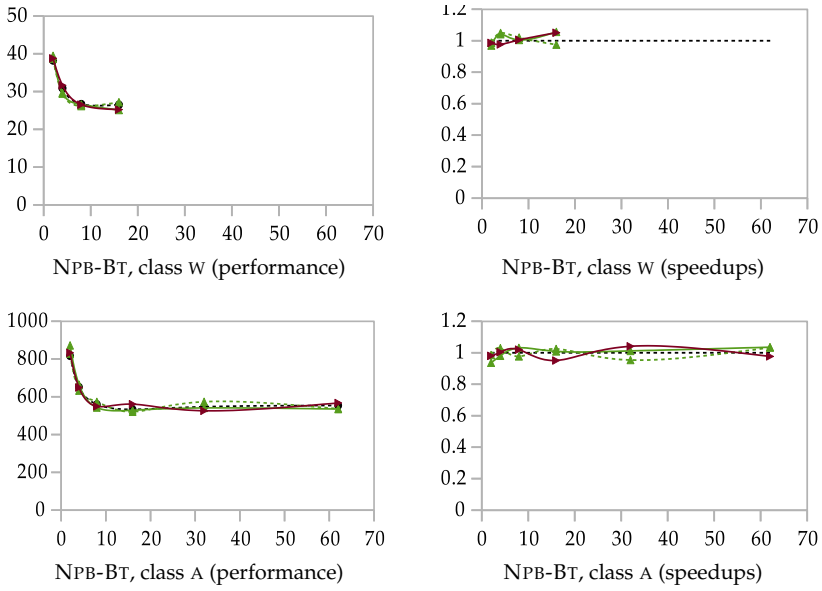


Figure 8.19: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

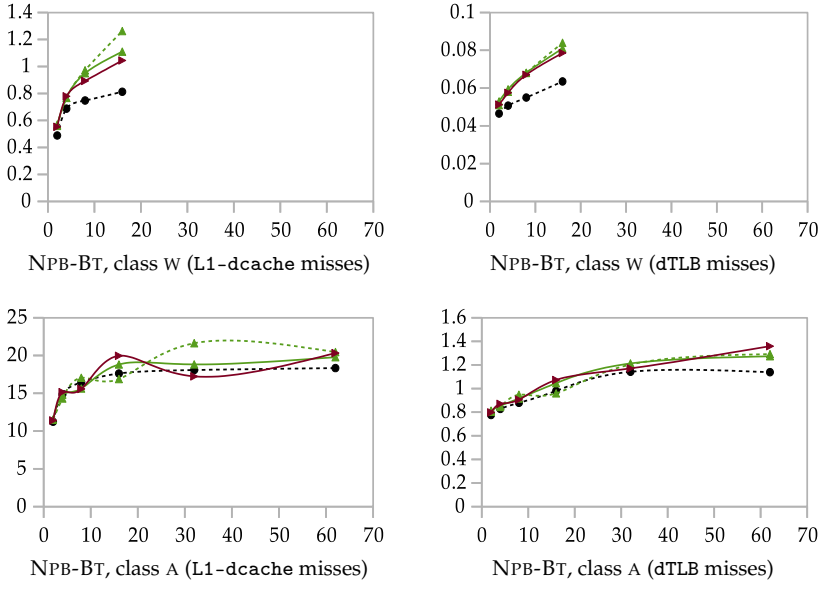


Figure 8.20: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

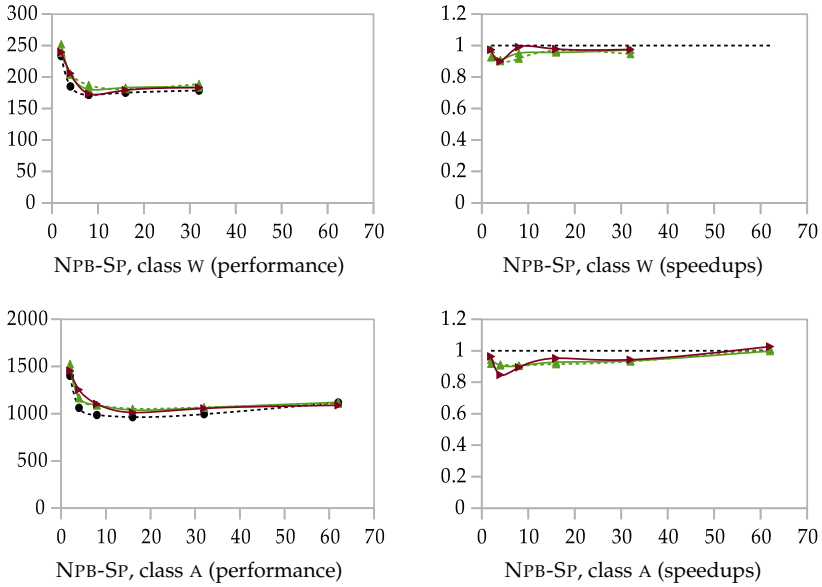


Figure 8.21: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

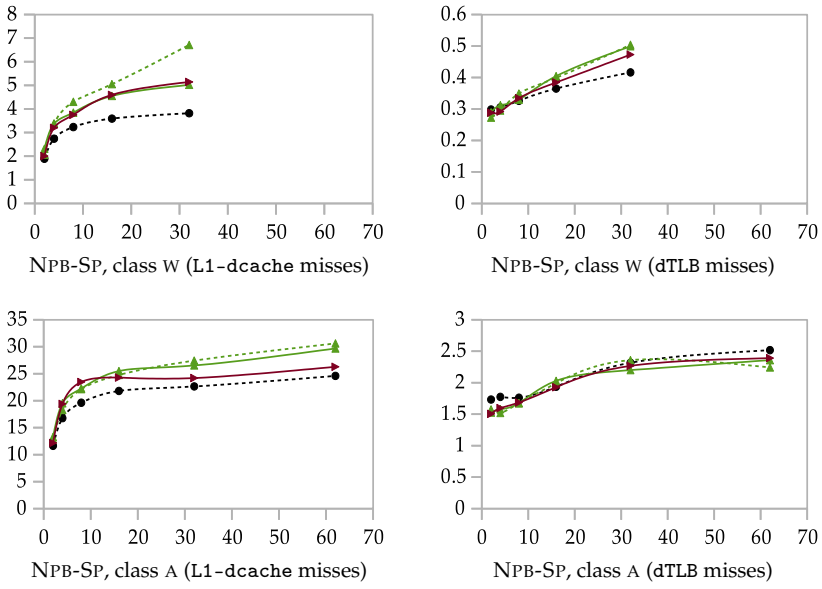


Figure 8.22: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

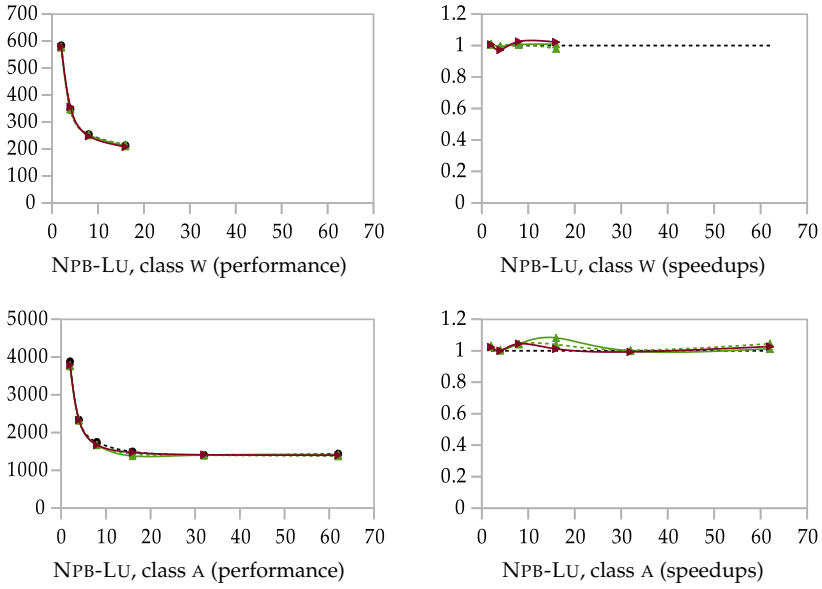


Figure 8.23: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

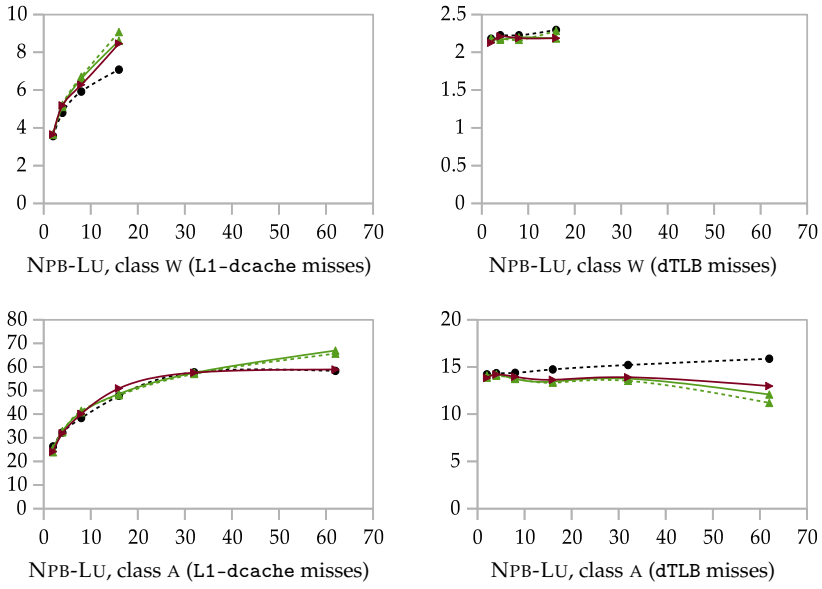


Figure 8.24: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.