Automata-theoretic protocol programming

Jongmans, Sung-Shik Theodorus Quirinus

Chapter 5

Improved Compilation I: Local Multiplication

The experimental results in Chapter 4 show that the Centralized Approach has two serious scalability problems. One of these problems manifests at compile-time (i.e., the inability to generate code for $\mathsf{Fifo}_{>6}$, $\mathsf{EarlyAsyncMerger}_{>4}$, and all FOCAML versions of the NPB benchmarks); the other manifests at run-time (i.e., the rapid performance degradation of Fifo_k as k increases, below inverse-proportionality).

In this chapter, to solve these problems, I develop a new compilation approach, which strikes a middle ground between the Distributed Approach and the Centralized Approach. In Section 5.1, I first provide a general description of this new compilation approach. Subsequently, I present the technical details of its most important new element: the computation of a partition of a set of constraint automata. These technicalities involve the introduction of a new multiplication and require a thorough study of the circumstances in which the old multiplication in Definition 29 coincides with this new multiplication. In Section 5.2, I present an improved version of Lykos using this new compilation approach, including new experimental results on performance.

Although the compilation approach presented in this chapter eventually results in improved compiler-generated code, 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.

5.1 Theory

*(With Arbab and Santini, I previously published fragments of the material in this section in conference/workshop papers [**J**A13a, **J**A14, **J**SA14] and journal papers [**J**A16, **J**SA15].)*

Hybrid Approach

I start by explaining the two scalability problems with the Centralized Approach in more detail. First, to explain its compile-time problem, I repeat the following observation from Chapter 4:

> "[...] in the Centralized Approach, compilation requires many resources [...] while execution requires few."

As the experimental results in Chapter 4 show, the Centralized Approach requires not *just* many resources but often *too* many resources. For instance, members of the $\mathsf{EarlyAsyncMerger}_k$ subfamily, defined in Figure 3.12, have as many as 2^k states. Each of those states models a permutation of the emptiness/fullness of k memory cells. Similarly, members of the FifoK_k subfamily have k memory cells, each of which can—at any instant—have content or not, yielding an exponentially-sized state space. Members of the $\mathsf{LateAsyncMerger}_k$ subfamily, in contrast, do not have exponentially-sized state spaces, because they have only one memory cell shared by all k producers (cf. one memory cell for every producer as members of EarlyAsyncMerger have). In any case, members of $\mathsf{EarlyAsyncMerger}_{256}$ have more states (roughly 10^{77}) than the observable universe has hydrogen atoms (overestimate: 10^{80}). Although anecdotal, this comparison "proves" not only the practical intractability of compiling instantiated family signatures such as `EarlyAsyncMerger(A[1..256],B)` under the Centralized Approach today but also its theoretical impossibility forever. Because 256 producers seems not unreasonably many, solving this *state space explosion problem* seems imperative.

To explain the run-time problem with the Centralized Approach, I repeat the following observation from Chapter 4:

> "a FOCAML compiler that generates code under the Centralized Approach yields protocol subprograms with low latency and low throughput."

Indeed, the compiler-generated protocol subprogram for a "large" automaton in the Centralized Approach defines exactly one protocol unit. Consequently, when generating code under the Centralized Approach, a FOCAML compiler effectively serializes all potential parallelism among the "small" automata that multiply into such a large automaton. My comparison between the Distributed Approach and the Centralized Approach in Chapter 4, involving `AliceBobCarolDave` in Figure 4.7, already showed such *oversequentialization*. Oversequentialization reduces throughput: in the worst case, the firing of one transition inhibits the parallel firing of other, completely independent transitions.

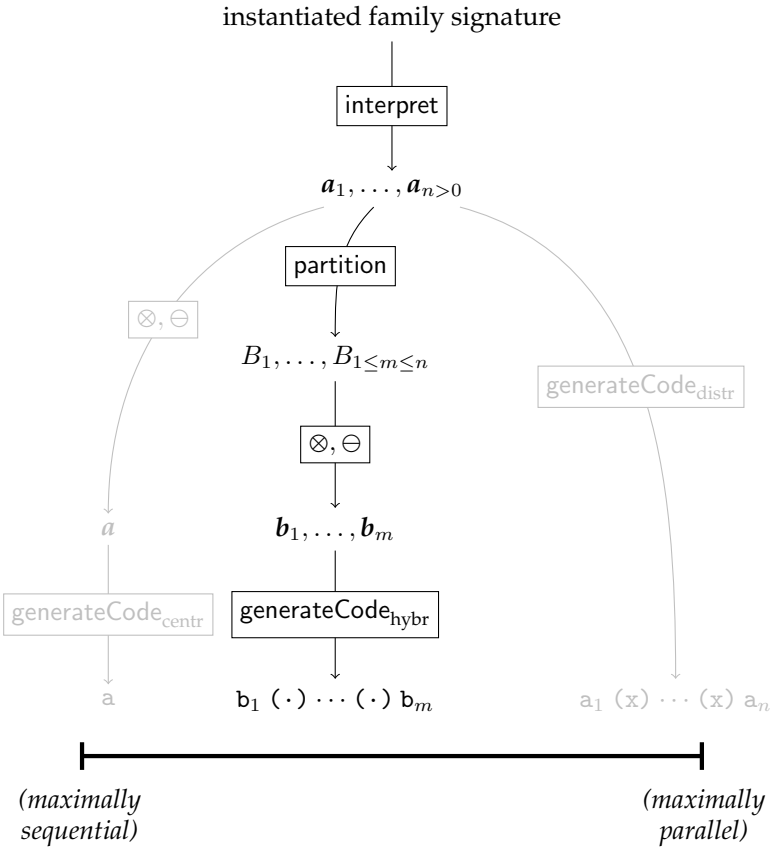


Figure 5.1: Hybrid compilation approach (cf. Figure 4.1)

As the number of independent transitions increases, this sequential bottleneck becomes an increasingly pressing problem.

To solve these problems of state space explosion and oversequentialization, I propose to adopt a new approach for FOCAML compilation: the *Hybrid Approach*, shown in Figure 5.1. On the sequentiality/parallelism-spectrum, the Hybrid Approach sits somewhere between the Centralized Approach and the Distributed Approach. The Hybrid Approach strikes a middle ground between those two ends: it sequentializes all *useless* parallelism to preserve only *useful* parallelism. I define useless/useful parallelism as follows.

- Protocol units exhibit useless parallelism whenever these protocol units must reach consensus about their global behavior before any of them can fire a local transition. In those cases, parallelism *does not* improve throughput, while the communication overhead of reaching consensus *does* reduce latency. Protocol units Alice, Bob, and Carol in the `AliceBobCarolDave` example in Chapter 4 illustrate useless parallelism: neither

Alice, nor Bob, nor Carol can fire a local transition without first having to communicate with the others.

Generally, protocol units that exhibit useless parallelism require an *expensive consensus algorithm*, with global communication among everyone. Previously, in Chapter 4, I denoted this algorithm by `(x)` and observed that this algorithm effectively multiplies constraint automata at run-time, in the sense of Definition 29.

- Protocol units exhibit useful parallelism whenever protocol units in one subset can fire local transitions independently of protocol units in another subset. In those cases, parallelism truly improves throughput. The subset of protocol units consisting of Alice, Bob, and Carol and the subset consisting of only Dave in the `AliceBobCarolDave` example illustrate useful parallelism: Dave can fire a local transition without first having to communicate with Alice, Bob, or Carol.

  Generally, protocol units that exhibit useful parallelism require a *cheap consensus algorithm*, with only local communication between neighbors. Henceforth, in this chapter, I denote this algorithm by `(·)` and argue that also this algorithm effectively multiplies constraint automata at run-time, albeit under a different—yet equivalent—multiplication.

Seen from the perspective of the Centralized Approach, the Hybrid Approach yields protocol subprograms that define $m \geq 1$ protocol units instead of just one (to preserve only useful parallelism); seen from the perspective of the Distributed Approach, the Hybrid Approach yields protocol subprograms that define $m \leq n$ protocol units instead of n (to sequentialize all useless parallelism). In cases of only useless parallelism, the Hybrid Approach reduces to the Centralized Approach (i.e., $m = 1$); in cases of only useful parallelism, the Hybrid Approach reduces to the Distributed Approach (i.e., $m = n$).

A FOCAML compiler that generates code under the Hybrid Approach takes four steps to generate GPL code on input of an instantiated family signature. In the first step, the compiler obtains a list of n small primitive constraint automata $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ in the same way as in the Distributed Approach and the Centralized Approach, by calling a FOCAML interpreter. In the second step, the compiler *partitions* the set of those small automata into m disjoint subsets $B_1, \ldots, B_m$. In particular, it computes a *reasonable partition*, where:

- protocol units for constraint automata in the same subset exhibit useless parallelism, while
- protocol units for constraint automata in different subset exhibit useful parallelism.

In the third step, the compiler multiplies the small automata in every part B_i and subtracts all internal ports, effectively serializing the useless parallelism among those automata. This step yields m "medium" composite constraint automata $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$. In the fourth step, the compiler translates these medium

automata to a protocol subprogram $\mathtt{b}_1\ \mathtt{(\cdot)} \cdots \mathtt{(\cdot)}\ \mathtt{b}_m$ in the GPL. This protocol subprogram defines m protocol units. Individually, every one of these protocol units locally simulates a medium automaton $\boldsymbol{b}_i$; collectively, these protocol units globally simulate the product of $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$. As in the Distributed Approach, to achieve the latter, the protocol units need to synchronize their local behavior with each other. Contrasting the Distributed Approach, however, by the construction of the reasonable partition in the second step, this synchronization needs to happen only locally between neighbors instead of globally among everyone. In Figure 5.1, I denote the code of the corresponding cheap consensus algorithm by `(·)`, placed between $\mathtt{b}_1, \ldots, \mathtt{b}_n$ to emphasize that this algorithm effectively multiplies constraint automata, just as the expensive consensus algorithm denoted by `(x)` does.

I discuss the event-handlers for protocol units that simulate medium automata later in this section. For now, suffice it to say that such event-handlers work roughly the same as the event-handler in Figure 4.5.

L-Multiplication

The main challenge with the Hybrid Approach lies in its second step: computing a reasonable partition in a potentially huge search space. After all, the number of unique partitions of a k-cardinality set grows superexponentially in k [Kla10]. In one corner of this search space, by putting every $\boldsymbol{a}_i$ in its own subset to get a partition $\{\{\boldsymbol{a}_1\}, \ldots, \{\boldsymbol{a}_n\}\}$, the Hybrid Approach reduces to the Distributed Approach; in its opposite corner, by putting every $\boldsymbol{a}_i$ in the same subset to get a partition $\{\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}\}$, the Hybrid Approach reduces to the Centralized Approach. Typically, however, neither of these two corners yields a reasonable partition. For now, I temporarily park the issue of computing reasonable partitions. Instead, at this point, just out of scientific curiosity, I study under which circumstances substituting `(x)` with `(·)` "preserves the original behavior" (i.e., under which circumstances synchronizing the behavior of protocol units with the cheap consensus algorithm instead of the expensive one preserves their original behavior, i.e., under which circumstances only local communication between neighbors can safely replace global communication among everyone). Incidentally, the insight resulting from this investigation yields an algorithm for computing reasonable partitions as well.

The previous observation that `(x)` applies $\otimes$ at run-time implies that—from the opposite perspective—$\otimes$ models `(x)`. Interestingly, I can similarly define *another* multiplication that models `(·)`. After doing so, instead of studying the interchangeability of `(x)` and `(·)` at the practical level of protocol units and their consensus algorithms, I can more conveniently study it at the theoretical level of constraint automata and their multiplications.

The "new" multiplication differs from the "old" multiplication only in how new transition relations come about. With the old multiplication in Definition 29, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ synchronously fire transitions that agree on the involvement of shared ports under a rather weak notion of agreement: in addition to their shared ports, $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ allow each other to involve also any number of un-

shared ports. This weak agreement, formalized in Definition 27 of $\Diamond$, gives rise to powerful multiparty and indirect synchronization as explained in Chapter 2. Exactly those properties, however, make `(x)` expensive: whenever a transition in constraint automaton $\boldsymbol{b}_1$ and a transition in constraint automaton $\boldsymbol{b}_k$ synchronously fire via transitions in constraint automata $\boldsymbol{b}_2, \ldots, \boldsymbol{b}_{k-1}$, at run-time, their protocol units must globally communicate to ensure that the protocol unit for $\boldsymbol{b}_1$ actually reaches consensus with the protocol unit for $\boldsymbol{b}_k$. With only local communication between neighbors, as stipulated for `(·)`, such multiparty and indirect synchronization cannot happen. Therefore, to model `(·)`, the new multiplication must restrict these forms of synchronization by strengthening the previous weak notion of agreement. Essentially, under the resulting *strong agreement*, constraint automata forbid each other to involve unshared ports in their synchronously firing transitions, whereas under the previous *weak agreement*, constraint automata allow each other to do so (which ultimately gives rise to multiparty and indirect synchronization). More precisely, under strong agreement, transitions in $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ can synchronously fire only if one of those transitions involves only shared ports, or if both transitions involve only unshared ports (i.e., at run-time, their protocol units communicate either only locally or not whatsoever).

Definition 31 (strong agreement)**.** $\blacklozenge \subseteq (2^{\mathbb{P}} \times 2^{\mathbb{P}}) \times (2^{\mathbb{P}} \times 2^{\mathbb{P}})$ *denotes the smallest relation induced by the following rule:*

$$\frac{P_1 \subseteq P_1^{\text{all}} \ \textbf{and} \ P_2 \subseteq P_2^{\text{all}} \ \textbf{and} \begin{bmatrix} P_1^{\text{all}} \cap P_2 = P_2^{\text{all}} \cap P_1 = \emptyset \\ \textbf{or} \ \ P_1 = P_1^{\text{all}} \cap P_2 \\ \textbf{or} \ \ P_2 = P_2^{\text{all}} \cap P_1 \end{bmatrix}}{(P_1^{\text{all}}, P_1) \blacklozenge (P_2^{\text{all}}, P_2)} \tag{5.1}$$

The following lemma states that strong agreement implies weak agreement. This, combined with Lemma 1, also means that $\blacklozenge$ actually constitutes an agreement relation in the sense of Definition 26.

Lemma 2. $(P_1^{\text{all}}, P_1) \blacklozenge (P_2^{\text{all}}, P_2)$ **implies** $(P_1^{\text{all}}, P_1) \Diamond (P_2^{\text{all}}, P_2)$

Lemma 3. $\blacklozenge \in \mathbb{A}\text{GREEM}$

The new multiplication, henceforth called *l(ocal)-multiplication*, consumes two constraint automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ as input and produces a new constraint automaton as output in nearly the same way as the old multiplication, henceforth called *g(lobal)-multiplication*: as g-multiplication, I define l-multiplication by instantiating the generalized multiplication in Definition 28 with strong agreement (instead of with weak agreement as in Definition 29 of $\otimes$).

Definition 32 (l-multiplication)**.** $\odot : \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM} \rightharpoonup \mathbb{A}\text{UTOM}$ *denotes the partial function defined by the following equation:*

$$\boldsymbol{a}_1 \odot \boldsymbol{a}_2 = \boldsymbol{a}_1 \circledast_{\blacklozenge} \boldsymbol{a}_2$$

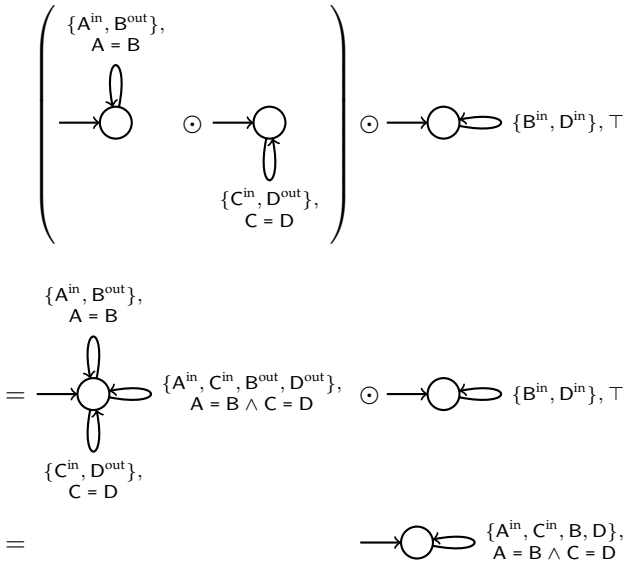


Figure 5.2: Left-associative l-multiplication of $\mathsf{Sync}(\mathsf{A};\mathsf{B})$, $\mathsf{Sync}(\mathsf{C};\mathsf{D})$, and $\mathsf{SyncDrain}(\mathsf{B},\mathsf{D};)$

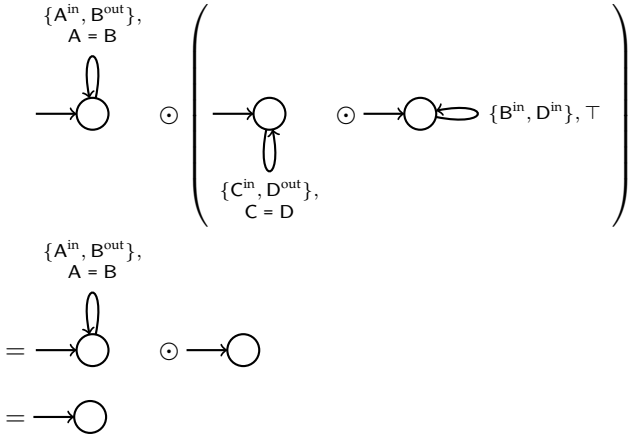


Figure 5.3: Right-associative l-multiplication of $\mathsf{Sync}(\mathsf{A};\mathsf{B})$, $\mathsf{Sync}(\mathsf{C};\mathsf{D})$, and $\mathsf{SyncDrain}(\mathsf{B},\mathsf{D};)$

L-multiplication satisfies commutativity and idempotence (up-to behavioral congruence), but in contrast to g-multiplication, it does not satisfy associativity: generally, $(\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot \boldsymbol{a}_3 \not\approx \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3)$. Figures 5.2 and 5.3 exemplify this phenomenon: both the l-product in Figure 5.2 and the l-product in Figure 5.3 have one state, but the former l-product has one transition, whereas the

latter l-product has no transitions. I postpone a more detailed discussion of l-multiplication's nonassociativity until later in this section. To minimize numbers of parentheses, I assume right-associative notation for $\odot$. For instance, I write $\boldsymbol{a}_1 \odot \boldsymbol{a}_2 \odot \boldsymbol{a}_3 \odot \boldsymbol{a}_4$ for $\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_3 \odot \boldsymbol{a}_4))$. The following theorem states that $\simeq$ denotes a congruence under $\odot$.

Theorem 6. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_3, \boldsymbol{a}_2 \odot \boldsymbol{a}_4 \in \mathbb{A}\text{UTOM} \\ \textbf{and } \boldsymbol{a}_1 \simeq \boldsymbol{a}_2 \textbf{ and } \boldsymbol{a}_3 \simeq \boldsymbol{a}_4 \end{bmatrix}$ **implies** $\boldsymbol{a}_1 \odot \boldsymbol{a}_3 \simeq \boldsymbol{a}_2 \odot \boldsymbol{a}_4$

First Characterization

The fact that g-multiplication satisfies associativity, whereas l-multiplication does not, already implies that substituting $\otimes$ with $\odot$ not always "preserves the original behavior" (in the sense previously explained). For instance, the g-product of the constraint automata in Figure 5.3 (under the same placement of parentheses) equals the l-product in Figure 5.2 but not the l-product in Figure 5.3. To determine when substituting $\otimes$ with $\odot$ preserves the original behavior, I first study under which conditions a g-product of two constraint automata simulates their l-product and vice versa.

Lemma 2 immediately implies that $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ simulates $\boldsymbol{a}_1 \odot \boldsymbol{a}_2$. In other words, a g-product of two constraint automata has at least the same transitions as their l-product. At run-time, this means that protocol units that use a consensus algorithm with global communication can effectuate at least the same instances of interaction as those that use a consensus algorithm with only local communication. This makes perfect sense. The inverse statement, in contrast, does not: protocol units that use a consensus algorithm with only local communication may not effectuate the same instances of interaction as those that use a consensus algorithm with global communication. In terms of constraint automata, this corresponds to the fact that $\boldsymbol{a}_1 \odot \boldsymbol{a}_2$ not necessarily simulates $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$. Indeed, if transitions of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ agree on the involvement of their shared ports (which $\otimes$ requires), this does not necessarily mean that they involve no other ports (which $\odot$ additionally requires). To characterize the cases in which it does, I define *conditional strong agreement* as a relation "between" $\blacklozenge$ and $\lozenge$ (and lifted from transitions to constraint automata): $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ conditionally strongly agree iff, for each of their transitions, their weak agreement on the involvement of their shared ports implies their strong agreement.

Definition 33 (Conditional strong agreement)**.** $\langle\!\blacklozenge \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\begin{bmatrix} q_1 \xrightarrow{P_1,\phi_1}_1 q_1' \textbf{ and } q_2 \xrightarrow{P_2,\phi_2}_2 q_2' \\ \textbf{and } (P_1^{\text{all}}, P_1) \lozenge (P_2^{\text{all}}, P_2) \end{bmatrix} \textbf{ implies } (P_1^{\text{all}}, P_1) \blacklozenge (P_2^{\text{all}}, P_2)\right] \quad \textbf{for all } q_1, q_2, q_1', q_2', P_1, P_2, \phi_1, \phi_2}{(\cdot, (P_1^{\text{all}}, P_1^{\text{in}}, P_1^{\text{out}}), \cdot, \longrightarrow_1, \cdot) \langle\!\blacklozenge (\cdot, (P_2^{\text{all}}, P_2^{\text{in}}, P_2^{\text{out}}), \cdot, \longrightarrow_2, \cdot)} \tag{5.2}$$

The following lemma states that conditional strong agreement between two constraint automata implies the behavioral congruence of their l-product and their g-product.

Lemma 4. $\left[\boldsymbol{a}_1 \mathbin{\langle\!\blacklozenge\!\rangle} \boldsymbol{a}_2 \text{ and } \boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM}\right]$ **implies** $\boldsymbol{a}_1 \odot \boldsymbol{a}_2 \simeq \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$

As a generalization of Lemma 4, suppose that I have a list of k constraint automata such that every i-th constraint automaton in this list conditionally strongly agrees with the l-product of all higher positioned ones. The order of the constraint automata matters, because $\odot$ does not exhibit associativity. The following theorem states that multiplying all constraint automata in the previous list with $\odot$ or $\otimes$, starting from the ones in the highest positions (i.e., the most deeply nested ones under right-associative notation), yields behaviorally congruent products.

Theorem 7.

$$\left[\begin{array}{l}\left[\begin{array}{l}1 \leq i < k \text{ implies} \\ \boldsymbol{a}_i \mathbin{\langle\!\blacklozenge\!\rangle} \boldsymbol{a}_{i+1} \odot \cdots \odot \boldsymbol{a}_k\end{array}\right] \text{ for all } i \\ \text{and } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \in \mathbb{A}\text{UTOM}\end{array}\right] \text{ implies } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_k$$

I call the premise in the previous theorem the $\langle\!\blacklozenge\!\rangle$*-based characterization* of when substituting $\otimes$ with $\odot$ preserves the original behavior.

The $\langle\!\blacklozenge\!\rangle$-based characterization has two disadvantages. First, to test if two constraint automata $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ conditionally strongly agree, one must pairwise compare their transitions. By itself, this may already require a significant amount of computation (i.e., $\mathcal{O}(k_1 k_2)$ for $k_1 = |\mathsf{Trans}(\boldsymbol{a}_1)|$ and $k_2 = |\mathsf{Trans}(\boldsymbol{a}_2)|$). Moreover, the $\langle\!\blacklozenge\!\rangle$-based characterization requires conditional strong agreement not between individual constraint automata in the list but between their l-products. Because l-multiplication generally does not preserve conditional strong agreement, checking a list of constraint automata for satisfaction of the $\langle\!\blacklozenge\!\rangle$-based characterization requires the potentially expensive computation of many l-products. These disadvantages make the $\langle\!\blacklozenge\!\rangle$-based characterization unattractive in practice. In the next subsection, I therefore study a cheaper characterization.

Cheaper Characterization

I develop a cheaper characterization through a number of observations. First, using only local communication between protocol units instead of global communication clearly preserves the original behavior of *independent* protocol units that do not need to communicate with each other whatsoever. Thus, substituting $\otimes$ with $\odot$ should preserve the original behavior at least when applied to constraint automata corresponding to such independent protocol units. I start by formally defining the notion of independence.

Definition 34 (independence). $\asymp \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{P_1^{\text{all}} \cap P_2^{\text{all}} = \emptyset}{(\cdot, (P_1^{\text{all}}, \cdot, \cdot), \cdot, \cdot, \cdot) \asymp (\cdot, (P_2^{\text{all}}, \cdot, \cdot), \cdot, \cdot, \cdot)} \tag{5.3}$$

The following lemmas state (i) that independent constraint automata conditionally strongly agree with each other and (ii) that l-multiplication preserves independence.

Lemma 5. $\boldsymbol{a_1 \asymp a_2}$ **implies** $\boldsymbol{a_1 \blacklozenge a_2}$

Lemma 6. $\left[\boldsymbol{a_1 \odot a_2 \in \mathbb{A}\text{UTOM}}\right.$ **and** $\left.\boldsymbol{a \asymp a_1, a_2}\right]$ **implies** $\boldsymbol{a \asymp a_1 \odot a_2}$

Lemmas 5 and 6 and Theorem 7 imply that substituting $\otimes$ with $\odot$ preserves the original behavior if their multiplicands satisfy independence. Moreover, checking for independence requires fewer resources than checking for conditional strong agreement, namely $\mathcal{O}(1)$ instead of $\mathcal{O}(k_1 k_2)$.

Although checking constraint automata for independence costs close to nothing, the result implied by Lemmas 5 and 6 and Theorem 7 in its present form has limited practical value: independent constraint automata only rarely occur outside artificial examples. To get a more useful result, I introduce the notion of *slavery* and afterward combine it with independence. I start by formally defining when a constraint automaton $\boldsymbol{a_2}$ has enslaved a constraint automaton $\boldsymbol{a_1}$. In that case, every transition in $\boldsymbol{a_1}$ that involves *some* port shared with $\boldsymbol{a_2}$, involves *only* ports shared with $\boldsymbol{a_2}$. In other words, $\boldsymbol{a_2}$ completely dictates what $\boldsymbol{a_1}$ does whenever a transition in $\boldsymbol{a_1}$ involves at least one of their shared ports. Importantly, this notion of slavery does not forbid $\boldsymbol{a_1}$ from firing transitions that involve only ports that $\boldsymbol{a_2}$ does not know about (i.e., slaves can secretly "rebel"). This enables other constraint automata to enslave $\boldsymbol{a_1}$ as well (albeit with respect to different ports).

Definition 35 (slavery). $\mapsto \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\begin{bmatrix} q_1 \xrightarrow{P_1, \phi_1}_1 q_1' \\ \textbf{and } P_1 \cap P_2^{\text{all}} \neq \emptyset \end{bmatrix} \textbf{ implies } P_1 \subseteq P_2^{\text{all}}\right] \textbf{ for all } q_1, q_1', P_1, \phi_1}{(\cdot, \cdot, \cdot, \longrightarrow_1, \cdot) \mapsto (\cdot, (P_2^{\text{all}}, \cdot, \cdot), \cdot, \cdot, \cdot)} \tag{5.4}$$

The following lemmas state that if one constraint automaton has enslaved another constraint automaton, they conditionally strongly agree with each other and that l-multiplication preserves slavery.

Lemma 7. $\boldsymbol{a_1 \mapsto a_2}$ **implies** $\boldsymbol{a_1 \blacklozenge a_2}$

Lemma 8. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \textbf{ and } \boldsymbol{a} \mapsto \boldsymbol{a}_1 \\ \textbf{and } \left[\boldsymbol{a} \asymp \boldsymbol{a}_2 \textbf{ or } \boldsymbol{a} \mapsto \boldsymbol{a}_2\right] \end{bmatrix}$ **implies** $\boldsymbol{a} \mapsto \boldsymbol{a}_1 \odot \boldsymbol{a}_2$

Lemma 9. $\begin{bmatrix} \boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \textbf{ and } \boldsymbol{a} \mapsto \boldsymbol{a}_2 \\ \textbf{and } \left[\boldsymbol{a} \asymp \boldsymbol{a}_1 \textbf{ or } \boldsymbol{a} \mapsto \boldsymbol{a}_1\right] \end{bmatrix}$ **implies** $\boldsymbol{a} \mapsto \boldsymbol{a}_1 \odot \boldsymbol{a}_2$

Lemmas 7, 8, and 9 and Theorem 7 imply that substituting $\otimes$ with $\odot$ preserves the original behavior if their multiplicands satisfy slavery. Moreover, checking for slavery requires fewer resources than checking for conditional strong agreement, namely $\mathcal{O}(k_1)$ instead of $\mathcal{O}(k_1 k_2)$.

By combining independence and slavery, I obtain *conditional slavery*.

Definition 36 (conditional slavery)**.** $\between \subseteq \mathbb{A}\text{UTOM} \times \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\boldsymbol{a}_1 \not\asymp \boldsymbol{a}_2 \textbf{ implies } \boldsymbol{a}_1 \mapsto \boldsymbol{a}_2}{\boldsymbol{a}_1 \between \boldsymbol{a}_2} \tag{5.5}$$

The following lemmas state (i) that if one constraint automaton has conditionally enslaved another constraint automaton, they conditionally strongly agree and (ii) that l-multiplication preserves conditional slavery.

Lemma 10. $\boldsymbol{a}_1 \between \boldsymbol{a}_2$ **implies** $\boldsymbol{a}_1 \blacklozenge \boldsymbol{a}_2$

Lemma 11. $\left[\boldsymbol{a}_1 \odot \boldsymbol{a}_2 \in \mathbb{A}\text{UTOM} \textbf{ and } \boldsymbol{a} \between \boldsymbol{a}_1, \boldsymbol{a}_2\right]$ **implies** $\boldsymbol{a} \between \boldsymbol{a}_1 \odot \boldsymbol{a}_2$

Lemmas 10 and 11 and Theorem 7 imply that substituting $\otimes$ with $\odot$ preserves the original behavior if their multiplicands satisfy conditional slavery. Moreover, checking for conditional slavery costs the same as checking for slavery (i.e., less than checking for conditional strong agreement).

With conditional slavery, in contrast to independence alone, I can define a sufficiently powerful characterization of when substituting $\otimes$ with $\odot$ preserves the original behavior. Similar to Theorem 7, suppose that I have a list of k constraint automata such that every i-th constraint automaton in this list has conditionally enslaved all constraint automata in a lower position. The following theorem states that multiplying all constraint automata in the previous list with $\odot$ or $\otimes$, starting from the ones in the highest positions (i.e., the most deeply nested ones under right-associative notation), yields behaviorally congruent products.

Theorem 8.

$$\begin{bmatrix} \left[\begin{bmatrix} 1 \leq i < k \textbf{ implies} \\ \boldsymbol{a}_i \between \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_k \end{bmatrix} \textbf{ for all } i\right] \\ \textbf{and } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \in \mathbb{A}\text{UTOM} \end{bmatrix} \textbf{ implies } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_k \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_k$$

I call the premise in the previous theorem the $\between$*-based characterization* of when substituting $\otimes$ with $\odot$ preserves the original behavior.

To give Theorem 8 a more natural interpretation, I strengthen its premise: the following theorem states that substituting $\otimes$ with $\odot$ preserves the original behavior whenever I have (i) k constraint automata conditionally enslaved by *all* other constraint automata and (ii) l pairwise independent "master" constraint automata.

Theorem 9.

$$\left[\left[\begin{bmatrix}1 \leq i \leq k \textbf{ implies}\\ \boldsymbol{a}_i \bowtie \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1},\\ \qquad \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_{k+l}\end{bmatrix} \textbf{ and } \begin{bmatrix}k+1 \leq i \leq k+l \textbf{ implies}\\ \boldsymbol{a}_i \asymp \boldsymbol{a}_{k+1}, \ldots, \boldsymbol{a}_{i-1},\\ \qquad \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_{k+l}\end{bmatrix}\right] \textbf{ for all } i\right]$$
$$\textbf{and } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \in \mathbb{A}\text{UTOM}$$
$$\textbf{implies } \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_{k+l}$$

Because of their pairwise independence, protocol units for masters never *directly* communicate with each other. If multiple masters share the same slave, though, their protocol units may communicate *indirectly* with each other, via that slave. Such indirect communication occurs always asynchronously: otherwise, in case of synchronous communication, the slave would have a transition involving ports shared with multiple masters, which slavery forbids.

The previous interpretation of constraint automata as masters and slaves corresponds to the notion of synchronous and asynchronous *regions* in the Reo literature, perhaps first mentioned by Clarke et al. [CCA07]. Roughly, one can always split a Reo circuit into subcircuits—its regions—such that interaction on ports in such a subcircuit occurs always either asynchronously (i.e., every firing transition involves at most one port) or eventually synchronously (i.e., at least one firing transition involves more than one port). Circuits have *maximal* synchronous regions in the sense that no two synchronous regions have shared ports: every circuit has, by definition, only pairwise independent synchronous regions. Consequently, the constraint automata for the synchronous regions of a circuit can act as the l masters in Theorem 9. Dually, circuits have *minimal* asynchronous regions in the sense that no asynchronous region consists of more than one primitive. Asynchronous regions effectively constitute asynchronous communication mediums between synchronous regions. Consequently, the constraint automata for the asynchronous regions of a circuit can act as a the k conditional slaves in Theorem 9.

Synchronous and asynchronous regions play an important role in Reo compilers/interpreters that *split* circuits along the boundaries of their regions to decouple those regions' execution and improve performance. For his PhD thesis [Pro11], Proença developed the first implementation based on these ideas and invented a new automaton model to reason about split circuits [PCdVA11, PCdVA12]. Later, Clarke and Proença studied circuit splitting in the context of *coloring semantics* [CP12]. They discovered that the standard version of coloring semantics has undesirable properties in the context of circuit splitting: some split circuits that intuitively *should* behave as their originals nevertheless have inequivalent coloring semantics. To address this problem, Clarke

and Proença proposed a new variant of coloring semantics that better supports locality and independence. Before developing the Hybrid Approach as presented in this section, with Clarke and Proença, I worked on a formalization of circuit splitting in a process algebraic setting [**J**CP12, **J**CP16]. All this earlier work on circuit splitting strongly inspired and influenced me in developing the Hybrid Approach.

Practical Characterization

Although cheaper than the $\langle\!\blacklozenge$-based characterization, the $\bowtie$-based characterization still requires relatively many computational resources in practice. Therefore, I strengthen it once more by introducing another relation on constraint automata: *no-synchronization*. Informally, a constraint automaton exhibits no-synchronization if it never synchronizes any of its ports (i.e., each of its transitions has a singleton synchronization constraint).

Definition 37 (no-synchronization). $\xrightarrow{1} \subseteq \mathbb{A}\text{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[q \xrightarrow{P,\phi} q' \ \textbf{implies} \ |P| = 1\right] \ \textbf{for all} \ q, q', P, \phi}{\xrightarrow{1} (\cdot, \cdot, \cdot, \longrightarrow, \cdot)} \tag{5.6}$$

The following lemma states that no-synchronization implies conditional slavery.

Lemma 12. $\xrightarrow{1} \boldsymbol{a}_1 \ \textbf{implies} \ \boldsymbol{a}_1 \bowtie \boldsymbol{a}_2$

The following theorem follows from Lemma 12 and Theorem 9.

Theorem 10.

$$\left[\left[\begin{bmatrix} 1 \leq i \leq k \\ \textbf{implies} \ \xrightarrow{1} \boldsymbol{a}_i \end{bmatrix} \textbf{and} \begin{bmatrix} k+1 \leq i \leq k+l \ \textbf{implies} \\ \boldsymbol{a}_i \asymp \boldsymbol{a}_{k+1}, \ldots, \boldsymbol{a}_{i-1}, \\ \boldsymbol{a}_{i+1}, \ldots, \boldsymbol{a}_{k+l} \end{bmatrix}\right] \textbf{for all} \ i\right] \textbf{and} \ \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \in \mathbb{A}\text{UTOM}$$
$$\textbf{implies} \ \boldsymbol{a}_1 \odot \cdots \odot \boldsymbol{a}_{k+l} \simeq \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_{k+l}$$

I call the premise in the previous theorem the $\xrightarrow{1}$*-based characterization* of when substituting $\otimes$ with $\odot$ preserves the original behavior.

Now, recall from earlier in this section that I actually wanted to find an algorithm for computing reasonable partitions. Incidentally, the $\xrightarrow{1}$-based characterization yields such an algorithm, namely Algorithm 1. Algorithm 1 iterates over an input set of n constraint automata and terminates in $\mathcal{O}(n^2)$. In each iteration, either it puts the current constraint automaton $\boldsymbol{a}_i$ in a new subset (if $\boldsymbol{a}_i$ satisfies no-synchronization), or it computes a new subset for $\boldsymbol{a}_i$, possibly including existing parts, such that the new subset contains all constraint

Algorithm 1 Algorithm for partitioning a set of constraint automata A

Require: $A^{\text{small}} = \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$

function ALGORITHM1(A^{small})
  $\mathcal{A}^{\text{sl}} := \emptyset$
  $\mathcal{A}^{\text{m}} := \emptyset$
  $i := 1$
  while $i \leq n$ **do**
    if $\xrightarrow{1} \boldsymbol{a}_i$ **then**
      $\mathcal{A}^{\text{sl}} := \mathcal{A}^{\text{sl}} \cup \{\{\boldsymbol{a}_i\}\}$
    else
      $\overline{\mathcal{A}} := \{\overline{A} \mid \overline{\boldsymbol{a}} \in \overline{A} \in \mathcal{A}^{\text{m}} \textbf{ and } \boldsymbol{a}_i \not\asymp \overline{\boldsymbol{a}}\}$
      $\mathcal{A}^{\text{m}} := (\mathcal{A}^{\text{m}} \setminus \overline{\mathcal{A}}) \cup \{\{\boldsymbol{a}_i\} \cup \bigcup \overline{\mathcal{A}}\}$
    $i := i + 1$
  return $\mathcal{A}^{\text{sl}} \cup \mathcal{A}^{\text{m}}$

Ensure:
$$\left[\begin{array}{l} \mathcal{A}^{\text{sl}} \cup \mathcal{A}^{\text{m}} \text{ denotes a partition of } A^{\text{small}} \\ \textbf{and } \mathcal{A}^{\text{sl}} = \{A_1, \ldots, A_k\} \textbf{ and } \mathcal{A}^{\text{m}} = \{A_{k+1}, \ldots, A_{k+l}\} \\ \textbf{and } \left[\left[1 \leq j \leq k \textbf{ implies } \xrightarrow{1} \bigotimes A_j\right] \textbf{ for all } j\right] \\ \textbf{and } \left[\left[\begin{array}{c} k+1 \leq j \leq k+l \textbf{ implies} \\ \bigotimes A_j \asymp \bigotimes A_{k+1}, \ldots, \bigotimes A_{j-1}, \\ \bigotimes A_{j+1}, \ldots, \bigotimes A_{k+l} \end{array}\right] \textbf{ for all } j\right] \\ \qquad \textbf{for some } A_1, \ldots, A_{k+l}, k, l \end{array}\right]$$

automata dependent—directly or indirectly—on $\boldsymbol{a}_i$. The following theorem states the algorithm's correctness: it yields a partition of the input set and this partition satisfies the premise in Theorem 10.

Theorem 11. *Algorithm 11 is correct.*

Recall that a partition qualifies as reasonable if protocol units for constraint automata in the same subset exhibit useless parallelism, while protocol units for constraint automata in different parts exhibit useful parallelism. With respect to the first requirement, because constraint automata in the same subset all depend—directly or indirectly—on each other, their protocol units exhibit useless parallelism. With respect to the second requirement, by ordering medium automata as in Theorem 10 (Theorem 11 guarantees the existence of a list so ordered), I can safely substitute all g-multiplications between those medium automata with l-multiplications. At run-time, because g- and l-multiplication model the expensive and the cheap consensus algorithm, the protocol units for those medium automata require only the cheap consensus algorithm instead of the expensive one and, as such, exhibit useful parallelism. Thus, the partition computed by Algorithm 1 indeed qualifies as reasonable.

The Hybrid Approach, and Algorithm 1 in particular, directly address the problem of oversequentialization as introduced in the beginning of this chapter. In contrast, the Hybrid Approach does not directly address the problem of state space explosion. Nevertheless, Algorithm 1 solves also this other problem *at least in this thesis*. To see this, recall from Figure 3.4 that this thesis' core set contains only one family of constraint automata with more than one state—Fifo—each of whose members satisfies no-synchronization. Consequently, every Fifo gets its own singleton part, while all nonsingleton parts contain only single-state primitives. Because the product of any number of single-state constraint automata has only one state as well, and because multiplication occurs only on a per-subset basis in the Hybrid Approach, no state space explosion can happen under the core set in Figure 3.4. Generally, if every family of constraint automata with more than one state in a core set consists only of members that satisfy no-synchronization, the Hybrid Approach has no state space explosion problem.

Suppose that I can construct every constraint automaton out of members of the families in Figure 3.4. Moreover, suppose that I have an algorithm for *decomposing* every constraint automaton into such instances. Then, regardless of the particular primitives in a core set, the Hybrid Approach *never* suffers from state space explosion. After all, in that case, I can decompose every constraint automaton with more than one state into a number of Fifos and a number of single-state primitives. For arbitrary constraint automata, at this point, I have evidence neither for nor against the feasibility of this approach. Other people have worked on this topic for particular classes of constraint automata, though: Arbab et al. and Baier et al. devised multiple algorithms for decomposing "original constraint automata" [ABdB+05, BKK14], Koehler and Clarke proved a decomposition theorem for port automata [KC09], while Pourvatan et al. developed a *division* for constraint automata with state memory [PSAB12].

Related Work on Distributed Coordination

Three decades ago, in the mid 1980s, Gelernter introduced the coordination language Linda [Gel85]. At the heart of Linda lies the concept of a *tuple space*, a structure in which both workers and tuples of data, originating from and accessible to those workers, "float". Although a tuple space gives the programmer the illusion of shared memory, at the hardware level, this memory may actually reside at n different locations. Several approaches to implementing physically distributed tuple spaces exist. For instance, one can maintain the entire tuple space at one of the n locations (e.g., Feng et al. [FWY96], Wyckoff et al. [WMLF98]), but although simple to implement, this does not scale well in the number of computation processes [FGY94]. The Centralized Approach presented in Chapter 4 actually has a similar scalability problem (due to oversequentialization). Alternatively, one can scatter (with or without replication) the tuples in the tuple space over all n locations. Although such an approach has better scalability, one must resolve several issues to obtain a workable implementation, such as deciding where to store which tuple, efficiently

retrieving tuples, and load balancing [PA98]. Examples include the work by Bjornson [Bjo93], Feng et al. [FGY94], Rowstron and Wood [RW96], Menezes and Tolksdorf [MT03], and Atkinson [Atk10]. Although both distributed tuple spaces and the Hybrid Approach facilitate a form of distributed coordination, they differ in one fundamental aspect: whereas distributed tuple spaces distribute data (i.e., tuples), the Hybrid Approach distributes control (i.e., medium automata).

Bonakdarpour et al. worked on an approach for automatically generating distributed implementations for specifications in BIP [BBJ+12], a framework for specifying component-based systems at three specification levels [BBS06]: behavior of components, interaction between components, and priorities on interaction. BIP forbids simultaneous execution of conflicting instances of interaction (i.e., instances that involve overlapping sets of ports). In automatically generated distributed implementations of BIP specifications, therefore, Bonakdarpour et al. have to ensure that such conflicting interactions execute mutually exclusively. To achieve this, Bonakdarpour et al. propose a three-layered implementation architecture: the bottom layer consists of distributed components, the middle layer consists of a number of interaction execution engines, each responsible for executing its own subset of all interactions, and the top layer resolves potential conflicts. In terms of this thesis and the Hybrid Approach, the bottom layer represents workers, while the middle layer roughly represents a multiplication expression of constraint automata. Importantly, however, Bonakdarpour et al. aim for a finer distribution granularity than I do, which requires them to handle conflicting interactions with their third layer. I avoid this problem in the Hybrid Approach, by putting constraint automata with "conflicting transitions" in the same subset at compile-time, thereby effectively serializing those transitions at run-time; for performance reasons, I prefer firing such transitions sequentially over adding an algorithm for conflict resolution.

Nonassociativity

G-multiplication satisfies associativity, whereas l-multiplication does not, as already shown in Figures 5.2 and 5.3. So far, I worked around this limitation by formulating the premises in Theorems 7, 9 and 10 in terms of carefully ordered lists of constraint automata. Although this works fine in theory, the need for such lists has a problematic consequence in practice: because $\odot$ models $(\cdot)$ (i.e., the cheap consensus algorithm, with only local communication between neighbors), the order of the constraint automata in the list at compile-time essentially fixes the order in which their protocol units may communicate with each other to reach consensus at run-time. In other words, because $(\cdot)$ actually l-multiplies constraint automata, and because l-multiplication does not satisfy associativity, $(\cdot)$ must ensure that its run-time multiplication "abides by compile-time parentheses". For instance, suppose that the third step in the Hybrid Approach yields three medium automata $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, and $\boldsymbol{b}_3$. Moreover, suppose that $\boldsymbol{b}_1 \odot (\boldsymbol{b}_2 \odot \boldsymbol{b}_3) \simeq \boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ by Theorem 10, whereas $(\boldsymbol{b}_1 \odot \boldsymbol{b}_2) \odot \boldsymbol{b}_3 \not\simeq$

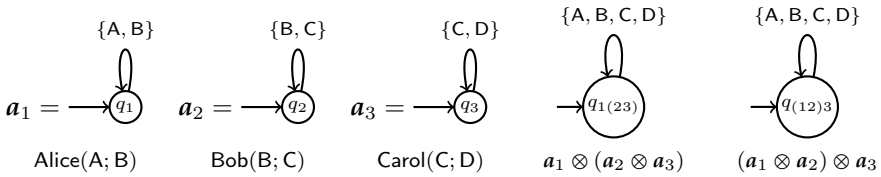


Figure 5.4: Two g-products of Alice(A; B), Bob(B; C), and Carol(C; D), without data constraints for simplicity. States $q_{1(23)}$ and $q_{(12)3}$ abbreviate $(q_1, (q_2, q_3))$ and $((q_1, q_2), q_3)$.

$\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ by nonassociativity. In that case, instead of denoting the protocol subprogram subsequently generated in the fourth step by $\mathtt{b}_1$ `(·)` $\mathtt{b}_2$ `(·)` $\mathtt{b}_3$ (as suggested by the notation in Figure 5.1), perhaps I should denote this subprogram more precisely by $\mathtt{b}_1$ `(·)` $\left[\mathtt{b}_2 \texttt{ (·) } \mathtt{b}_3\right]$. After all, this notation explicitly shows that the protocol units defined by $\mathtt{b}_2$ and $\mathtt{b}_3$ must always communicate first with each other—and reach a local consensus (i.e., multiply $\boldsymbol{b}_2$ and $\boldsymbol{b}_3$)—before any of them can communicate with the protocol unit defined by $\mathtt{b}_1$. Generally, such fixed communication orders deteriorate performance.

Interestingly, *strictly speaking*, one may apply the same reasoning to `(x)`, even though $\otimes$ satisfies associativity. First, one may argue that although I do not write parentheses in $\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ out of notational convenience (and because `(x)`'s associativity and commutativity make the placement of parentheses immaterial under behavioral congruence), *strictly speaking*, because $\otimes$ takes two multiplicands by Definition 29, that expression nevertheless has parentheses somewhere. Subsequently, one may argue that *strictly speaking*, I should denote the corresponding protocol subprogram either by $\mathtt{b}_1$ `(x)` $\left[\mathtt{b}_2 \texttt{ (x) } \mathtt{b}_3\right]$ or by $\left[\mathtt{b}_1 \texttt{ (x) } \mathtt{b}_2\right]$ `(x)` $\mathtt{b}_3$. Finally, one may conclude that *strictly speaking*, also the Distributed Approach fixes a communication order at compile-time, which protocol units should abide by at run-time. *Intuitively speaking*, however, such strictness makes little sense: $\otimes$ satisfies associativity, therefore the order of applying multiplications on constraint automata does not matter, therefore the order in which protocol units in the Distributed Approach communicate with each other does not matter.

The previous mismatch results from the lack in formal precision about how the protocol units defined by $\mathtt{b}_1$ `(x)` $\mathtt{b}_2$ `(x)` $\mathtt{b}_3$ should behave. Under a *strict perspective*, protocol units must respect not only the behavior of $\boldsymbol{b}_1 \otimes \boldsymbol{b}_2 \otimes \boldsymbol{b}_3$ but also this expression's structure (i.e., the hidden placement of parentheses). For performance reasons, however, I prefer a *loose perspective*: as long as protocol units respect the behavior of an expression, whether or not their communication order respects the structure of the expression should not matter.

(Perhaps I seem to diverge from my original goal at this point, dwelling on associativity in the Distributed Approach instead of dealing with nonassociativity in the Hybrid Approach. Shortly, however, I use the technique presented in the former context to give a solution for the problem in the latter context.)

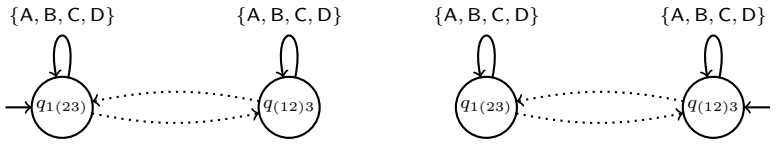


Figure 5.5: Behaviorally equivalent constraint automata to $\boldsymbol{a}_1 \otimes (\boldsymbol{a}_2 \otimes \boldsymbol{a}_3)$ and $(\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \otimes \boldsymbol{a}_3$ in Figure 5.4. Dotted arrows represent internal transitions.

To make the two different perspectives more concrete, recall the `AliceBob-CarolDave` example in Figure 4.7. As before, I anthropomorphize the protocol units in this example as Alice, Bob, and Carol (Dave plays no role here). Figure 5.4 shows two g-products of $\mathsf{Alice}(\mathsf{A};\mathsf{B})$, $\mathsf{Bob}(\mathsf{B};\mathsf{C})$, and $\mathsf{Carol}(\mathsf{C};\mathsf{D})$. These g-products differ primarily in the structure of their states, which reflects the placement of parentheses in their expressions. For Alice, Bob, and Carol to behave correctly under the strict perspective, depending on the placement of parentheses at compile-time, they must behave either as the right-associative or as the left-associative g-product in Figure 5.4 at run-time. The loose perspective, in contrast, allows Alice, Bob, and Carol to behave as *any* behaviorally equivalent—not necessarily behaviorally congruent—constraint automaton at run-time. For instance, they may behave as the constraint automata in Figure 5.5. In that case, whenever Alice, Bob, and Carol fire an internal transition (cf. silent transitions in process calculi), they effectively change their communication order: in $q_{1(23)}$, Bob and Carol must go first, while in $q_{(12)3}$, Alice and Bob must go first. In other words, if Alice started communicating with Bob to effectuate their previous instance of interaction, but if now Bob (instead of Alice) starts communicating with Carol (instead of Bob) to effectuate their next instance of interaction, somewhere in between those two instances, Alice, Bob, and Carol must have fired a silent transition from $q_{(12)3}$ to $q_{1(23)}$.

I can extend Figure 5.5 by taking not only associativity into account but also commutativity. Doing so yields the behaviorally equivalent "huge" automaton in Figure 5.6. One can easily check that, under this automaton, Alice, Bob, and Carol can freely switch between all possible communication orders at run-time. As such, this huge automaton provides a formal justification for why, in the Distributed Approach, Alice, Bob, and Carol may dynamically change their communication order. Next, I generalize this argument with a construction of huge automata for arbitrary $\otimes$-expressions.

Let $=_{\mathrm{AC}}$ denote the smallest equivalence relation induced by the following rules (i.e., add also rules for reflexivity, symmetry, and transitivity):

$$\frac{\boldsymbol{a}_1, \boldsymbol{a}_2 \in \mathbb{A}\mathrm{UTOM}}{\boldsymbol{a}_1 \otimes \boldsymbol{a}_2 =_{\mathrm{AC}} \boldsymbol{a}_2 \otimes \boldsymbol{a}_1} \qquad \frac{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3 \in \mathbb{A}\mathrm{UTOM}}{\boldsymbol{a}_1 \otimes (\boldsymbol{a}_2 \otimes \boldsymbol{a}_3) =_{\mathrm{AC}} (\boldsymbol{a}_1 \otimes \boldsymbol{a}_2) \otimes \boldsymbol{a}_3}$$

Essentially, $=_{\mathrm{AC}}$ denotes equality up-to associativity and commutativity. Given this equality, let $\mathbb{A}\mathrm{UTOM}/=_{\mathrm{AC}}$ denote the quotient set of $\mathbb{A}\mathrm{UTOM}$ under $=_{\mathrm{AC}}$. Because $\otimes$ satisfies associativity and commutativity up-to behavioral congruence, every equivalence class in $\mathbb{A}\mathrm{UTOM}/=_{\mathrm{AC}}$ contains only behaviorally con-

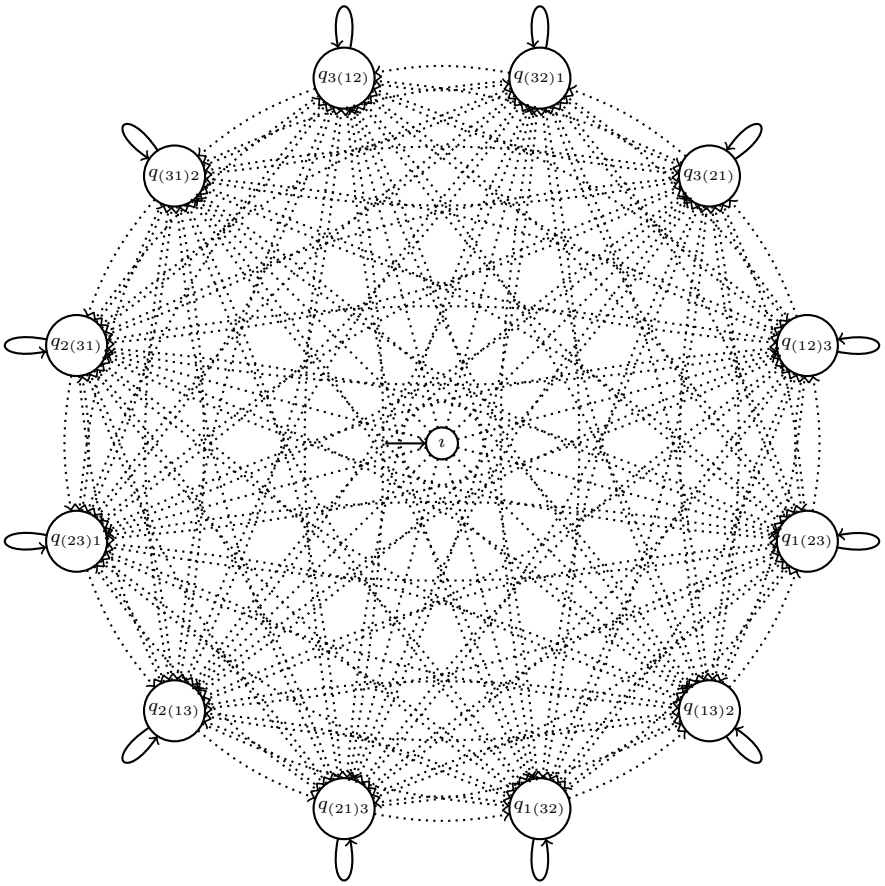


Figure 5.6: Huge automaton for Alice(A; B), Bob(B; C), and Carol(C; D), without data constraints for simplicity. Dotted arrows represent internal transitions; continuous arrows represent {A, B, C, D}-labeled transitions.

gruent constraint automata. Let $\boldsymbol{a}$ denote a constraint automaton, and let $A \in \mathbb{A}\text{UTOM}/=_{\text{AC}}$ denote its equivalence class. To construct a huge automaton for $\boldsymbol{a}$, first, I take the union of the constraint automata in A. This means that I take the union of their state spaces and transition relations and, because all constraint automata in A have the same ports, memory cells, and initial configuration, I simply copy those elements from any one of them into the huge automaton under construction. Second, I add a fresh initial state q^0 and connect this state to the (former) initial state of every $\boldsymbol{a} \in A$ with a $(\emptyset, \mathtt{K}(M))$-labeled transition. Such internal transitions have $\mathtt{K}(M)$ as their data constraint to ensure that their firing does not nondeterministically change the content of memory cells. Finally, I add a $(\emptyset, \mathtt{K}(M))$-labeled transition between all behaviorally congruent states, in both directions.

More formally, let $\mathcal{R}$ denote the largest set of relations such that $R \in \mathcal{R}$ implies both $\boldsymbol{a}_1 \preceq^{R} \boldsymbol{a}_2$ and $\boldsymbol{a}_2 \preceq^{R^{-1}} \boldsymbol{a}_1$ for some $\boldsymbol{a}_1, \boldsymbol{a}_2 \in A$ (cf. behavioral congruence in Definition 25).

$$\begin{aligned} Q \quad &= \{q^0\} \cup \bigcup\{\mathsf{Stat}(\boldsymbol{a}') \mid \boldsymbol{a}' \in A\} \\ \longrightarrow &= \bigcup\{\mathsf{Trans}(\boldsymbol{a}') \mid \boldsymbol{a}' \in A\} \\ &\quad \cup \{(q^0, \emptyset, \mathtt{K}(M), q') \mid \boldsymbol{a}' \in A \textbf{ and } \mathsf{init}(\boldsymbol{a}') = (q', \mu')\} \\ &\quad \cup \{(q, \emptyset, \mathtt{K}(M), q') \mid q \; R \; q' \textbf{ and } R \in \mathcal{R}\} \end{aligned}$$

I conjecture that on input of an arbitrary constraint automaton, "hugeification" so defined yields a behaviorally equivalent, yet behaviorally incongruent (because of its internal transitions), constraint automaton. After all, the added internal transitions by themselves have no effect on the accepted interaction language, and because these transitions connect only behaviorally congruent—hence, behaviorally equivalent by Theorem 1—states, a huge automaton has equivalent options for its behavior before and after firing these transitions.

The previous construction of huge automata essentially shows that protocol units in the Distributed Approach may dynamically change their communication order: under the loose perspective, protocol units collectively simulate not a constraint automaton that strictly reflects the placement of parentheses but a huge, behaviorally equivalent one. As such, hugeification—and in particular the definition of $=_{\mathrm{AC}}$—makes the importance of $\otimes$'s associativity formally precise: it (together with commutativity) characterizes sets of behaviorally equivalent constraint automata, each of which reflects a different communication order, between which protocol units may freely switch at run-time. Of course, no compiler should ever actually construct huge automata; I use hugeification only as a means for formal reasoning.

L-multiplication, in contrast to g-multiplication, does not satisfy associativity. Hence, I cannot directly apply the previous hugeification technique in the Hybrid Approach. Instead, I first need to establish under which conditions—if any—$\odot$ does satisfy associativity. I present such conditions in three lemmas. The first lemma states that if a constraint automaton $\boldsymbol{a}_2$ has enslaved a constraint automaton $\boldsymbol{a}_1$, neither of which depends on a constraint automaton $\boldsymbol{a}_3$, l-multiplication satisfies associativity.

Lemma 13.

$$\begin{bmatrix} \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3), (\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot \boldsymbol{a}_3 \in \mathbb{A}\mathrm{UTOM} \\ \textbf{and } \boldsymbol{a}_1, \boldsymbol{a}_2 \asymp \boldsymbol{a}_3 \textbf{ and } \boldsymbol{a}_1 \bowtie \boldsymbol{a}_2 \end{bmatrix}$$
$$\textbf{implies } \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3) \simeq (\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot$$

The second and the third lemma do not state associativity per se, but they state important properties otherwise implied by associativity (together with commutativity). The second lemma states that if both $\boldsymbol{a}_2$ and $\boldsymbol{a}_3$ have conditionally enslaved $\boldsymbol{a}_1$, while both $\boldsymbol{a}_1$ and $\boldsymbol{a}_3$ have conditionally enslaved $\boldsymbol{a}_2$ (i.e., $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ have enslaved each other), I can "swap" $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$. The third lemma has the same consequence but a different premise, namely that $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, and $\boldsymbol{a}_3$ have no dependencies between each other.

$$
\begin{aligned}
& \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_6 \odot (\boldsymbol{a}_7 \odot \boldsymbol{a}_8)))))) \\
\simeq\ & a_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & a_1 \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & a_1 \odot (a_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & a_1 \odot (a_3 \odot (a_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_7 \odot (\boldsymbol{a}_6 \odot a_8)))))) \\
\simeq\ & \boldsymbol{a}_3 \odot (\boldsymbol{a}_1 \odot (a_4 \odot (a_5 \odot (a_2 \odot (a_7 \odot (a_6 \odot a_8)))))) \\
\simeq\ & a_3 \odot (\boldsymbol{a}_4 \odot (\boldsymbol{a}_1 \odot (a_5 \odot (a_2 \odot (a_7 \odot (a_6 \odot a_8)))))) \\
\simeq\ & a_3 \odot (a_4 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_1 \odot (a_2 \odot (a_7 \odot (a_6 \odot a_8)))))) \\
\simeq\ & a_3 \odot (a_4 \odot (a_5 \odot ((\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_7)) \odot (a_6 \odot a_8)))) \\
\simeq\ & a_3 \odot (a_4 \odot (a_5 \odot (\boldsymbol{a}_6 \odot ((\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_7)) \odot a_8)))) \\
\simeq\ & a_3 \odot (\boldsymbol{a}_5 \odot (\boldsymbol{a}_4 \odot (a_6 \odot ((a_1 \odot (a_2 \odot a_7)) \odot a_8)))) \\
\simeq\ & a_3 \odot (a_5 \odot ((\boldsymbol{a}_4 \odot \boldsymbol{a}_6) \odot ((a_1 \odot (a_2 \odot a_7)) \odot a_8))) \\
\simeq\ & a_3 \odot (a_5 \odot ((\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_7)) \odot ((\boldsymbol{a}_4 \odot \boldsymbol{a}_6) \odot a_8))) \\
\simeq\ & a_3 \odot (a_5 \odot (\boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot (\boldsymbol{a}_7 \odot ((a_4 \odot a_6) \odot a_8)))))
\end{aligned}
$$

$\boldsymbol{a}_7$ acquires $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$

$\boldsymbol{a}_6$ acquires $\boldsymbol{a}_4$

$\boldsymbol{a}_7$ releases $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$

Figure 5.7: Example reordering of a $\odot$-expression over slaves $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, $\boldsymbol{a}_3$, $\boldsymbol{a}_4$, and $\boldsymbol{a}_5$ and masters $\boldsymbol{a}_6$, $\boldsymbol{a}_7$, and $\boldsymbol{a}_8$

Lemma 14.

$$
\begin{bmatrix} \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3), \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3) \in \mathbb{AUTOM} \\ \textbf{and}\ \ \boldsymbol{a}_1 \bowtie \boldsymbol{a}_2, \boldsymbol{a}_3\ \ \textbf{and}\ \ \boldsymbol{a}_2 \bowtie \boldsymbol{a}_1, \boldsymbol{a}_3 \end{bmatrix}
$$
$$
\textbf{implies}\ \ \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3) \simeq \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3)
$$

Lemma 15.

$$
\begin{bmatrix} \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3), \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3) \in \mathbb{AUTOM} \\ \textbf{and}\ \ \boldsymbol{a}_1 \asymp \boldsymbol{a}_2, \boldsymbol{a}_3\ \ \textbf{and}\ \ \boldsymbol{a}_2 \asymp \boldsymbol{a}_1, \boldsymbol{a}_3 \end{bmatrix}
$$
$$
\textbf{implies}\ \ \boldsymbol{a}_1 \odot (\boldsymbol{a}_2 \odot \boldsymbol{a}_3) \simeq \boldsymbol{a}_2 \odot (\boldsymbol{a}_1 \odot \boldsymbol{a}_3)
$$

Each of these three lemmas has an intuitive meaning in terms of the masters/slaves interpretation of constraint automata in the premise in Theorem 9. Lemma 14 means that I can "move" every slave "rightward" until it neighbors the leftmost master. Dually, Lemma 15 means that I can move every master "leftward" until it neighbors the rightmost slave. Finally, Lemma 13 means that I can "group" any number of rightmost slaves and any number of leftmost masters together (i.e., put parentheses around them). The actual proofs of these results go by induction on the number of slaves and the number of masters. Combined, for a given master $\boldsymbol{a}_j$, I can use these lemmas to move $\boldsymbol{a}_j$ leftward, move all $\boldsymbol{a}_j$'s slaves rightward, and finally group $\boldsymbol{a}_j$ and its slaves such that, within this group, $\boldsymbol{a}_j$ constitutes a most deeply nested l-multiplication (several of which may exist). Figure 5.7 exemplifies such reordering.

Every reordering of a $\odot$-expression as in the premise in Theorem 9 transforms that expression into another expression. By Lemmas 13, 14, and 15, the l-products to which these $\odot$-expressions evaluate live in the same equivalence class (under behavioral congruence). Similarly, $\otimes$-expressions up-to associativity live in the same equivalence class (under behavioral congruence). Thus, by replacing the rule for associativity in the definition of $=_{\mathrm{AC}}$ with a rule for *conditional associativity* as stated in Lemmas 13, 14, and 15, I can construct a new equivalence relation $=_{\mathrm{cAC}}$, take the quotient $\mathbb{A}\mathrm{UTOM}/=_{\mathrm{cAC}}$, and hugeify in the same way as before. Every added internal transition in a huge automaton so constructed models a reordering. Because constraint automata in the same equivalence class under $=_{\mathrm{cAC}}$ have congruent behavior—hence, equivalent behavior by Theorem 1—the same argument for the correctness of hugeification applies as before.

The previous construction of huge automata essentially shows that only protocol units *for masters* in the Hybrid Approach may start communicating with their slaves at run-time. Moreover, by Definition 35 of slavery, none of these slaves need to communicate with protocol units other than their masters (i.e., each of their transitions that involves a shared port with a master involves only shared ports with that master). It does not matter which particular master among all masters goes first; expression reordering can occur for any master at any time. In fact, even multiple masters (with disjoint sets of slaves) may start communicating simultaneously. In that case, the corresponding expression just has multiple most-deeply nested multiplications (e.g., $(\boldsymbol{a}_1 \odot \boldsymbol{a}_2) \odot (\boldsymbol{a}_3 \odot (\boldsymbol{a}_4 \odot \boldsymbol{a}_5))$). Figures 5.8 and 5.9 show simplified event-handlers for a protocol unit that simulates a master/slave (cf. Figures 4.5 and 4.6). I do not intend these figures to convey real "algorithms"; they serve just as a stylized description of what event-handling roughly entails in the Hybrid Approach.

Nonassociative parallel composition operators (such as $\odot$) occur also in the literature on concurrency theory, where authors usually consider such operators defective. For instance, Vrancken set out to improve an earlier version of the Algebra of Communicating Processes (with the empty process), because the merge operator in that version "turned out not associative" [Vra97]; Baeten and Van Glabbeek acknowledge that "this problem [a nonassociative merge operator] was remedied" by Vrancken [BvG87]. In the context of timed automata with shared variables and action synchronization, Berendsen and Vaandrager point out that "the approach [to support shared variables and action synchronization] in [6] [sic] is flawed since parallel composition is not associative" [BV08]. Berendsen and Vaandrager also states that "commutativity and associativity are highly desirable properties for parallel composition operators" [BV08]. Anantharaman et al., in turn, consider a process algebra with a nonassociative synchronous composition operator, but they subsequently characterize a class of processes for which this operator actually exhibits associativity and work only with processes from that class [ACH05]. Segala discusses problems of defining a parallel composition operator for general probabilistic automata, symptomized by nonassociativity [Seg95]. Finally, Klin and Sassone notice that parallel composition in stochastic π-calculus generally fails

Input: a port p on which an event occurred, a *context* $P^{\text{ctxt}} \subseteq P^{\text{in}} \cup P^{\text{out}}$ of global boundary ports with a pending I/O operation, and the current local state q_i of $\boldsymbol{b}_i$
Output: q_i' holds the next local state of $\boldsymbol{b}_i$.
Effect: either, through the firing of enabled local transitions (including a local transition of $\boldsymbol{b}_i$), an enabled global transition fires (if the I/O operations pending on the ports in P^{ctxt} satisfy that transition's label), or all global transitions are disabled (otherwise).

1. Wake up, and assign q_i to q_i'.
2. Assign $\emptyset$ to Φ, a variable for a set of data constraints.
3. For all transitions $q_i \xrightarrow{P_i,\phi_i}_i q_i'$, ordered nondeterministically:
   (a) If $p \notin P_i$, continue (i.e., skip to the next iteration).
   (b) If $P_i \cap (P^{\text{in}} \cup P^{\text{out}}) \not\subseteq P^{\text{ctxt}}$ (i.e., not all boundary ports involved in the current local transition have a pending I/O operation), continue.
   (c) Assign $\{\phi\}$ to Φ', a variable for a set of data constraints.
   (d) For all ports $p' \in P_i \setminus (P^{\text{in}} \cup P^{\text{out}})$:
      i. Send a message to the protocol unit that shares access to p' to ask which data constraints must hold for that unit to fire a transition involving p'.
      ii. Await an answer message Φ'' from that protocol unit.
      iii. Assign $\{\phi' \wedge \phi'' \mid \phi' \in \Phi' \textbf{ and } \phi'' \in \Phi''\}$ to Φ'.
   (e) Assign $\Phi \cup \Phi'$ to Φ.
4. For all data constraints $\phi \in \Phi$:
   (a) Compute a data assignment σ that respects the pending I/O operations and satisfies ϕ; continue if no such σ exists.
   (b) Distribute data among local ports and memory cells according to σ.
   (c) Send σ to all protocol units sent messages to in Step 3.
   (d) Compute a q' such that $q_i \xrightarrow{P_i,\phi_i} q'$ and $P_i \subseteq \mathrm{Dom}(\sigma)$ and $\sigma \models \phi_i$.
   (e) Assign q' to q_i', and abort the loop.
5. If the previous loop never made it to Step 4-e, send $\emptyset$ (i.e., the empty data assignment) to all protocol units sent messages to in Step 3.
6. Go dormant.

Figure 5.8: Simplified p-event-handler for a protocol unit that simulates a master medium automaton $\boldsymbol{b}_i = (Q_i, (P_i^{\text{all}}, P_i^{\text{in}}, P_i^{\text{out}}), M_i, \longrightarrow_i, (q_i^0, \mu_i^0))$ in the Hybrid Approach, where P^{in} and P^{out} denote the sets of global input and output ports

to exhibit associativity and investigate under which conditions it does [KS08]. However, as with $\odot$ in this thesis, nonassociativity does not always pose problems. For instance, in the context of reachability analysis, Yeh investigates a state space reduction technique for processes by adding distinguished actions for suspending and resuming processes; in the resulting theory, parallel composition does not exhibit associativity [Yeh93]. As another example, Kuske &

Input: a port p on which an event occurred, a *context* $P^{\text{ctxt}} \subseteq P^{\text{in}} \cup P^{\text{out}}$ of global boundary ports with a pending I/O operation, and the current local state q_i of $\boldsymbol{b}_i$
Output: q_i' holds the next local state of $\boldsymbol{b}_i$.
Effect: either, through the firing of enabled local transitions (including a local transition of $\boldsymbol{b}_i$), an enabled global transition fires (if the I/O operations pending on the ports in P^{ctxt} satisfy that transition's label), or all global transitions are disabled (otherwise).

1. Wake up, and assign q_i to q_i'.
2. Assign $\emptyset$ to Φ, a variable for a set of data constraints.
3. For all transitions $q_i \xrightarrow{P_i,\phi_i}_i q_i'$, ordered nondeterministically:
   (a) If $p \notin P_i$, continue (i.e., skip to the next iteration).
   (b) If $P_i \cap (P^{\text{in}} \cup P^{\text{out}}) \not\subseteq P^{\text{ctxt}}$ (i.e., not all boundary ports involved in the current local transition have a pending I/O operation), continue.
   (c) Assign $\Phi \cup \{\phi\}$ to Φ.
4. Send an answer message Φ to the protocol unit from which the p-event originated.
5. Await a message with a data assignment σ.
6. If $\sigma \neq \emptyset$, distribute data among local ports and memory cells according to σ.
7. If $\sigma \neq \emptyset$, compute a q' such that $q_i \xrightarrow{P_i,\phi_i} q'$ and $P_i \subseteq \mathrm{Dom}(\sigma)$ and $\sigma \models \phi_i$.
8. If $\sigma \neq \emptyset$, assign q' to q_i'.
9. Go dormant.

Figure 5.9: Simplified p-event-handler for a protocol unit that simulates a slave medium automaton $\boldsymbol{b}_i = (Q_i, (P_i^{\text{all}}, P_i^{\text{in}}, P_i^{\text{out}}), M_i, \longrightarrow_i, (q_i^0, \mu_i^0))$ in the Hybrid Approach, where P^{in} and P^{out} denote the sets of global input and output ports

Meinecke introduce a nonassociative product operator on branching automata with costs [KM03]. Finally, the Orc orchestration language has three combinators to express parallel execution, two of which exhibit neither associativity nor commutativity [KQCM09].

L-multiplication as introduced in this chapter differs from previous nonassociative parallel composition operators in the sense that even though it fails to exhibit associativity in general, it exhibits associativity *in all relevant cases* (Lemmas 13–15). As such, l-multiplication's nonassociativity does not render its definition defective. On the contrary: l-multiplication's nonassociativity essentially reflects the inherent asymmetry between masters and their slaves and, consequently, constitutes a *feature*, not a *bug*.

5.2 Practice

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

Compiler

I extended Lykos with the ability to generate code under the Hybrid Approach, controllable through flag `PARTITION`. When raised, Lykos partitions the set of small automata as described in Section 5.1. Consequently, instead of generating only a single protocol subprogram (as in Chapter 4), Lykos generates multiple protocol subprograms, one for every previously computed medium automaton. Each of these protocol subprograms defines a protocol unit, and every such a protocol unit corresponds to either a master or a slave.

Recall from Chapter 4 that every port data structure at run-time has two users: a protocol unit and either another protocol unit or a worker unit. In the latter case, if a port data structure has a protocol unit and a worker unit as its users, I call this protocol unit "on the boundary". During compilation, to ease code generation, Lykos ensures that every protocol unit on the boundary corresponds to a master. The technique to do this consists, essentially, of adding for every small automaton that satisfies no-synchronization an extra Sync "before" each of its input ports and an extra Sync "after" each of its output ports (where "input" and "output" qualify ports from the protocol perspective). For instance, Lykos may replace Fifo$(p_1; p_4)$ with Sync$(p_1; p_2)$, Fifo$(p_2; p_3)$, and Sync$(p_3; p_4)$. First, observe that Fifo$(p_1; p_4)$ and the product of Sync$(p_1; p_2)$, Fifo$(p_2; p_3)$, and Sync$(p_3; p_4)$ have equivalent behavior. Indeed, Sync forms some kind of neutral element for multiplication; I come back to this point in Chapter 6. Thus, Lykos does not affect the original behavior of a set of small automata by adding Syncs as just described. Second, observe that because Fifo satisfies no-synchronization, it ends up in its own subset in the partition and, by itself, constitutes one medium automaton—a slave. At the same time, Sync$(p_1; p_2)$ and Sync$(p_3; p_4)$ share no ports and, thus, satisfy independence. Consequently, also each of these two Syncs ends up in its own subset in the partition and, by itself, constitutes a medium automaton—a master. By adding Syncs for all small automata that satisfy no-synchronization in this way, Lykos ensures that only protocol units for masters lie on the boundary.

As explained in Chapter 4, worker threads execute not only computation code but also interaction code, on behalf of protocol units for masters that lie on the boundary. In contrast to the situation in Chapter 4, however, every protocol unit for a master—including those on the boundary—additionally has its own thread. Usually, these protocol threads lie dormant. Only after a neighboring protocol unit (for a slave) has made a transition, a protocol thread awakes and starts a new round of event-handling. After all, if this neighbor has changed state during its previous transition, in its new current state, it may have different outgoing transitions, with different synchronization constraints, than before. Consequently, this neighbor may now agree to involve shared ports in transitions that it could not agree to before, in its previous state (where it had different outgoing transitions, with different synchronization constraints). All threads ensure that they execute event-handlers in mutually exclusive fashion, to avoid race conditions.

While every protocol unit for a master has its own thread, protocol units

for slaves do not have their own thread. Instead, threads that execute code (on behalf) of protocol units for masters also execute code on behalf of protocol units for slaves, in the same way that worker threads execute code on behalf of protocol units. In other words, the same thread interleaves the execution of Figure 5.8 (i.e., the protocol unit for a master) with the execution of Figure 5.9 (i.e., the protocol units for that master's slaves), without going to sleep, awaking other threads, or explicitly sending messages—the thread does all the work itself. As a consequence, protocol units for slaves essentially degenerate into purely passive entities—data structures—at run-time. For instance, the protocol unit for a Fifo effectively consists just of a variable (to hold the content of the buffer) and a lock (to guarantee mutual exclusion). Whenever a thread fires a transition of such a passive protocol unit for a slave, this thread also notifies all neighboring protocol units (for masters) of a possible state change, as already discussed above.

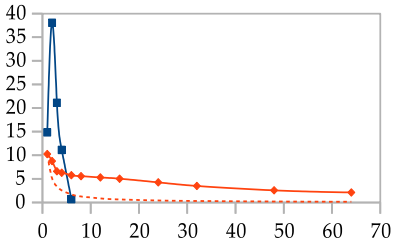
As another, minor optimization, Lykos carries out static analysis at compile-time in an attempt to safely *predict* whether a protocol unit for a master can reach consensus with its neighboring protocol units (for slaves) at run-time. This works as follows. Suppose that a master and its slave share a port p (and nobody else knows about p). Moreover, suppose that the slave has only one transition (q, P, ϕ, q') involving p (i.e., $p \in P$). Then, at compile-time, Lykos can establish two facts about the situation at run-time in which the master successfully fires a local transition involving p: (i) its slave necessarily has q as its current state, and (ii) the computed data assignment satisfies ϕ. Given those facts, by adding ϕ already to the data constraint of every transition involving p in the master at compile-time, communication and composition of data constraints at run-time becomes unnecessary. Such manipulation at compile-time does not strengthen the original data constraints too much, because the corresponding transitions can fire *only together* with transition (q, P, ϕ, q'), in the master's slave anyway. Moreover, if the compiler manipulates data constraints in this way already at compile-time, to see if the master and its slave can agree at run-time, a simple check for whether the slave has q as its current state suffices. This further reduces the overhead of the already cheap consensus algorithm necessary in the Hybrid Approach.

I skip code examples in this chapter, because the run-time library and the compiler-generated code remain largely the same as what I showed already in Chapter 4.

Experiments I: Protocols

I repeated the same experiments as in Chapter 4, generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the `PARTITION`-flag raised, but otherwise under the same conditions as in Chapter 4. Figure 5.10 shows per-family performance charts, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. Recall from Chapter 4 that inverse-proportional growth forms

—same as in Figure 4.26—



SyncK

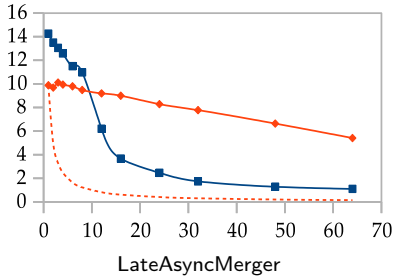
FifoK

—same as in Figure 4.26—

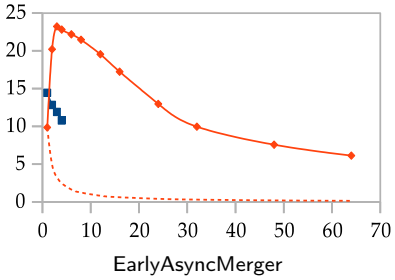
—same as in Figure 4.26—

Merger

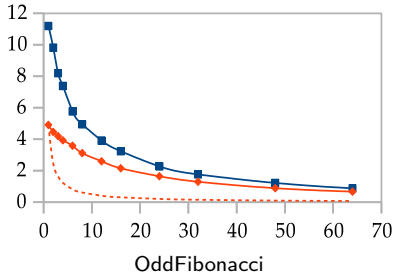
Router



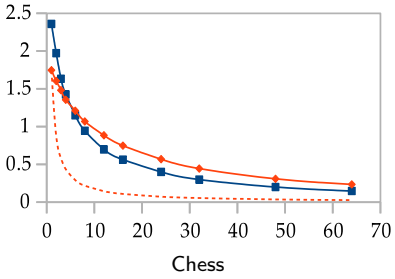
LateAsyncMerger



EarlyAsyncMerger



OddFibonacci



Chess

Figure 5.10: 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.

a necessary condition (but not necessarily a sufficient one) for good scalability. The red lines represent the new results; the blue lines represent the results from Chapter 4. For SyncK, Merger, and Router, Lykos generated exactly the same code as in Chapter 4. In these cases, the Hybrid Approach degenerates into the Centralized Approach.

Figure 5.11 shows per-family speedup charts corresponding to the measurements in Figure 5.10; the dotted lines represent equal performance. For FifoK, the previous scalability problems, both at compile-time and at run-time, have disappeared: at compile-time, Lykos (with the `PARTITION`-flag raised) succeeded in generating code for all values of k without exhausting its available resources, while at run-time, the performance of the generated code stays above the critical threshold of inverse-proportionality. For EarlyAsyncMerger, the same observations hold true. In these cases, thus, the Hybrid Approach indeed solves the previous scalability problems. For LateAsyncMerger and Chess, performance has also improved to greater and to lesser extent, indicating that the recovery of useful parallelism as achieved in the Hybrid Approach can pay off. But, code generated under the Centralized Approach for members of LateAsyncMerger and Chess with smaller values of k outperforms code generated under the Hybrid Approach for those same members. This indicates that parallelism becomes more important as the number of workers increases.

Interestingly, FifoK and EarlyAsyncMerger exemplify situations where code generated under the Hybrid Approach has, in fact, *lower* latency than code generated under the Centralized Approach. This may come as a surprise, as the Hybrid Approach requires a (cheap) consensus algorithm (which nevertheless inflicts overhead and thereby generally increases latency), whereas the Centralized Approach does not. Sometimes, thus, code generated under the Centralized Approach has *another* major source of overhead, which increases latency more dramatically than the Hybrid Approach's consensus algorithm does (e.g., for members of FifoK and EarlyAsyncMerger for certain values of k). This source of overhead consists of the number of transitions that a big automaton, as computed in the Centralized Approach, may have: whenever a big automaton has many more transitions than every medium automaton in the Hybrid Approach—this happens, for instance, if that number of transitions increases exponentially in k, as with FifoK—it may take much longer for that big automaton's protocol unit to find an enabled transition than for the protocol units for those medium automata. This observation, then, constitutes another point in favor of the Hybrid Approach.

Finally, for OddFibonacci, all code generated under the Centralized Approach actually outperforms all code generated under the Hybrid Approach. Here, one witnesses *overparallelization*, where a number of parallel threads implement an inherently sequential protocol specification. For all members of OddFibonacci, the computed partition consists of seven subsets (independent of the number of consumers), subsequently resulting in seven medium automata: three masters and four slaves (i.e., Fifos). However, as shown in the constraint automaton for the $\mathsf{OddFibonacci}_2$ protocol in Figure 3.25, this protocol has no real parallelism to exploit among its workers: either all workers

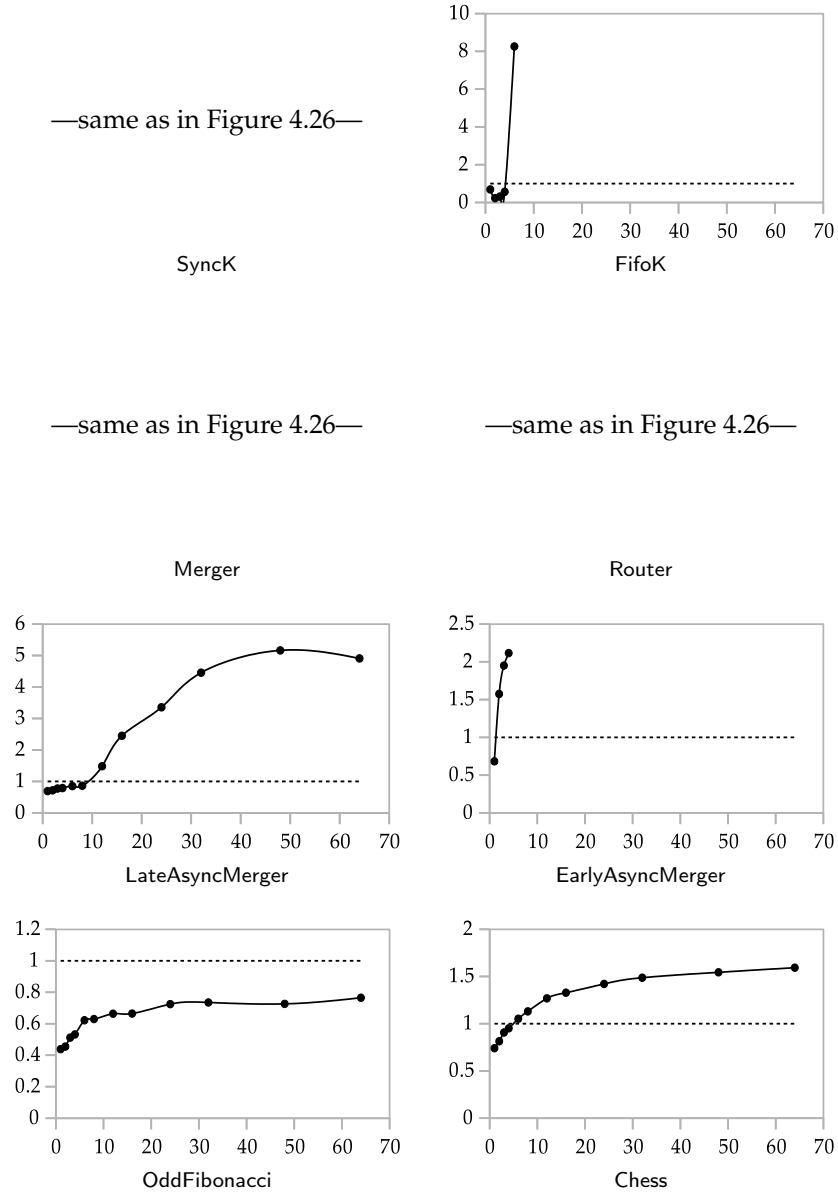


Figure 5.11: Speedup (relative to compiler-generated code in Chapter 4) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

synchronously complete their I/O operations (the left transition in Figure 3.25), or the producer completes an I/O operation just by itself (the right transition). Consequently, a parallel implementation of seven protocol units (three active ones and four passive ones) incurs the overhead that parallelism entails without gaining anything. As a result, such parallel implementations have poorer performance than sequential ones, as shown in Figures 5.10 and 5.11.

Despite their overparallelization, the seven protocol units for every OddFibonacci member exhibit "useful parallelism" according to its definition in Section 5.1, a concept that I used as a guideline for computing reasonable partitions. This suggests that to avoid overparallelization through more clever partitioning, I need to refine the definition of useful parallelism. After all, as the experiments with OddFibonacci members demonstrate, the current definition sometimes erroneously qualifies parallelism among protocol units as useful—and thereby essentially misguides the computation of reasonable partitions—while in practice, it leads to overparallelization.

To better explain the problem at hand, let a "sequence of dependent protocol units" start with one protocol unit, which depends on the next protocol unit, which, in turn, depends on the third protocol unit, and so on. If such a sequence of dependent protocol units starts and ends with the same protocol unit, and none of the intermediate protocol units lie on the boundary, the parallelism among those intermediate protocol units serves no real purpose. For instance (cf. OddFibonacci in Figures 3.26 and 3.27), suppose that the intermediate protocol units in a sequence of dependent protocol units correspond to a Fifo, followed by a Sync, followed by another Fifo. Also, suppose that the first/last protocol unit corresponds to a constraint automaton with a transition involving (at least) the input port of the first Fifo and the output port of the second Fifo. After firing this transition, the first/last protocol unit must wait for the intermediate protocol units to fire their transitions (i.e., to transport the new datum in the first buffer to the second buffer) before it can fire this transition again. The parallelism among those protocol units, therefore, has no real advantage—but, in contrast, negatively affects performance—and sequentializing this whole sequence of dependent protocol units seems the better option. In terms of the theory presented in this chapter, I can achieve such sequentialization by letting the subset for the first/last protocol unit in the sequence *gobble up* the subsets for the intermediate protocol units. Although I have a fair understanding of the theory involved in this proposal, the actual practical consequences remain unclear to me and require further investigation. In particular, one possible adverse side-effect that I foresee consists of the reintroduction of state space explosion (e.g., if the sequence of dependent protocol units in the previous example consists of 64 Fifos, merging their corresponding subsets in the partition eventually gives rise to a constraint automaton with 2^{64} states). I leave a thorough study of this topic for future work.

	NPB-FT	NPB-MG	NPB-CG	NPB-IS	NPB-BT	NPB-SP	NPB-LU
W	32	∞	∞	∞	22	34	31
A	128	∞	∞	∞	62	62	62
B	256	∞	∞	∞	100	100	100
C	512	∞	∞	∞	160	160	160

Figure 5.12: Maximum number of slaves in the NPB benchmarks

Experiments II: Programs

I repeated the same experiments as in Chapter 4, generating code for the NPB benchmarks with the `PARTITION`-flag raised, but otherwise under the same conditions as in Chapter 4. Using the Hybrid Approach, in contrast to the Centralized Approach as used in Chapter 4, Lykos succeeded in generating code for all values of k for all NPB kernel benchmarks, except NPB-MG, and for all but the last value of k for the three NPB application benchmarks and NPB-MG.

To facilitate a fair comparison of the performance of different implementations of NPB, the NPB documentation—available at the NASA website—defines five problem size classes, with predefined inputs, to run the benchmarks on: class S (small size, just for testing), class W (1990s workstation size), class A (1990s supercomputer size, larger than class W), class B (1990s supercomputer size, roughly four times larger than class A), and class C (1990s supercomputer size, roughly four times larger than class B). I ran every FOCAML-to-Java-compiled version of the NPB kernel benchmarks with inputs from class W, class A, class B, and class C, while I ran every FOCAML-to-Java-compiled version of the NPB application benchmarks with inputs only from class W and class A; the latter benchmarks took, in the slowest cases, already over an hour, which made further upscaling the problem size impractical. I ran these benchmarks for every value of $k \in \{2, 4, 8, 16, 32, 64\}$ for which Lykos succeeded in generating code, where k denotes the number of slaves, except if such a k exceeded the maximum number of slaves for a given benchmark/class combination in Figure 5.12 (these limits come from the Java implementation of NPB). As before, I used a machine with 24 cores (two Intel E5-2690V3 processors in two sockets), without Hyper-Threading and without Turbo Boost (i.e., with a static clock frequency).

Figures 5.13–5.20 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speed-up charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The dotted red lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks; the solid red lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represents the Java versions by Frumkin et al.

The machine on which I performed my experiments allowed me to monitor

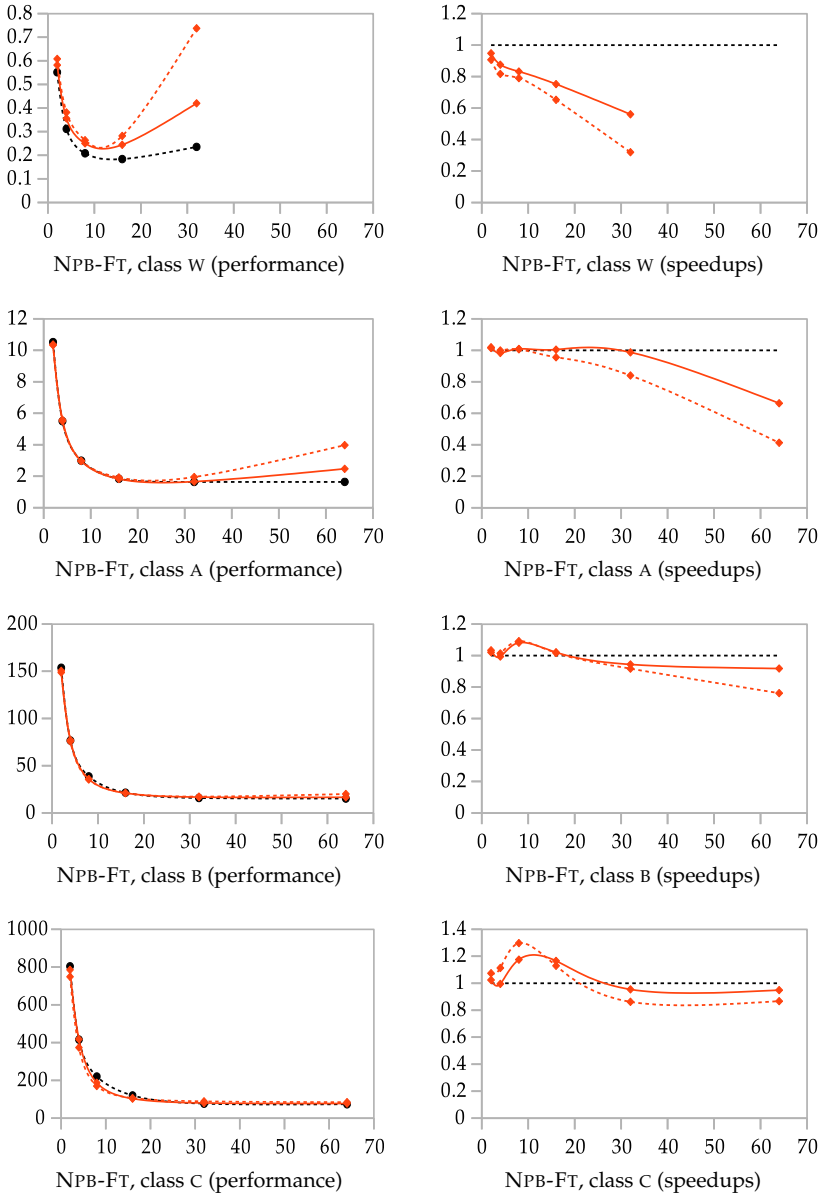


Figure 5.13: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

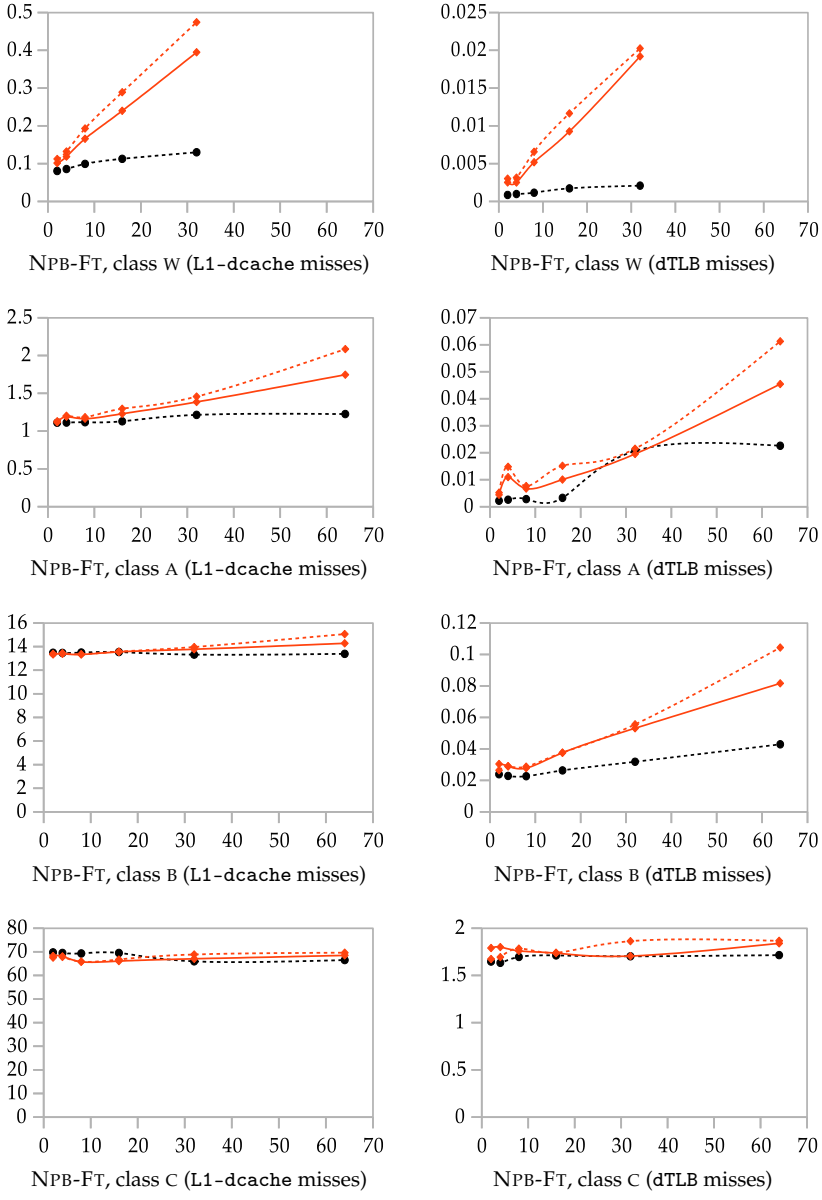


Figure 5.14: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

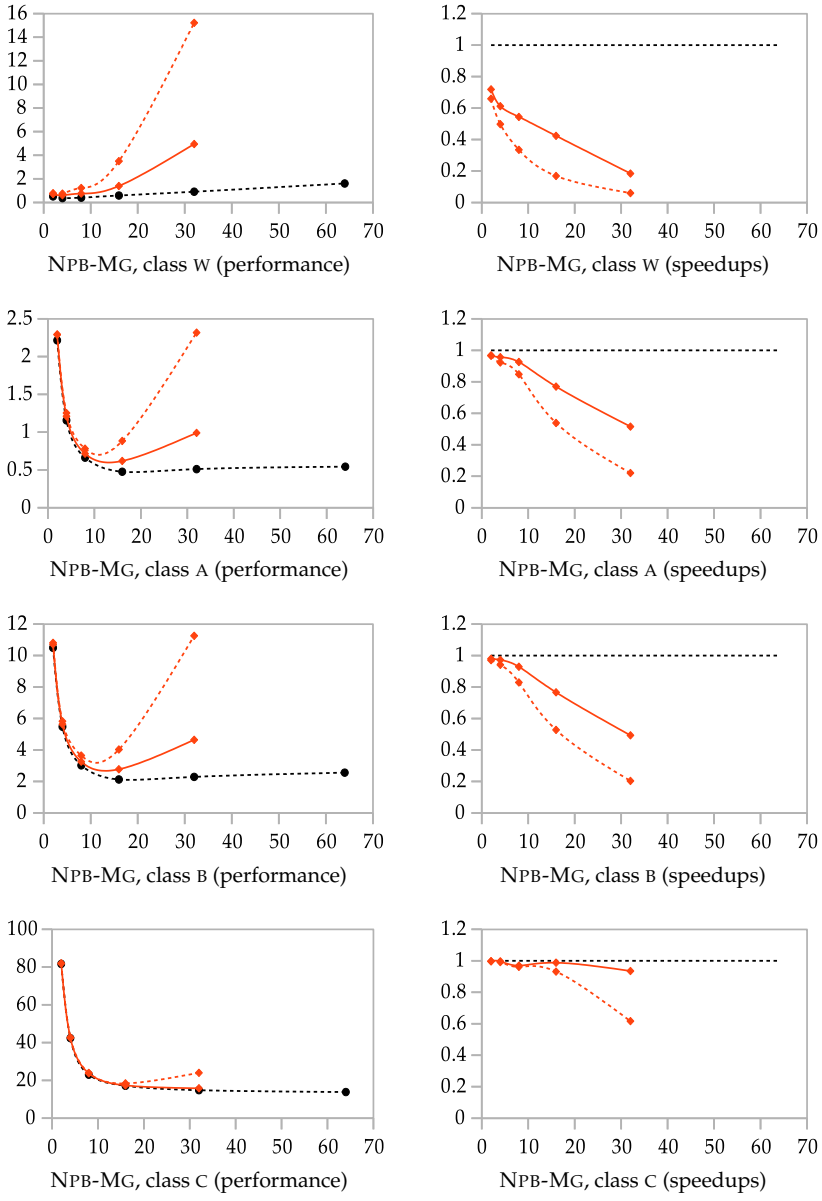


Figure 5.15: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

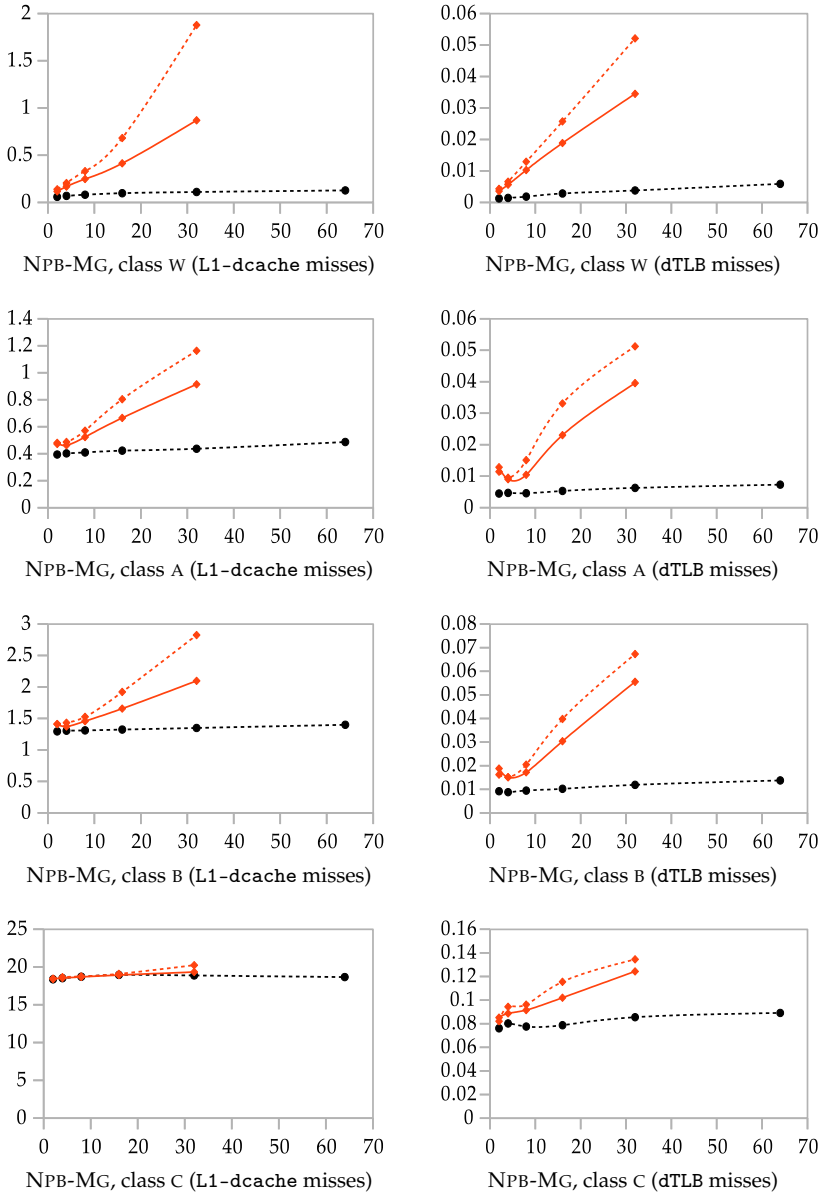


Figure 5.16: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

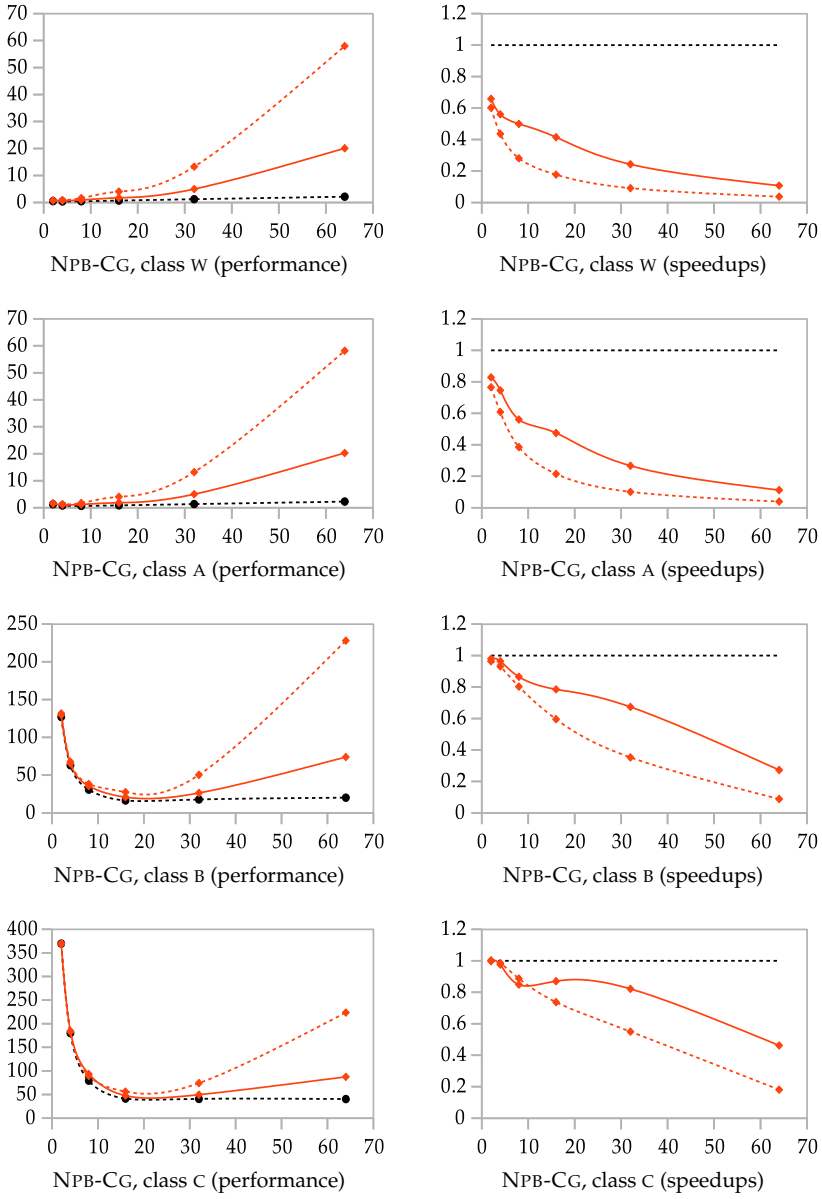


Figure 5.17: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

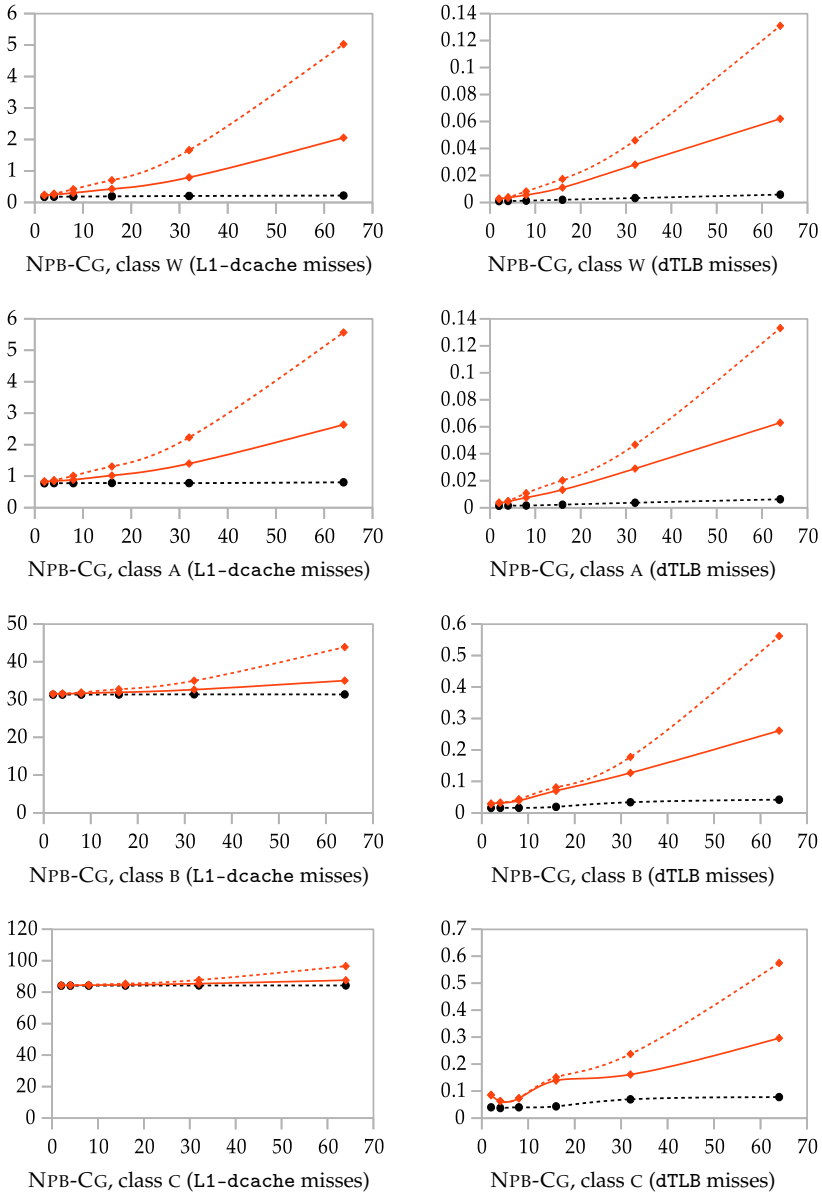


Figure 5.18: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

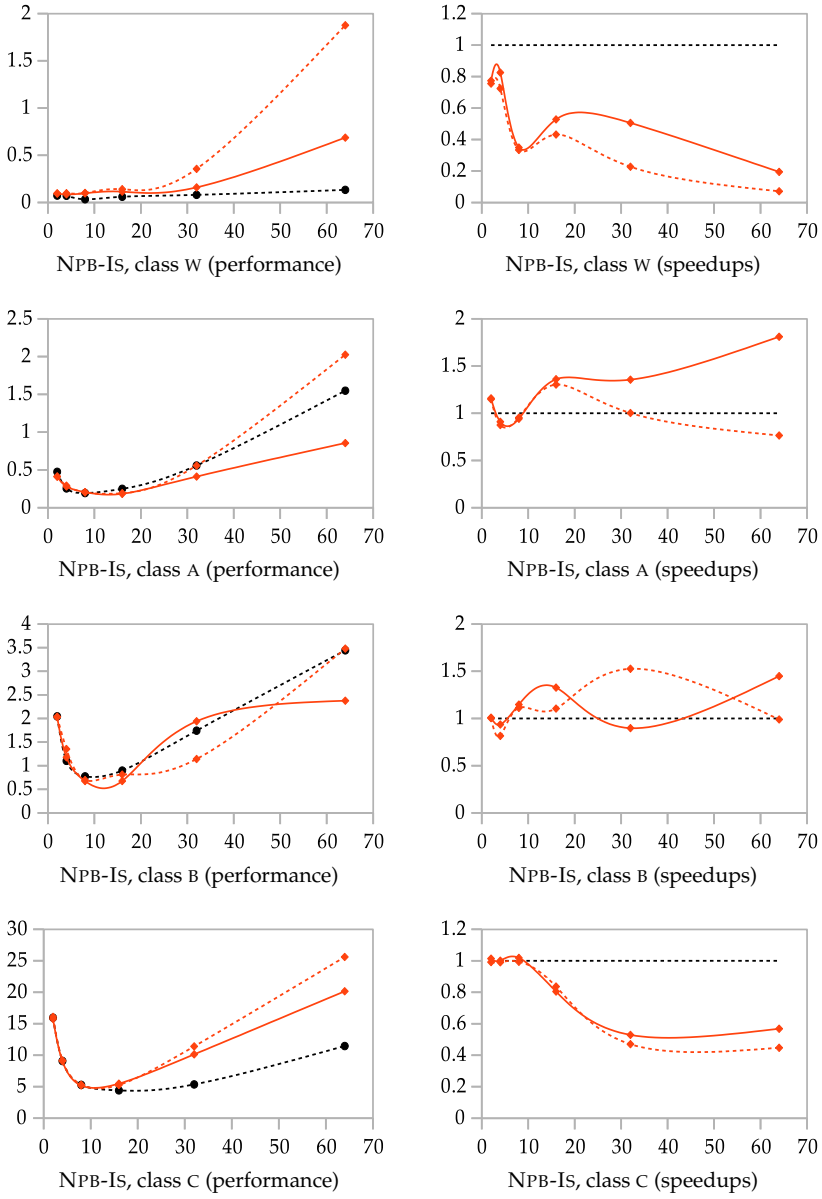


Figure 5.19: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

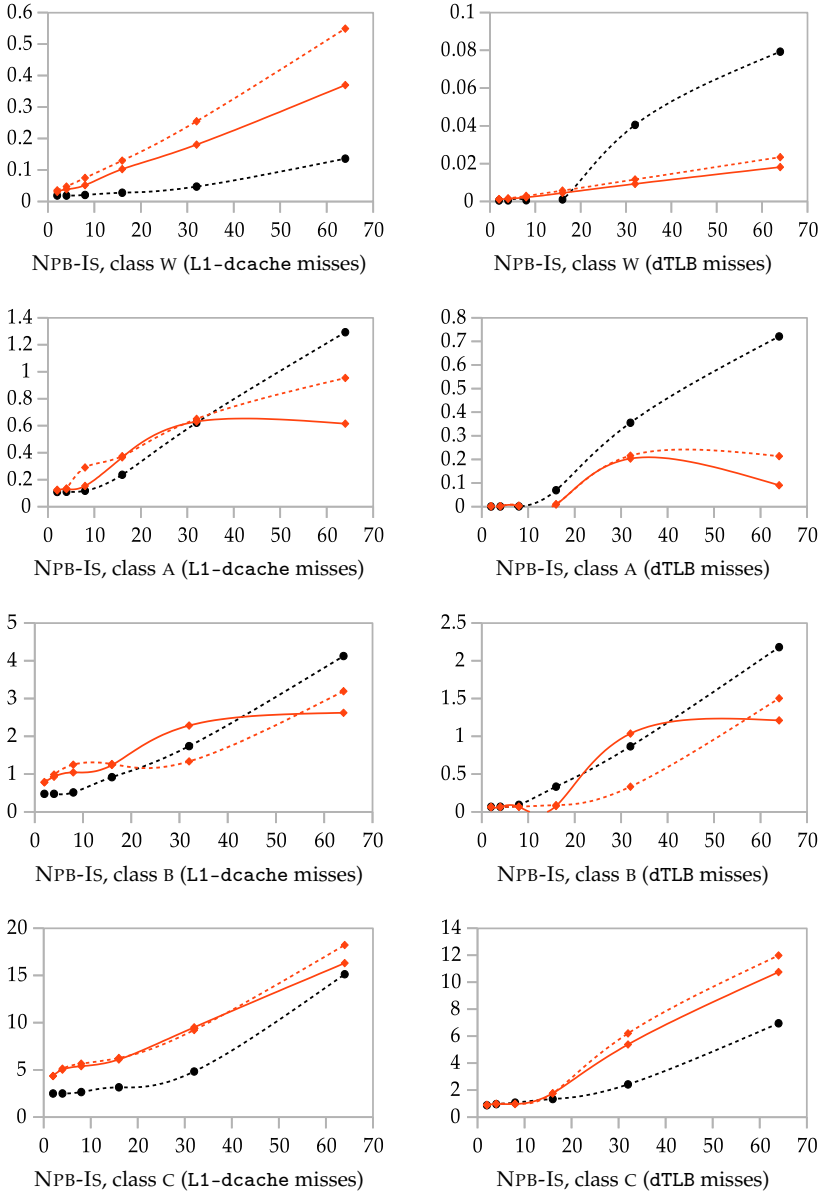


Figure 5.20: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

	cycles/miss	nanoseconds/miss	seconds/billion misses
`L1-dcache`	≥ 10	≥ 3.8	≥ 3.8
`dTLB`	7	2.3	2.3

Figure 5.21: Indication of costs involved in `L1-dcache` and `dTLB` misses on a 2.6 GHz machine with Intel i7 architecture [HP11b]

	NPB-FT	NPB-MG	NPB-CG	NPB-IS	NPB-BT	NPB-SP	NPB-LU
W	34	881	1155	20	1800	5600	2400
A	34	121	1155	20	1800	5600	2000
B	104	601	5775	20	1800	5600	2000
C	104	681	5775	20	1800	5600	2000

Figure 5.22: Number of times that a master dispatches work to its slaves and subsequently waits for their signal in the NPB benchmarks. To get the number of transitions fired by a MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled version with k slaves, compute $4kn$, where n comes from this figure; to get the number of transitions fired by a MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled version with k slaves, compute $(1 + 3k)n$.

two kinds of *cache misses*: `L1-dcache` misses, which occur when the L1 cache of the CPU does not contain a requested piece of data, and `dTLB` misses, which occur when the data TLB contains no entry for a provided virtual address to translate into a physical address. Importantly, I measured cache misses during the whole run of a program, from start to end, using `perf-stat`. These numbers, thus, include also cache misses incurred during a program's initialization and finalization. In contrast, the time measurements, for which I reused the original code by Frumkin et al., start only after initialization and end already before finalization (i.e., my cache miss measurements span a longer interval than my time measurements). Therefore, take my measurements on cache misses with a grain of salt and use them just as a rough indication of cache behavior. Even under this proviso, though, these measurements give meaningful insight into the performance of compiler-generated code, as explained shortly. Figure 5.21 shows a conservative estimation of the costs involved in cache misses.

I make the following main observations about my experimental results:

- Overall, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks outperform their MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions (solid red lines versus dotted red lines). Recall from Chapter 3 that the latter versions impose an order in which the master sends signals to its slaves and vice versa, whereas the former versions do not impose such an order. As hinted at already in that chapter, not imposing an order indeed seems

to result in better performance.

- Despite the previous point, the Java versions of the NPB kernel benchmarks by Frumkin et al. outperform many of their FOCAML-to-Java-compiled versions by a substantial margin. This shows that the code generated by Lykos leaves room for further improvements.

- The numbers of cache misses seem a fair indicator of performance: fewer cache misses generally means better performance. This suggest that not only the number of machine instructions derived from compiler-generated code matters, but also its impact on memory and cache usage. For instance, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks generally incur substantially fewer cache misses than their MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions.

  Cache misses alone do not completely determine performance, though; the number of machine instructions surely plays a role too (demonstrated more clearly in the previous subsection, where I reported on experiments with protocols in isolation). Consider, for instance, NPB-IS, class A. Although the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled version for $k = 64$ incurs substantially fewer cache misses than the corresponding Java version, the latter version nevertheless substantially outperforms the former one.

- Going from class W to class A, from class A to class B, and from class B to class C, the speedup of the FOCAML-to-Java-compiled versions—or rather, their *slowdown*—generally improve: as the problem size increases, computation time progressively dominates total time, because interaction time stays nearly constant (i.e., the fraction of interaction time divided by computation time decreases as the problem size increases). As a measure for interaction time, Figure 5.22 shows the number of times that a master dispatches work to its slaves and subsequently waits for their signal. Increasing problem sizes, thus, work in favor of relatively slow implementations of protocol specifications due to "amortization of slowness" over a longer total time.

- The previous point applies to NPB-FT, NPB-MG, and NPB-CG but seems not to apply to NPB-IS, for which Figure 5.19 shows perhaps confusing experimental results, especially for $k = 64$: in class W and class C, the Java versions outperform the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions (dotted black lines versus solid red lines), but in class A and class B, the latter versions somewhat surprisingly outperform the former versions. I speculate that this has something to do with numbers of cache misses: as shown in Figure 5.20, for $k = 64$, in class W and class C, the former versions incur substantially fewer cache misses than the latter versions, but in class A and class B, the latter versions incur substantially fewer cache misses than the former versions.

To substantiate my speculation, as a first sanity check, I try to estimate the effect of cache misses on performance in the following *conservative* back-of-the-envelope calculation for $k = 64$ in class A. Using the numbers in Figure 5.21, the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled version spends *at least* 2.5 seconds more on cache misses than the Java version. By dividing this difference by the number of cores—a coarse estimation of how cache misses affect wall clock time—I get roughly a 100 milliseconds delay per core, about six times smaller than the difference in performance in Figure 5.19 (assuming a uniform distribution of the computational load over cores). Because I calculated conservatively (i.e., L1 cache misses cost substantially more than 10 cycles if also the L2 cache does not contain the required data; cache misses and computational load do not uniformly distribute over cores), these 100 milliseconds form a lower bound to the real delay per core caused by cache misses. If anything, this calculation shows that my speculation, that cache misses have a large enough impact to account for the observed difference in performance, seems at least *not un*reasonable.

Having passed the sanity check, next, I investigate the phenomenon at hand by studying the underlying reason for differences in cache misses. It turns out that this correlates with how the Java virtual machine manages the heap, the details of which I skip. Essentially, because the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled version has a different memory usage than the Java version, the Java virtual machine infers different sizes for the young/old generation portions of the heap for these two versions. These different sizes manifest in different cache behavior. To witness this, by explicitly setting the size of the young generation portion to the same value for both versions under study, not only their number of cache misses become similar *but also their performance*. This, then, serves as evidence for my previous speculation that their difference in number of cache misses causes the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions to outperform the Java versions in class A and class B for $k = 64$.

Figures 5.23–5.28 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 previous charts 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 in class W. Therefore, I have no measurements beyond $k = 16$ in class W for those benchmarks.

Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks. Different from the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks, however, the FOCAML-to-Java-compiled versions of the NPB application benchmarks have similar performance as their Java versions—and in some cases even better. The previous point about increasing problem sizes applies here too, though. Finally,

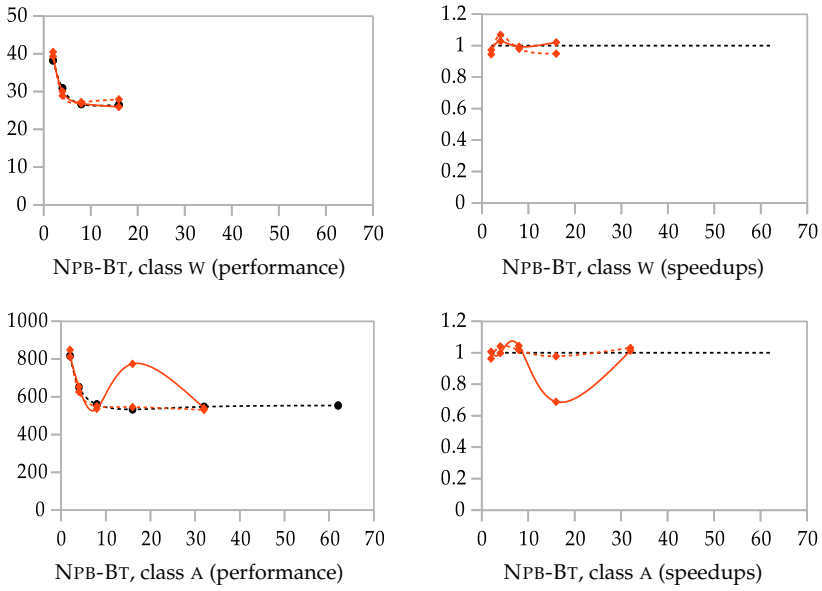


Figure 5.23: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

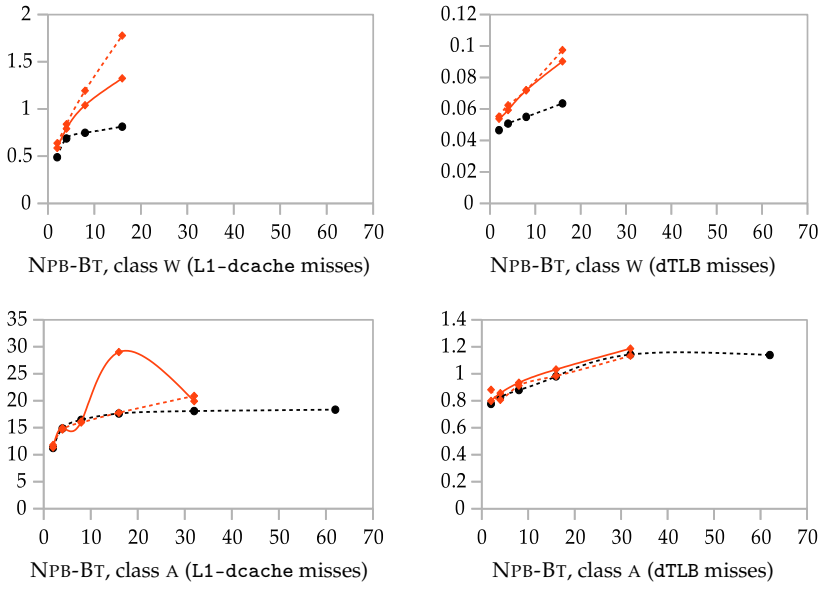


Figure 5.24: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

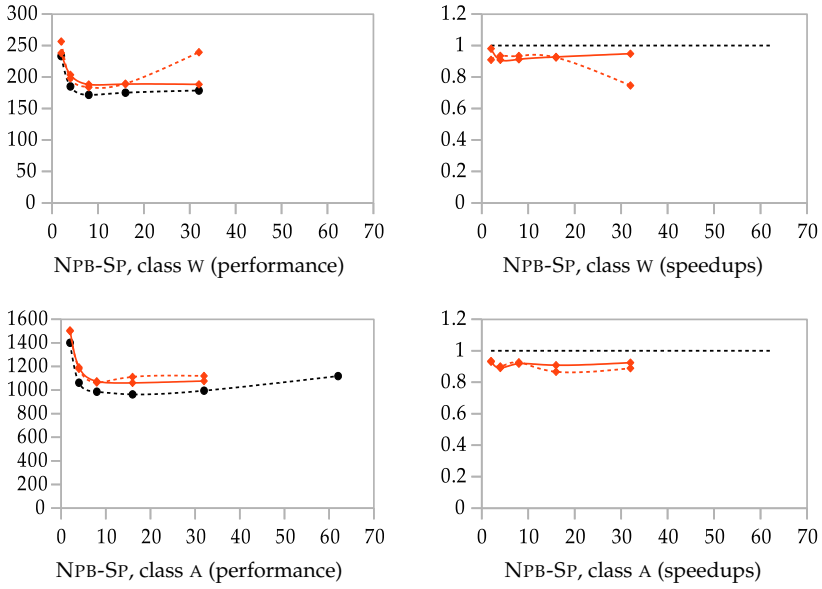


Figure 5.25: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

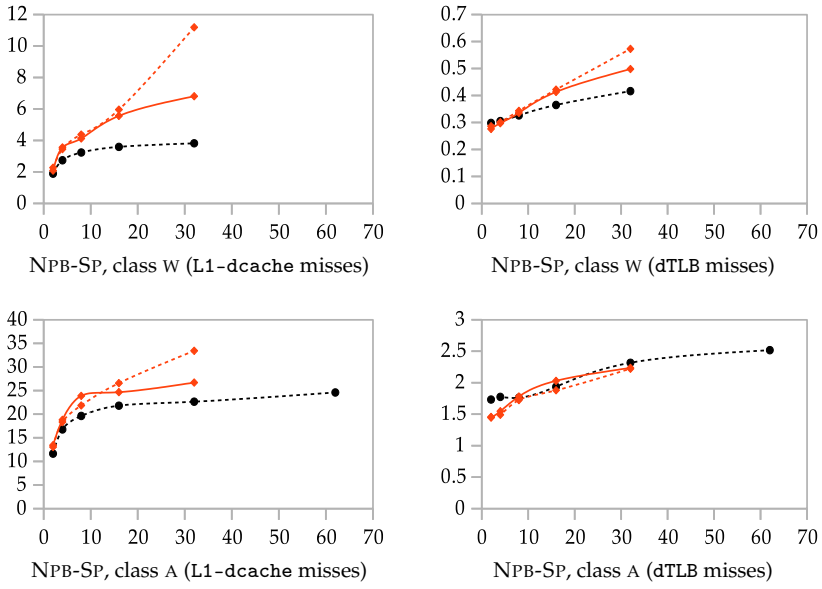


Figure 5.26: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

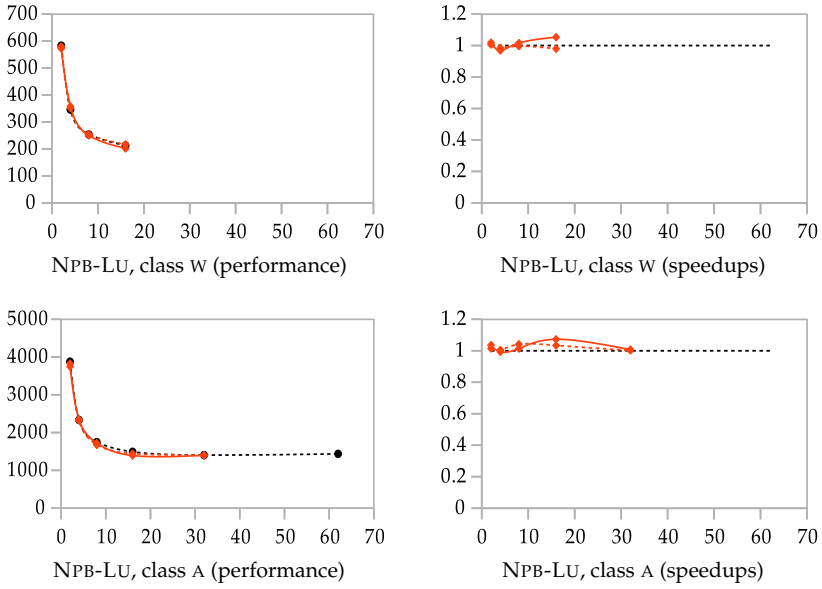


Figure 5.27: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

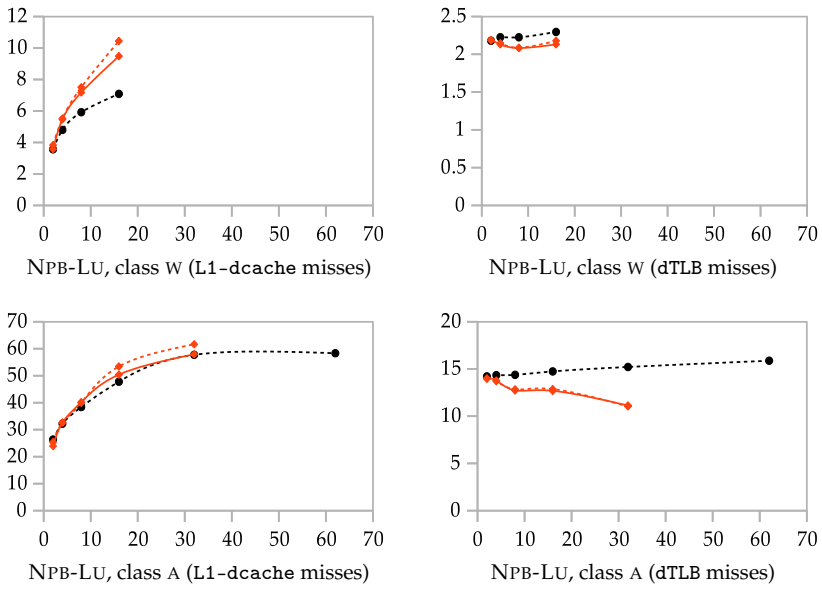


Figure 5.28: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

one anomalous run among five runs, which inexplicably took over three times longer to finish (i.e., half an hour versus 10 minutes), causes the spike for $k = 16$ in NPB-BT, class A, in Figure 5.24.

Their concrete results aside, this first round of experiments with the NPB benchmarks teaches an important lesson: evaluating the performance of compiler-generated code in the context of full programs yields new insights and requires more advanced analysis techniques (e.g., measuring and interpreting cache misses) than evaluating compiler-generated code in isolation, with empty producers and consumers. The experiments in this subsection, thus, nicely complement the experiments in the previous subsection.