Automata-theoretic protocol programming

Jongmans, Sung-Shik Theodorus Quirinus

Chapter 7

Improved Compilation III: Commandification

The experimental results in Chapter 6 show that syntactic subtraction already significantly improves performance. In this chapter, I present a complementary technique to further improve the data constraint checks involved in determining whether a transition can fire and, in particular, the expensive constraint solver calls made during such checks. Essentially, this new technique comprises the generation of a little, dedicated constraint solver for every data constraint at compile-time. At run-time, then, instead of calling a general-purpose constraint solver to check a data constraint, a protocol unit calls the more efficient constraint solver generated specifically for that data constraint.

In Section 7.1, I first introduce a basic sequential language in which to formalize these dedicated constraint solvers, called *data commands*. Subsequently, I present a translation of data constraints into data commands. Finally, I formally show that I can replace data constraints with their corresponding data commands in transition labels of constraint automata. In Section 7.2, I present an improved version of Lykos using the translation of data constraints into data commands, including new experimental results on performance.

Although the improvement presented in this chapter eventually results in improved compiler-generated code, as in Chapters 5 and 6, 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.

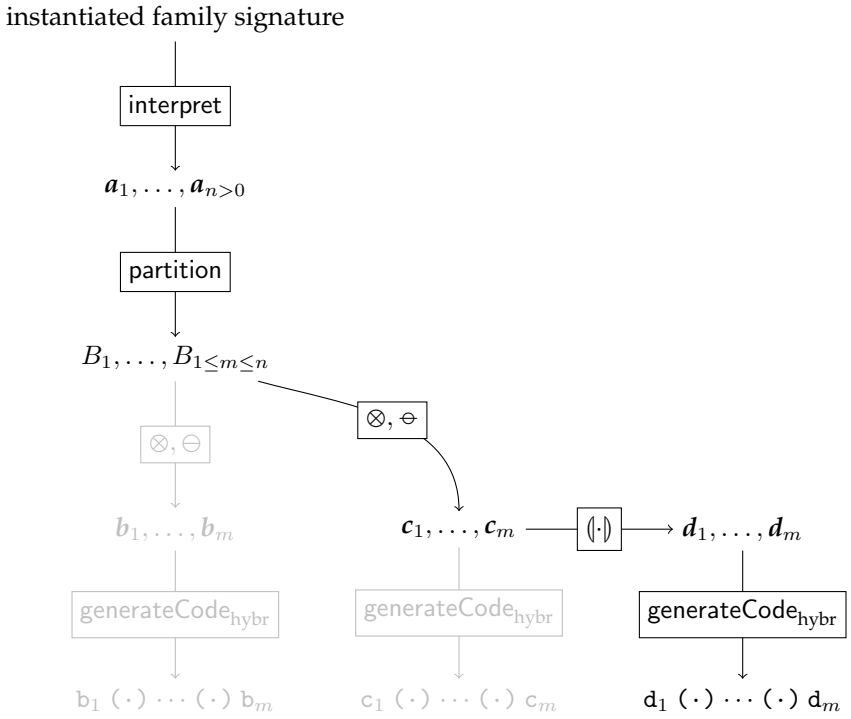

Figure 7.1: Hybrid compilation approach with syntactic subtraction and commandification

7.1 Theory

*(With Arbab, I previously published fragments of the material in this section in a conference paper [**J**A15b].)*

Data Commands

At run-time, general-purpose techniques for constraint solving—an NP-complete problem for finite domains—inflict not only overhead proportional to the size of a data constraint but also a constant overhead for preparing, making, and processing the result of the constraint solver call. Although one generally cannot escape using such techniques for checking arbitrary data constraints, a better alternative exists for many data constraints in practice. It starts with the observation that the data constraints in *all* constraint automata that I know of in the literature really constitute declarative specifications of a relatively straightforward imperative program. In this section, I therefore develop a technique for statically translating such a data constraint ϕ, off-line at compile-time, into a *data command*: a little imperative program that computes a data assignment σ

such that $\sigma \models \phi$, without general-purpose constraint solving. I call this kind of translation *commandification*. Essentially, I formalize and automate what programmers do when they write an imperative implementation of a declarative specification expressed as a data constraint. Figure 7.1 shows the resulting compilation approach, where $(\!|\cdot|\!)$ denotes commandification. By the end of this section, I make the class of data constraints currently supported by commandification precise.

I start by defining data commands, their semantics, and a proof system for reasoning about their execution.

Definition 44 (data commands)**.** *A data command is an object generated by the following grammar:*

$$\pi ::= \texttt{skip} \mid x := t \mid \phi \texttt{ -> } \pi \mid \pi \texttt{ ; } \pi \mid \varepsilon \qquad \textit{(data commands)}$$

$\mathbb{C}$OMM *denotes the set of all data commands.*

In the previous definition, ε denotes the empty data command, $x := t$ denotes an *assignment*, and ϕ -> π denotes a *failure statement*. Henceforth, I often write "value of x" instead of "the datum assigned to x".

I define an operational semantics for data commands based on an operational semantics for a sequential language by Apt et al. [AdBO09]. True to the idea that data commands solve data constraints, I model the *data state* that a data command executes in with either a function from data variables to data—a data assignment—or the distinguished object fail, which models abnormal termination. A *data configuration*, then, consists of a data command and a data state to execute that data command in.

Definition 45 (abnormal termination)**.** fail *is an unstructured object such that* fail $\notin \mathbb{A}$SSIGNM*.*

Definition 46 (data configurations)**.** *A data configuration is a pair* (π, ς) *where:*

- $\pi \in \mathbb{C}$OMM *(data command)*
- $\varsigma \in \mathbb{A}$SSIGNM $\cup \{\mathsf{fail}\}$ *(data state)*

$\mathbb{C}$ONF *denotes the set of all data configurations.*

A *transition system* on configurations formalizes their evolution in time.

Definition 47 (transition system on data configurations)**.** $\Longrightarrow \subseteq \mathbb{C}$ONF $\times$ $\mathbb{C}$ONF *denotes the smallest relation induced by the rules in Figure 7.2.*

Note that ϕ -> π indeed denotes a failure statement rather than a *conditional statement*: if the current data state violates the *guard* ϕ, execution abnormally terminates.

Through the transition system in Definition 47, I associate two different semantics with data commands. The *partial correctness semantics* of a data command π under a set of *initial* data state Σ consists of all the *final* data states Σ' to

$$\frac{}{(\texttt{skip}, \sigma) \Longrightarrow (\varepsilon, \sigma)} \tag{7.1}$$

$$\frac{}{(x := t, \sigma) \Longrightarrow (\varepsilon, \sigma[x \mapsto \mathsf{eval}_\sigma(t)])} \tag{7.2}$$

$$\frac{\sigma \models \phi}{(\phi \texttt{ -> } \pi, \sigma) \Longrightarrow (\pi, \sigma)} \tag{7.3} \qquad \frac{\sigma \not\models \phi}{(\phi \texttt{ -> } \pi, \sigma) \Longrightarrow (\varepsilon, \mathsf{fail})} \tag{7.4}$$

$$\frac{(\pi, \sigma) \Longrightarrow (\pi', \sigma') \textbf{ and } \pi' \neq \varepsilon}{(\pi \texttt{ ; } \pi'', \sigma) \Longrightarrow (\pi' \texttt{ ; } \pi'', \sigma')} \tag{7.5} \qquad \frac{(\pi, \sigma) \Longrightarrow (\varepsilon, \sigma')}{(\pi \texttt{ ; } \pi'', \sigma) \Longrightarrow (\pi'', \sigma')} \tag{7.6}$$

Figure 7.2: Addendum to Definition 47

which any of those initial states may evolve through execution of π. Notably, this partial correctness semantics ignores abnormal termination. In contrast, the *total correctness semantics* of π under Σ consists not only of Σ' but, if at least one execution abnormally terminates, also of fail.

Definition 48 (correctness semantics of data commands)**.** $\mathsf{Final}, \mathsf{Final}_{\mathsf{fail}} : \mathbb{C}\textsc{omm} \times 2^{\mathbb{A}\textsc{ssignm}} \to 2^{\mathbb{A}\textsc{ssignm} \cup \{\mathsf{fail}\}}$ *denote the functions defined by the following equations:*

$$\begin{aligned} \mathsf{Final}(\pi, \Sigma) &= \{\sigma' \mid \sigma \in \Sigma \textbf{ and } (\pi, \sigma) \Longrightarrow^* (\varepsilon, \sigma')\} \\ \mathsf{Final}_{\mathsf{fail}}(\pi, \Sigma) &= \mathsf{Final}(\pi, \Sigma) \cup \{\mathsf{fail} \mid \sigma \in \Sigma \textbf{ and } (\pi, \sigma) \Longrightarrow^* (\pi', \mathsf{fail})\} \end{aligned}$$

Apt et al. showed that all programs from a superset of the set of all data commands execute deterministically [AdBO09]. Consequently, also data commands execute deterministically.

Lemma 17 ([AdBO09])**.** $|\mathsf{Final}(\pi, \{\sigma\})| \leq 1 \textbf{ and } |\mathsf{Final}_{\mathsf{fail}}(\pi, \{\sigma\})| = 1$

To prove the correctness of commandification, I use Hoare logic [Hoa69], where *triples* $\{\phi\}\ \pi\ \{\phi'\}$ play a central role. In such a triple, *precondition* ϕ characterizes the set of initial data states, π denotes the data command to execute on those states, and *postcondition* ϕ' characterizes the set of final data states after executing π.

Definition 49 (triples)**.** $\mathbb{T}\textsc{ripl} = \mathbb{DC} \times \mathbb{C}\textsc{omm} \times \mathbb{DC}$ *denotes the set of all triples, typically denoted by* $\{\phi\}\ \pi\ \{\phi'\}$.

Let $[\![\phi]\!]$ denote the set of data states that satisfy ϕ (i.e., the data assignments characterized by ϕ). I interpret triples in two senses: that of partial correctness and that of total correctness. In the former case, a triple $\{\phi\}\ \pi\ \{\phi'\}$ holds true iff every final data state to which an initial data state characterized by ϕ can

$$\frac{}{\vdash_{\text{part}} \{\phi\}\ \texttt{skip}\ \{\phi\}} \quad (7.9) \qquad \frac{}{\vdash_{\text{tot}} \{\phi\}\ \texttt{skip}\ \{\phi\}} \quad (7.10)$$

$$\frac{}{\vdash_{\text{part}} \{\phi[t/x]\}\ x := t\ \{\phi\}} \quad (7.11) \qquad \frac{}{\vdash_{\text{tot}} \{\phi[t/x]\}\ x := t\ \{\phi\}} \quad (7.12)$$

$$\frac{\vdash_{\text{part}} \{\phi_1\}\ \pi_1\ \{\phi\} \quad \textbf{and} \quad \vdash_{\text{part}} \{\phi\}\ \pi_2\ \{\phi_2\}}{\vdash_{\text{part}} \{\phi_1\}\ \pi_1\ ;\ \pi_2\ \{\phi_2\}} \quad (7.13) \qquad \frac{\vdash_{\text{tot}} \{\phi_1\}\ \pi_1\ \{\phi\} \quad \textbf{and} \quad \vdash_{\text{tot}} \{\phi\}\ \pi_2\ \{\phi_2\}}{\vdash_{\text{tot}} \{\phi_1\}\ \pi_1\ ;\ \pi_2\ \{\phi_2\}} \quad (7.14)$$

$$\frac{\vdash_{\text{part}} \{\phi_1'\}\ \pi\ \{\phi_2'\} \quad \textbf{and} \quad \phi_1 \Rightarrow \phi_1' \quad \textbf{and} \quad \phi_2' \Rightarrow \phi_2}{\vdash_{\text{part}} \{\phi_1\}\ \pi\ \{\phi_2\}} \quad (7.15) \qquad \frac{\vdash_{\text{tot}} \{\phi_1'\}\ \pi\ \{\phi_2'\} \quad \textbf{and} \quad \phi_1 \Rightarrow \phi_1' \quad \textbf{and} \quad \phi_2' \Rightarrow \phi_2}{\vdash_{\text{tot}} \{\phi_1\}\ \pi\ \{\phi_2\}} \quad (7.16)$$

$$\frac{\vdash_{\text{part}} \{\phi \wedge \ell\}\ \pi\ \{\phi'\}}{\vdash_{\text{part}} \{\phi\}\ \ell\ \texttt{->}\ P\ \{\phi'\}} \quad (7.17) \qquad \frac{\vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\} \quad \textbf{and} \quad \phi \Rightarrow \ell}{\vdash_{\text{tot}} \{\phi\}\ \ell\ \texttt{->}\ \pi\ \{\phi'\}} \quad (7.18)$$

$$\frac{\vdash_{\text{part}} \{\phi\}\ \pi\ \{\phi_1\} \quad \textbf{and} \quad \vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi_2\}}{\vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi_1 \wedge \phi_2\}} \quad (7.19)$$

Figure 7.3: Addendum to Definition 51

evolve under π satisfies ϕ'; in the latter case, additionally, execution of π does not abnormally terminate.

Definition 50 (interpretation of triples). $\models_{\text{part}}, \models_{\text{tot}} \subseteq \mathbb{T}\text{RIPL}$ *denote the smallest relations induced by the following rules:*

$$\frac{\mathsf{Final}(\pi, [\![\phi]\!]) \subseteq [\![\phi']\!]}{\models_{\text{part}} \{\phi\}\ \pi\ \{\phi'\}} \quad (7.7) \qquad \frac{\mathsf{Final}_{\text{fail}}(\pi, [\![\phi]\!]) \subseteq [\![\phi']\!]}{\models_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\}} \quad (7.8)$$

To prove properties of data commands, I use the following sound *proof systems* for partial and total correctness, adopted from Apt et al. with some minor cosmetic changes [AdBO09].

Definition 51 (proof systems of triples). $\vdash_{\text{part}}, \vdash_{\text{tot}} \subseteq \mathbb{T}\text{RIPL}$ *denote the smallest relations induced by the rules in Figure 7.3.*

Theorem 16 ([AdBO09]). $\vdash_{\text{part}} \{\phi\}\ \pi\ \{\phi'\}$ **implies** $\models_{\text{part}} \{\phi\}\ \pi\ \{\phi'\}$

Theorem 17 ([AdBO09]). $\vdash_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\}$ **implies** $\models_{\text{tot}} \{\phi\}\ \pi\ \{\phi'\}$

Note that the first four rules for $\vdash_{\text{part}}$ and the first four rules for $\vdash_{\text{tot}}$ have the same premise/consequence. I use $\vdash_{\text{part}}$ to prove the *soundness* of commandification; I use $\vdash_{\text{tot}}$ to prove commandification's *completeness*.

Commandification

At run-time, to check if a transition (q, P, ϕ, q') can fire, a protocol unit first checks every port in P for that port's *readiness*. For instance, every input port should have a pending `put` (where "input" qualifies that port from the protocol perspective). Subsequently, the protocol unit checks whether a data state σ exists that (i) satisfies ϕ and (ii) subsumes an *initial data state* σ_{init} (i.e., $\sigma_{\text{init}} \subseteq \sigma$). If so, I call σ a *solution* of ϕ under σ_{init}. The domain of σ_{init} contains all *uncontrollable data variables* in ϕ: the input ports in P (intersected with $\mathsf{Free}(\phi)$) and ${}^\bullet m$ for every memory cell m in the constraint automaton (also intersected with $\mathsf{Free}(\phi)$). More precisely, σ_{init} maps every input port p in $\mathsf{Free}(\phi)$ to the particular datum *forced* to pass through p by the unit of parallelism on p's other side (e.g., the datum involved in p's pending `put`, performed by a neighboring worker unit), while σ_{init} maps every ${}^\bullet m$ in $\mathsf{Free}(\phi)$ to the datum that currently resides in m. Thus, before the protocol unit invokes a constraint solver for ϕ, it already fixes values for all uncontrollable data variables in ϕ; when subsequently invoked, a constraint solver may, in search of a solution for ϕ under σ_{init}, select values only for data variables outside σ_{init}'s domain. Slightly more formally:

$$\sigma_{\text{init}} = \left\{ p \mapsto d \;\middle|\; \begin{array}{l}[\text{the } \texttt{put} \text{ pending on input port } p \text{ involves datum } d] \\ \textbf{and}\ \ p \in \mathsf{Free}(\phi)\end{array} \right\} \cup \left\{ {}^\bullet m \mapsto d \;\middle|\; \begin{array}{l}[\text{memory cell } m \text{ currently contains datum } d] \\ \textbf{and}\ \ {}^\bullet m \in \mathsf{Free}(\phi)\end{array} \right\}$$

With commandification, instead of invoking a constraint solver, the protocol unit executes a compiler-generated data command for ϕ on σ_{init}, thereby gradually extending σ_{init} to a full solution. This compiler-generated data command essentially works as an efficient, little, dedicated constraint solver for ϕ.

To commandify a data constraint of the form $\ell_1 \wedge \cdots \wedge \ell_k$, I construct a data command that (i) enforces as many data literals of the form $t_1 = t_2$ as possible with assignments and (ii) checks all remaining data literals with failure statements. I call data literals of the form $t_1 = t_2$ *data equalities*. To exemplify such commandification, recall data constraint ϕ_{eg} on page 178. In this data constraint, let C denote an input port and let x denote a memory cell. In that case, the set of uncontrollable data variables in ϕ_{eg} consists of C and ${}^\bullet$x. Now, ϕ_{eg}

has six correct commandifications:

$\pi_1 =$ B := •x ;
D := C ;
E := add(B, D) ;
F := E ;
G := E
¬Odd(G) -> skip ;

$\pi_2 =$ B := •x ;
D := C ;
E := add(B, D) ;
G := E
F := E ;
¬Odd(G) -> skip ;

$\pi_3 =$ B := •x ;
D := C ;
E := add(B, D) ;
G := E
¬Odd(G) -> skip ;
F := E ;

$\pi_4 =$ D := C ;
B := •x ;
E := add(B, D) ;
F := E ;
G := E
¬Odd(G) -> skip ;

$\pi_5 =$ D := C ;
B := •x ;
E := add(B, D) ;
G := E
F := E ;
¬Odd(G) -> skip ;

$\pi_6 =$ D := C ;
B := •x ;
E := add(B, D) ;
G := E
¬Odd(G) -> skip ;
F := E ;

I stipulate the same precondition for each of these data commands, namely that •x and C have a non-nil value (later formalized as data literals •x = •x and C = C). This precondition models that the execution of these data commands should always start on an initial data state over the uncontrollable data variables •x and C. Under this precondition, if a protocol unit executes π_1, it first assigns the values of •x and C to B and D. Subsequently, it assigns the evaluation of add(B, D) to E. Next, it assigns the value of E to F and G. Finally, it checks ¬Odd(G) with a failure statement. Data commands π_2 and π_3 differ from data command π_1 only in the order of the last three steps; data commands π_4, π_5 and π_6 differ from π_1, π_2 and π_3 only in the order of the first two steps. If execution of π_i on σ_{init} successfully terminates, the resulting final data state σ satisfies ϕ_{eg}. I call this *soundness*. Moreover, if a σ' exists such that $\sigma' \models \phi_{\text{eg}}$ and $\sigma_{\text{init}} \subseteq \sigma'$, execution of π_i successfully terminates. I call this *completeness*.

Generally, soundness and completeness crucially depend on the order in which assignments and failure statements follow each other in π. For instance, changing the order of G := E and ¬Odd(G) -> skip in the previous example yields a data command whose execution always fails (because G does not have a value yet on evaluating the guard of the failure statement). Such a trivially sound but incomplete data constraint serves no use. As another complication, not every data equality can become an assignment. In a first class of cases, neither the left-hand side nor the right-hand side of a data equality matches data variable x. For instance, I *must* translate add(B, D) = mult(B, D) into a failure statement, because I clearly cannot assign either of its two operands to the other. In a second class of cases, multiple data equalities in a data constraint have a left-hand side or a right-hand side that matches the same data variable x. For instance, I can translate only one data equality in E = add(B, D) ∧ E = mult(B, D) into an assignment, after which I *must* translate the other one into a failure statement, to avoid conflicting assignments to E.

To deal with these complications, I define a *precedence relation* on the data literals in a normal data constraint that formalizes their dependencies. Re-

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{and} \ x \in \mathsf{Variabl}(\ell)}{x = t \sqsubseteq \ell} \tag{7.20}$$

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{and} \ \left[\ell \neq x' = t' \ \textbf{for all} \ x', t'\right]}{x = t \sqsubseteq \ell} \tag{7.21}$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \ \textbf{and} \ \ell_2 \sqsubseteq \ell_3 \ \textbf{and} \ \ell_2 \notin \{\ell_1, \ell_3\}}{\ell_1 \sqsubseteq \ell_3} \tag{7.22}$$

Figure 7.4: Addendum to Definition 53

call from Definition 38 that every normal data constraint consists of a conjunctive kernel of data literals, enveloped with existential quantifications. First, for technical convenience, I introduce a function that extends $\mathsf{Liter}(\varphi)$ (i.e., the data literals in the kernel of φ) with its "symmetric data equalities".

Definition 52 (=-symmetric closure)**.** $\mathsf{Liter}^{=} : \mathbb{DC}_{\exists,\wedge} \to 2^{\mathbb{DC}_{\exists,\wedge}}$ *denotes the function defined by the following equation:*

$$\mathsf{Liter}^{=}(\varphi) = \mathsf{Liter}(\varphi) \cup \{t_2 = t_1 \mid t_1 = t_2 \in \mathsf{Liter}(\varphi)\}$$

Obviously, because $t_1 = t_2 \equiv t_2 = t_1$, I have $\bigwedge \mathsf{Liter}(\varphi) \equiv \bigwedge \mathsf{Liter}^{=}(\varphi)$ for all φ.

Definition 53 (precedence I)**.** $\sqsubseteq : \mathbb{DC}_{\exists,\wedge} \to 2^{\mathbb{DC} \times \mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\sqsubseteq(\varphi) = \sqsubseteq$$

where $\sqsubseteq$ *denotes the smallest relation induced by the rules in Figure 7.4.*

I usually write $\sqsubseteq_\varphi$ instead of $\sqsubseteq(\varphi)$ and use $\sqsubseteq_\varphi$ as an infix relation. In words, $x = t \sqsubseteq_\varphi \ell$ means that the assignment $x := t$ precedes the commandification of ℓ (i.e., ℓ depends on x). Rule 7.20 deals with the previously discussed first class of data-equalities-that-cannot-become-assignments, by imposing precedence only on data literals of the form $x = t$; shortly, I comment on the second class of data-equalities-that-cannot-become-assignments. Rule 7.21 conveniently ensures that every $x = t$ precedes all differently shaped data literals. Strictly speaking, I probably do not need this rule, but it simplifies some notation and proofs later on.

For the sake of argument—generally, this does *not* hold true—suppose that a precedence relation $\sqsubseteq_\varphi$ denotes a *strict partial order* on $\mathsf{Liter}^{=}(\varphi)$. In that case, I can *linearize* $\sqsubseteq_\varphi$ to a strict total order $<$ (i.e., embedding $\sqsubseteq_\varphi$ into $<$ such that $\sqsubseteq_\varphi \subseteq <$) with a topological sort on the digraph $(\mathsf{Liter}^{=}(\varphi), \sqsubseteq_\varphi)$ [Kah62, Knu97]. Intuitively, such a linearization gives me an order in which I can translate data literals in $\mathsf{Liter}^{=}(\varphi)$ to data commands in a sound and complete way.

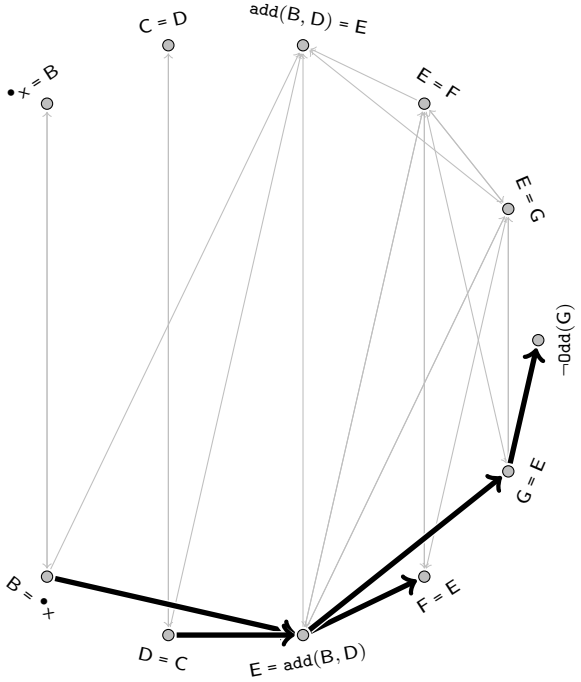

Figure 7.5: Digraph for precedence relation $\sqsubseteq_{\phi_{\text{eg}}}$ (without loop arcs and without arcs induced by Rule 7.21, to avoid further clutter). An arc (ℓ, ℓ') corresponds to $\ell \sqsubseteq_{\phi_{\text{eg}}} \ell'$. Arcs between the same data vertices, but in different directions, lie on top of each other. Bold arcs represent a fragment of the strict partial order extracted from $\sqsubseteq_{\phi_{\text{eg}}}$.

Shortly, I give an algorithm for doing so and indeed prove its correctness. Problematically, however, $\sqsubseteq_\varphi$ generally does not denote a strict partial order: generally, it violates asymmetry and irreflexivity (i.e., graph-theoretically, it contains many cycles). For instance, Figure 7.5 shows the digraph $(\mathsf{Liter}^{=}(\phi_{\text{eg}}), \sqsubseteq_{\phi_{\text{eg}}})$, which indeed contains cycles. For now, I defer this issue to the next subsection, because it forms a concern orthogonal to the commandification algorithm and its correctness. Until then, I simply assume the existence of a procedure for extracting a strict partial order from $\sqsubseteq_\varphi$, represented by bold arcs in Figure 7.5.

Algorithm 2 translates a normal data constraint φ, a set of variables X, and a binary relation on data literals $<$ to a data command π. It requires the following on its input. First, $<$ should denote a strict total order on the =-symmetric closure of φ's data literals. Let n denote *a*—not necessarily *the*—number of data equalities in $\mathsf{Liter}^{=}(\varphi)$, and let m denote the number of remaining data literals in $\mathsf{Liter}^{=}(\varphi)$. Then, $\ell_1, \ldots, \ell_{n+m}$ denote the data literals in $\mathsf{Liter}^{=}(\varphi)$ such that (i) their indices respect $<$ and (ii) every ℓ_i denotes $x_i = t_i$ for $1 \leq i \leq n$. Next, for every data variable in a data literal in $\mathsf{Liter}^{=}(\varphi)$, but outside the set

Algorithm 2 Algorithm for translating a normal data constraint φ, a set of variables X, and a binary relation on data literals $<$ to a data command π

Require: $<$ denotes a strict total order on $\mathsf{Liter}^{=}(\varphi)$
and $\mathsf{Liter}^{=}(\varphi) = \{\ell_1, \ldots, \ell_{n+m}\}$
and $\ell_1 < \cdots < \ell_n < \ell_{n+1} < \cdots < \ell_{n+m}$
and $\ell_1 = x_1 = t_1$ **and** $\cdots$ **and** $\ell_n = x_n = t_n$
and $\mathsf{Variabl}(\varphi) \setminus X \subseteq \{x_1, \ldots, x_n\}$
and
$$\left[\begin{array}{c}\left[\begin{array}{l}\left[x = t \in \mathsf{Liter}^{=}(\varphi) \text{ and } x' \in \mathsf{Variabl}(t)\right] \text{ implies}\\ \quad \left[x' \in X \text{ or } \left[x' = t' < x = t \text{ for some } t'\right]\right]\end{array}\right]\\ \text{for all } x, x', t\end{array}\right]$$

function ALGORITHM2($\varphi, X, <$)
$\pi := \texttt{skip}$
$i := 1$
while $i \leq n$ **do**
if $x_i \in X \cup \{x_1, \ldots, x_{i-1}\}$ **then**
$\pi := (\pi \texttt{ ; } x_i = t_i \texttt{ -> skip})$
else
$\pi := (\pi \texttt{ ; } x_i := t_i)$
$i := i + 1$
while $i \leq n + m$ **do**
$\pi := (\pi \texttt{ ; } \ell_i \texttt{ -> skip})$
$i := i + 1$
return π

Ensure: $\vdash_{\text{part}} \{\bigwedge\{x = x \mid x \in X\}\}\ \pi\ \{\ell_1 \wedge \cdots \wedge \ell_{n+m}\}$
and
$$\left[\left[\begin{array}{l}\sigma \models \ell_1 \wedge \cdots \wedge \ell_{n+m} \text{ implies}\\ \vdash_{\text{tot}} \{\bigwedge\{x = \sigma(x) \mid x \in X\}\}\\ \quad \pi\\ \quad \{\bigwedge\{x = \sigma(x) \mid x \in X \cup \{x_1, \ldots, x_n\}\}\}\end{array}\right] \text{ for all } \sigma\right]$$

of uncontrollable variables X, a data equality $x_i = t_i$ should exist. Otherwise, such a data variable can get a value only through search—exactly what commandification tries to avoid—and not through assignment; *underspecified* data constraints fundamentally lie outside the scope of commandification in general and Algorithm 2 in particular. Finally, if term t in a data equality $x = t$ depends on a variable x', a data equality $x' = t'$ should precede $x = t$ under $<$. The rules in Definition 53 induce precedence relations for which all these requirements hold true, except that those precedence relations not necessarily denote strict partial orders and, hence, may not admit linearization. Consequently, the precedence relations in Definition 53 may not yield strict total orders as required by Algorithm 2. I address this issue in the next subsection.

Assuming satisfaction of its requirements, Algorithm 2 works as follows. It first loops over the first n (according to $<$) x_i = t_i data literals. If an assignment for x_i already exists in the data command under construction π, Algorithm 2 translates x_i = t_i to a failure statement; otherwise, it translates x_i = t_i to an assignment. This approach resolves issues with the previously discussed second class of equalities-that-cannot-become-assignments. After the first loop, the algorithm uses a second loop to translate the remaining m data literals to failure statements. The algorithm runs in time linear in $n + m$, and it terminates.

Algorithm 2 ensures the soundness and completeness of π. Note that I use a different proof system for soundness (partial correctness, $\vdash_{\text{part}}$) than for completeness (total correctness, $\vdash_{\text{tot}}$).

Theorem 18. *Algorithm 2 is correct.*

Algorithm 2 has the minor issue that it may produce more failure statements than strictly necessary. For instance, if I run Algorithm 2 on the total order extracted from $\sqsubseteq_{\phi_{\text{eg}}}$ in Figure 7.5, I get both the assignment `D := C` and the unnecessary failure statement `C = D -> skip`. After all, the digraph contains both `D = C` and `C = D`, one of which I added while computing $\mathsf{Liter}^{=}(\phi_{\text{eg}})$ to account for the symmetry of `=`. Generally, such symmetric data literals result either in one assignment and one failure statement or in two failure statements; one can easily prove that symmetric data literals never result in two assignments. In both cases, one can safely remove one of the failure statements, because successful termination of the remaining statement already accounts for the removed failure statement.

Commandification with Cycles

Algorithm 2 requires that $<$ denotes a strict total order. Precedence relations in Definition 53 of $\sqsubseteq$, however, do not yield such orders: graph-theoretically, they may contain cycles. In this subsection, I present a solution for this problem. I start by extending the previous precedence relations with a unique least element, denoted by $\bigstar$, and by making dependencies of data literals on uncontrollable variables explicit. In the following definition, let X denote such a set of uncontrollable variables.

Definition 54 (precedence II)**.** $\sqsubseteq : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to 2^{(\mathbb{DC} \cup \{\bigstar\}) \times \mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\sqsubseteq(\varphi, X) = \sqsubseteq$$

where $\sqsubseteq$ *denotes the smallest relation induced by the rules in Figure 7.6.*

I usually write $\sqsubseteq_{\varphi}^{X}$ instead of $\sqsubseteq(\varphi, X)$ and use $\sqsubseteq_{\varphi}^{X}$ as an infix relation. The two new rules state that data literals in which only uncontrollable variables occur "depend" on $\bigstar$.

$$\frac{\ell_1 \sqsubseteq_\phi \ell_2}{\ell_1 \sqsubseteq \ell_2} \tag{7.23}$$

$$\frac{\ell \in \mathsf{Liter}^{=}(\varphi) \quad \textbf{and} \quad \mathsf{Variabl}(\ell) \subseteq X}{\bigstar \sqsubseteq \ell} \tag{7.24}$$

$$\frac{x = t \in \mathsf{Liter}^{=} \quad \textbf{and} \quad \mathsf{Variabl}(t) \subseteq X}{\bigstar \sqsubseteq x = t} \tag{7.25}$$

Figure 7.6: Addendum to Definition 54

A precedence relation $\sqsubseteq_\varphi^X$ denotes a strict partial order if its corresponding digraph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}, \sqsubseteq_\varphi^X)$ defines a $\bigstar$-*arborescence*: a digraph consisting of $n-1$ arcs such that a path exists from $\bigstar$ to each of its n vertices [KV08]. Equivalently, in a $\bigstar$-arborescence, $\bigstar$ has no incoming arcs, every other vertex has exactly one incoming arc, and the arcs form no cycles [KV08]. The first formulation seems more intuitive here: every path from $\bigstar$ to some data literal ℓ represents an order in which Algorithm 2 should translate the data literals on that path to ensure the correctness of the translation of ℓ. The second formulation simplifies observing that arborescences correspond to strict partial orders.

A naive approach to extract a strict partial order from $\sqsubseteq_\varphi^X$ consists of computing a $\bigstar$-arborescence of the digraph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}, \sqsubseteq_\varphi^X)$. Even if such a $\bigstar$-arborescence exists, however, this approach does not work as expected if $\mathsf{Liter}^{=}(\varphi)$ contains a data literal $x = t$ where t has more than one data variable. For instance, by definition, every arborescence of the digraph in Figure 7.5 has only one incoming arc for $\mathsf{E} = \mathtt{add}(\mathsf{B}, \mathsf{D})$, even though assignments to *both* B *and* D must precede an assignment to E. Because these dependencies exist as two separate arcs, no arborescence can capture them. To solve this, I should somehow represent the dependencies of $\mathsf{E} = \mathtt{add}(\mathsf{B}, \mathsf{D})$ with a single incoming arc. I can do so by allowing arcs to have multiple tails (i.e., one for every data variable). In that case, I can replace the two separate incoming arcs of $\mathsf{E} = \mathtt{add}(\mathsf{B}, \mathsf{D})$ with a single two-tailed incoming arc as in Figure 7.7. The two tails make explicit that to evaluate `add`, I need values for both its arguments: multiple tails represent a conjunction of dependencies of a data literal.

By combining single-tailed arcs into multiple-tailed arcs, I effectively transform the digraphs considered so far into B-*graphs*, a special kind of hypergraph with only B-*arcs* (i.e., *b*ackward hyperarcs, i.e., hyperarcs with exactly one head) [GLPN93]. Generally, I cannot derive such B-graphs from precedence relations as in Definition 54: their richer structure makes B-graphs more expressive—they convey strictly more information—than digraphs. In contrast, I can easily transform a B-graph into a precedence relation by splitting B-arcs into single-tailed arcs in the obvious way. Deriving precedence relations from more expressive B-graphs therefore constitutes a correct way of obtaining strict total orders that satisfy the requirements of Algorithm 2; doing so just

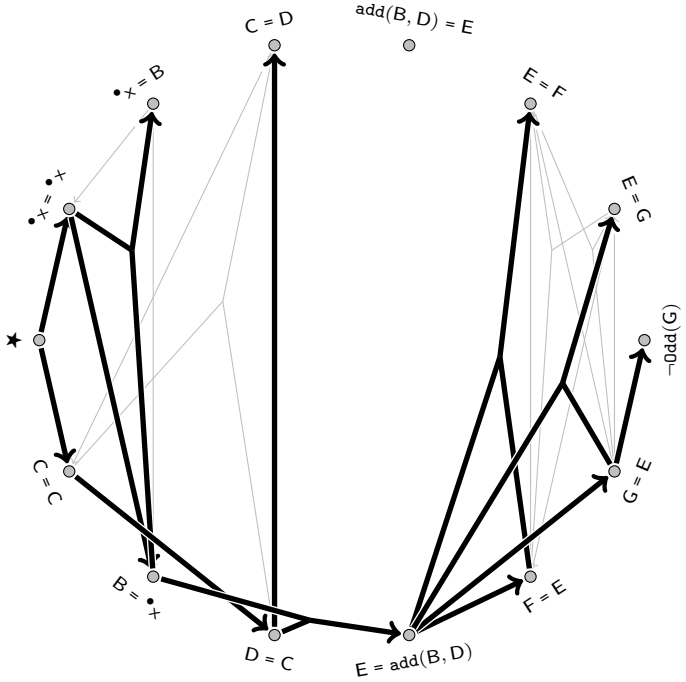

Figure 7.7: B-graph corresponding to the digraph in Figure 7.5 (without loop B-arcs and without three-tailed B-arcs, to avoid further clutter). An arc (ℓ, ℓ') corresponds to $\ell \sqsubseteq_{\phi_{\text{eg}}} \ell'$. Bold arcs represent an arborescence.

eliminates information that this algorithm does not care about.

Thus, I propose the following. Instead of formalizing dependencies among data literals in a set $\text{Liter}^{=}(\varphi) \cup \{\bigstar\}$ directly as a precedence relation, I first formalize those dependencies as a B-graph. If the resulting B-graph defines a $\bigstar$-arborescence, I can directly extract a cycle-free precedence relation $\sqsubset$. Otherwise, I compute a $\bigstar$-arborescence of the resulting B-graph and extract a cycle-free precedence relation $\sqsubset$ afterward. Either way, $\sqsubset$ denotes a strict partial order whose linearization satisfies the requirements in Algorithm 2.

Definition 55 (B-precedence)**.** $\blacktriangleleft : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to 2^{(2^{\mathbb{DC}} \cup \{\bigstar\}) \times \mathbb{DC}}$ *denotes the function defined by the following equation:*

$$\blacktriangleleft(\varphi, X) = \blacktriangleleft$$

where $\blacktriangleleft$ *denotes the smallest relation induced by the rules in Figure 7.8.*

I usually write $\blacktriangleleft_\varphi^X$ instead of $\blacktriangleleft(\varphi, X)$ and use $\blacktriangleleft_\varphi^X$ as an infix relation. Rule 7.26 generalizes Rule 7.20 in Definition 53, by joining sets of dependencies of a data literal in a single B-arc. Rule 7.27 states that $x = t$ does not necessarily depend

$$\frac{\begin{array}{l}\ell \in \mathsf{Liter}^{=}(\varphi) \\ \textbf{and } \mathsf{Variabl}(\ell) = \{x_1, \ldots, x_k\} \\ \textbf{and } x_1 = t_1, \ldots, x_k = t_k \in \mathsf{Liter}^{=}(\varphi) \cup \{\hat{x} = \hat{x} \mid \hat{x} \in X\}\end{array}}{\{x_1 = t_1, \ldots, x_k = t_k\} \blacktriangleleft \ell} \tag{7.26}$$

$$\frac{\begin{array}{l}x = t \in \mathsf{Liter}^{=}(\varphi) \\ \textbf{and } \mathsf{Variabl}(t) = \{x_1, \ldots, x_k\} \\ \textbf{and } x_1 = t_1, \ldots, x_k = t_k \in \mathsf{Liter}^{=}(\varphi) \cup \{\hat{x} = \hat{x} \mid \hat{x} \in X\}\end{array}}{\{x_1 = t_1, \ldots, x_k = t_k\} \blacktriangleleft x = t} \tag{7.27}$$

$$\frac{x \in X}{\bigstar \blacktriangleleft x = x} \tag{7.28}$$

Figure 7.8: Addendum to Definition 55

on x—as implied by Rule 7.26—but only on the free variables in t (i.e., I can derive a value for x from values of the data variables in t). Note that through Rules 7.26 and 7.27, I extend the previous domain $\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\}$ with *semantically insignificant* data equalities of the form $x = x$, each of which I relate to $\bigstar$ with Rule 7.28. I do this only for the technical convenience of treating both uncontrollable variables in X (which may have no data equalities in $\mathsf{Liter}^{=}(\varphi)$) and the other variables (which must have) in a uniform way. For instance, Figure 7.7 shows the B-graph for data constraint ϕ_{eg}.

Generally, in a B-graph, data literals can have multiple incoming B-arcs. Such multiple incoming B-arcs represent a disjunction of conjunctions of dependencies. Importantly, as long as Algorithm 2 respects the dependencies represented by *one* incoming B-arc, the other incoming B-arcs do not matter. An arborescence, which contains one incoming B-arc for every data literal, therefore preserves enough dependencies. Theorem 19 makes this more precise.

One can straightforwardly compute an arborescence of a B-graph

$$(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}, \blacktriangleleft_{\varphi}^{X})$$

with an exploration algorithm reminiscent of breadth-first search. First, let $\lhd \subseteq \blacktriangleleft_{\varphi}^{X}$ denote the arborescence under computation, and let $L_{\text{done}} \subseteq \mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}$ denote the set of vertices (i.e., data literals) already explored; initially, $\lhd = \emptyset$ and $L_{\text{done}} = \{\bigstar\}$. Now, given some L_{done}, compute a set of vertices L_{next} connected only to vertices in L_{done} by a B-arc in $\blacktriangleleft_{\varphi}^{X}$. Then, for every vertex in L_{next}, add an incoming B-arc to $\lhd$.[1] Afterward, add L_{next} to L_{done}. Repeat this process until L_{next} becomes empty. Once that happens,

[1] If a vertex ℓ in L_{next} has multiple incoming B-arcs, the choice among them matters not: the choice remains local, because every B-arc has only one head (i.e., adding an ℓ-headed B-arc to $\lhd$ cannot cause another vertex to get multiple incoming B-arcs, which would invalidate the arbores-

$$\frac{\ell_1 \in \mathsf{Liter}^{=}(\varphi) \cap L \ \textbf{and} \ L \lhd_{\varphi}^{X} \ell_2}{\ell_1 \sqsubset \ell_2} \tag{7.29}$$

$$\frac{x = t, \ell \in \mathsf{Liter}^{=}(\varphi) \ \textbf{and} \ \left[\ell \neq x' = t' \ \textbf{for all} \ x', t'\right]}{x = t \sqsubset \ell} \tag{7.30}$$

$$\frac{\ell_1 \sqsubset \ell_2 \ \textbf{and} \ \ell_2 \sqsubset \ell_3 \ \textbf{and} \ \ell_2 \notin \{\ell_1, \ell_3\}}{\ell_1 \sqsubset \ell_3} \tag{7.31}$$

Figure 7.9: Addendum to Definition 56

either $\lhd$ contains an arborescence (if $L_{\text{done}} = L$) or no arborescence exists. This computation runs in linear time, in the size of the B-graph. See also Footnote 1. Henceforth, let $\lhd_{\varphi}^{X}$ denote the final arborescence so computed; if no arborescence exists, I stipulate $\lhd_{\varphi}^{X} = \emptyset$.

Definition 56 (precedence III). $\sqsubset : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to \mathbb{DC} \times \mathbb{DC}$ *denotes the function defined by the following equation:*

$$\sqsubset(\varphi, X) = \sqsubset$$

where $\sqsubset$ denotes the smallest relation induced by the rules in Figure 7.9.

I usually write $\sqsubset_{\varphi}^{X}$ instead of $\sqsubset(\varphi, X)$. Rules 7.30 and 7.31 have the same premise/consequence as Rules 7.21 and 7.22; Rule 7.29 straightforwardly splits B-arcs into single-tailed arcs. For instance, the bold arcs in Figure 7.5 represent a fragment of the precedence relation so derived from the arborescence in Figure 7.7.

For every $\sqsubset_{\varphi}^{X}$ induced from a nonempty $\bigstar$-arborescence (i.e., $\lhd_{\varphi}^{X} \neq \emptyset$), let $<_{\varphi}^{X}$ denote its linearization. The following theorem states that this linearization satisfies the requirements of Algorithm 2.

Theorem 19. $\lhd_{\varphi}^{X} \neq \emptyset$ **implies** $\left[(\varphi, X, <_{\varphi}^{X})\ \textit{satisfies Algorithm 2}\right]$

If the B-graph $(\mathsf{Liter}^{=}(\varphi) \cup \{\bigstar\} \cup \{x = x \mid x \in X\}, \blacktriangleleft_{\varphi}^{X})$ neither defines nor contains a $\bigstar$-arborescence, no B-graph equivalent of a path [AFF01] exists from $\bigstar$ to at least one vertex ℓ. In that case, the other vertices fail to resolve at least one of ℓ's dependencies. This occurs, for instance, when ℓ depends on x, but the B-graph contains no $x = t$ vertex. As another example, consider a recursive data equality $x = t$ with $x \in \mathsf{Variabl}(t)$: unless another data equality $x = t'$

cence). General hypergraphs, whose hyperarcs can have multiple heads, violate this property (i.e., the choice of which hyperarc to add becomes global instead of local). As a result, and in stark contrast to B-graphs, one cannot compute arborescences of general hypergraphs—an NP-complete problem [Woe92]—in polynomial time (if P $\neq$ NP).

with $t \neq t'$ exists, all its incoming B-arcs contain loops to itself. Consequently, no arborescence exists. In practice, such cases inherently require constraint solving techniques with backtracking to find a value for x. Nonexistence of a ★-arborescence thus signals a fundamental limitation to the applicability of Algorithm 2 (although mixed techniques of translating some parts of a data constraint to a data command at compile-time and leaving other parts to a constraint solver at run-time seem worthwhile to explore; I leave those for future work). Thus, the set of data constraints to which I can apply Algorithm 2 contains those (i) whose B-graph has a ★-arborescence, which guarantees linearizability of the induced precedence relation, and (ii) that satisfy also the rest of the requirements in Algorithm 2.

The constraint programming community has already observed that, for constraint solving, "if domain specific methods are available they should be applied *instead* [sic] of the general methods" [Apt09a]. Commandification pushes this piece of conventional wisdom to an extreme: essentially, every data command generated for a data constraint ϕ by Algorithm 2 constitutes a little, dedicated constraint solver capable of solving only ϕ. Nevertheless, execution of data commands bears similarities with *constraint propagation* techniques, in particular with *forward checking* [BMFL02]. Generally, constraint propagation aims to reduce the search space of a constraint satisfaction problem by transforming it into an equivalent "simpler" one, where variables have smaller domains, or where constraints refer to fewer variables. With forward checking, whenever a variable x gets a value d, a constraint solver removes values from the domains of all subsequent variables that, given d, violate a constraint. In the case of an equality $x = x'$, for instance, forward checking reduces the domain of x' to the singleton $\{d\}$ after an assignment of d to x. Commandification implicitly uses that same property of equality, but instead of explicitly representing the domain of a variable and the reduction of this domain to a singleton at run-time, commandification already turns the equality into an assignment at compile-time. Commandification may also remind one of classical *Gaussian elimination* for solving systems of linear equations over the reals [Apt09b]: there too, one orders variables and substitutes values/expressions for variables in other expressions. Data constraints, however, have a significantly different structure from real numbers, which makes solving data constraints directly via Gaussian elimination at least not obvious.

Commandification in Constraint Automata

To formally introduce data commands in constraint automata, I introduce commandification as a unary operation on constraint automata. First, because I want to avoid ad-hoc modifications to Definitions 15 and 19 (of data constraints and constraint automata), I present an encoding of data commands as data relations. In the following definition, let φ denote a normal data constraint in a normalized constraint automaton, let X denote the set of uncontrollable data variables in φ, and let $x_1, \ldots, x_k$ denote the free data variables in φ, ordered by $<_{\mathbb{T}\text{ERM}}$. Then, the data relation R, which encodes the commandification π

of φ, holds true of a data tuple $(d_1, \ldots, d_k)$ iff execution of π on an initial data state (over the variables in X) successfully terminates on a data state σ that maps every x_i to d_i.

Definition 57 (data commands as data relations)**.** $\mathsf{comm} : \mathbb{DC}_{\exists,\wedge} \times 2^{\mathbb{X}} \to \mathbb{DC}_{\exists,\wedge}$ *denotes the function defined by the following equation:*

$$\mathsf{comm}(\varphi, X) = \begin{cases} R(x_1, \ldots, x_k) \ \textbf{if} \ \begin{bmatrix} \mathsf{Free}(\varphi) = \{x_1, \ldots, x_k\} \\ \textbf{and} \ x_1 <_{\mathbb{TERM}} \cdots <_{\mathbb{TERM}} x_k \\ \textbf{and} \ \lhd_\varphi^X \neq \emptyset \\ \textbf{and} \ X \subseteq \mathsf{Free}(\varphi) \end{bmatrix} \\ \varphi \qquad\qquad \textbf{otherwise} \end{cases}$$

where R denotes the smallest relation induced by the following rule:

$$\frac{\begin{array}{l} \pi = \textsc{Algorithm2}(\varphi, X, \sqsubset_\varphi^X) \\ \textbf{and} \ \sigma \in \mathsf{Final}(\pi, [\![\bigwedge\{x = x \mid x \in X\}]\!]) \\ \textbf{and} \ \sigma(x_1), \ldots, \sigma(x_k) \in \mathbb{D} \end{array}}{(\sigma(x_1), \ldots, \sigma(x_k)) \in R} \tag{7.32}$$

Note that σ in Rule 7.32 may map also data variables outside $\mathsf{Free}(\varphi)$. This happens, for instance, with data constraints with existential quantifiers. The data commands for such data constraints explicitly assign values to quantified data variables, even though those variables do not qualify as free. Because $\{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ contains the free data variables in φ, however, the additional data variables mapped by σ cannot affect the truth of φ (i.e., generally, $\models$ satisfies monotonicity of a data constraint ϕ in data states whose domain contains at least the free data variables in ϕ).

I define commandification in constraint automata in terms of comm.

Definition 58 (commandification)**.** $(\!|\cdot|\!) : \mathbb{AUTOM} \to \mathbb{AUTOM}$ *denotes the function defined by the following equation:*

$$\begin{array}{l} (\!|(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0))|\!) = \\ \quad (Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, (\!|\longrightarrow|\!), (q^0, \mu^0)) \end{array}$$

where $(\!|\longrightarrow|\!)$ denotes the smallest relation induced by the following rules:

$$\frac{\begin{array}{l} q \xrightarrow{P,\phi} q' \ \textbf{and} \ \phi \in \mathbb{DC}_{\exists,\wedge} \\ \textbf{and} \ X^{\text{init}} = P^{\text{in}} \cup {}^\bullet M \end{array}}{q \ (\!|\xrightarrow{P,\mathsf{comm}(\phi,\mathsf{Free}(\phi)\cap X^{\text{init}})}|\!) \ q'} \tag{7.33}$$

$$\frac{q \xrightarrow{P,\phi} q' \ \textbf{and} \ \phi \notin \mathbb{DC}_{\exists,\wedge}}{q \ (\!|\xrightarrow{P,\phi}|\!) \ q'} \tag{7.34}$$

Before I can actually adopt the compilation approach in Figure 7.1 in practice, I must establish the correctness and effectiveness of commandification. I consider commandification correct if it yields a behaviorally congruent—hence, behaviorally equivalent by Theorem 1—constraint automaton to the

original one. Before formulating this property as a theorem, the following lemma states the equivalence of a data constraint and its commandification.

Lemma 18. $\varphi \equiv \mathsf{comm}(\varphi, X)$

From Lemma 18, I conclude the following correctness theorem.

Theorem 20. $\boldsymbol{a} \simeq (\!|\boldsymbol{a}|\!)$

Note that this theorem works not only for normalized constraint automata but also for arbitrary ones. Normalization plays a role only in the sequel, where I show that commandification has its intended effect when applied to normalized constraint automata.

I consider commandification effective if, after commandifying a normalized constraint automaton $|\boldsymbol{a}|$, every data constraint in this automaton either encodes a data command as in Definition 57 or has no data variables in it (in which case a compiler can statically check that data constraint). Generally, however, such unconditional effectiveness does not hold true. After all, if the B-graph for a data constraint φ in $|\boldsymbol{a}|$ has has no $\bigstar$-arborescence, commandification has no strict precedence relation to run Algorithm 2 with. In that case, $\mathsf{comm}(\varphi, X) = \varphi$, and consequently, commandification does not have its intended effect. Fortunately, commandification does satisfy a weaker—but still useful—form of effectiveness. To formulate this as a theorem, I first define a relation that holds true of *arborescent* constraint automata. I consider a constraint automaton arborescent if the B-graph for each of its data constraints has a $\bigstar$-arborescence.

Definition 59 (arborescentness)**.** $\clubsuit \subseteq \mathbb{A}\mathrm{UTOM}$ *denotes the smallest relation induced by the following rule:*

$$\frac{\left[\phi \in \mathsf{Dc}(\boldsymbol{a}) \ \textbf{implies} \ \lhd_{\phi}^{X} \neq \emptyset\right] \ \textbf{for all} \ \phi}{\clubsuit\, \boldsymbol{a}} \tag{7.35}$$

Note that by the previous definition, implicitly, an arborescent constraint automaton has only normal data constraints (otherwise, $\lhd_{\phi}^{X}$ does not exist for at least one $\phi \in \mathsf{Dc}(\boldsymbol{a})$ in Rule 7.35). The following theorem states the effectiveness of commandification, conditional on arborescentness: after commandifying an arborescent—hence normalized—constraint automaton $\boldsymbol{a}$, every data constraint in this automaton encodes a data command as a data relation (as in Definition 57). Let R range over the set of data relations defined in Definition 57 of comm.

Theorem 21. $\clubsuit\, \boldsymbol{a}$ **implies** $\mathsf{Dc}((\!|\boldsymbol{a}|\!)) \subseteq \{R(x_1, \ldots, x_k) \mid \mathbf{true}\}$

A FOCAML compiler can check constraint automata for arborescentness before applying commandification. To reduce the number of such checks, however, I also present a conjecture about preservation of arborescentness by operations on constraint automata. Before stating this conjecture, I first show

that multiplication generally does not preserve arborescentness. For instance, both $\mathsf{BinOp}\langle\mathtt{add}\rangle(\mathsf{A},\mathsf{B};\mathsf{C})$ and $\mathsf{Sync}(\mathsf{C};\mathsf{B})$ individually satisfy arborescentness, but their product does not. To see this, observe that the single transition in this product has $\mathtt{add}(\mathsf{A},\mathsf{B}) = \mathsf{C} \wedge \mathsf{C} = \mathsf{B}$ as its data constraint. One can easily verify that the B-graph for this data constraint indeed contains no arborescence, as one may have already expected: if a commandification of this data constraint *would* exist (which satisfaction of arborescentness would imply), that data command effectively assigns $\mathtt{add}(\mathsf{A},\mathsf{B})$ to B itself. Commandification, however, does not support such self-dependencies. This example instantiates a well-known problem in the theory and practice of synchronous systems: *causality loops*, "where the input is not known until the output is known, and the output can't be known until the input is known" [LNZ14]. In the previous example, Sync closes a causality loop between ports C and D. My conjecture about preservation of arborescentness by operations on constraint automata therefore states that multiplication preserves arborescentness only in the absence of causality loops.

Conjecture 2.

- $\big[\clubsuit\, \boldsymbol{a}_1, \boldsymbol{a}_2$ **and** $\boldsymbol{a}_1 \otimes \boldsymbol{a}_2$ *has no causality loops*$\big]$ **implies** $\clubsuit\, \boldsymbol{a}_1 \otimes \boldsymbol{a}_2$
- $\big[\clubsuit\, \boldsymbol{a}$ **and** $p \notin \mathsf{Input}(\boldsymbol{a})\big]$ **implies** $\clubsuit\, \boldsymbol{a} \ominus p$
- $\big[\clubsuit\, \boldsymbol{a}$ **and** $p \notin \mathsf{Input}(\boldsymbol{a})\big]$ **implies** $\clubsuit\, \boldsymbol{a} \oplus p$

For now, I leave these preservation properties as a conjecture, because its truth or falsehood does not matter much in practice: although its (dis)proof would yield more insight in the theory of commandification, practical consequences remain insignificant. After all, this conjecture helps only in *predicting* when commandification may have its intended effect; it does not affect commandification's correctness whatsoever. Still, if this conjecture indeed holds true as I strongly suspect, a FOCAML compiler can accurately predict whether a product of primitives satisfies arborescentness, based on the arborescentness of these primitives, without again having to check this product for arborescentness (assuming that this product has no causality loops).

Before I did the work presented in this chapter, Clarke et al. already worked on purely constraint-based implementations of protocols [CPLA11]. Essentially, Clarke et al. specify not only the transition labels of an automaton as boolean constraints but also its state space and transition relation. In recent work, Proença and Clarke developed a variant of compile-time *predicate abstraction* to improve performance [PC13a]. They also used this technique to allow a form of interaction between a constraint solver and its environment during constraint solving [PC13b]. The work of Proença and Clarke resembles my work in the sense that we all try to "simplify" constraints at compile-time. I see also differences, though: (i) commandification fully avoids constraint solving and (ii) I adopted a richer language of data constraints in this thesis. For instance, Proença and Clarke have only unary functions in their language, which

would have cleared my need for B-graphs.

7.2 Practice

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

Compiler

I extended Lykos with the ability to apply commandification as in Figure 7.1, controllable through flag `COMMANDIFY`. When raised, Lykos first checks whether all primitives in the core set satisfy arborescentness. If so, by Conjecture 2, Lykos does not need to check products for arborescentness during the compilation process (as long as it encounters no causality loops). Note that even if Conjecture 2 turns out not to hold true, Lykos does not generate faulty code: in the worst case, Lykos unsuccessfully tries to commandify a constraint automaton, but once such commandification fails, Lykos simply defaults to the original data constraint. If this happens, compilation just takes a little longer. In contrast to Lykos's internals, the run-time library requires no modifications. Generated code looks a bit different if commandification succeeds, though, because in that case, Lykos has injected little pieces of sequential Java code—syntactically very similar to data commands—instead of calls to a constraint solver.

Experiments I: Protocols

I repeated the same experiments as in Chapter 6 (and Chapters 4 and 5), generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the `COMMANDIFY`-flag raised, but otherwise under the same conditions as in Chapter 6. Figure 7.10 shows the per-family experimental results, averaged over five runs. The solid lines represent the actual measurements; the dotted lines represent inverse-proportional growth with respect to $k = 1$. The green lines represent the new results; the yellow lines represent the results from Chapter 6.

Figure 7.11 shows per-family speedup charts corresponding to the measurements in Figure 7.10; the dotted lines represent equal performance. For all constraint automata with which I experimented, except members of Merger and Router, commandification indeed improves performance, to a lesser or to a greater extent. The code generated for Mergers and Routers has, overall, similar performance with and without commandification (because these constraint automata have very simple data constraints, which cost relatively few computational resources compared to their nondeterministic choice among increasingly many options). Because of the scale on the y-axis, the speedup for members of OddFibonacci seems almost nonexistent in Figure 7.10, but in fact, Figure 7.11 shows that speedup ranges from 28% (for $k = 64$) up to 275%

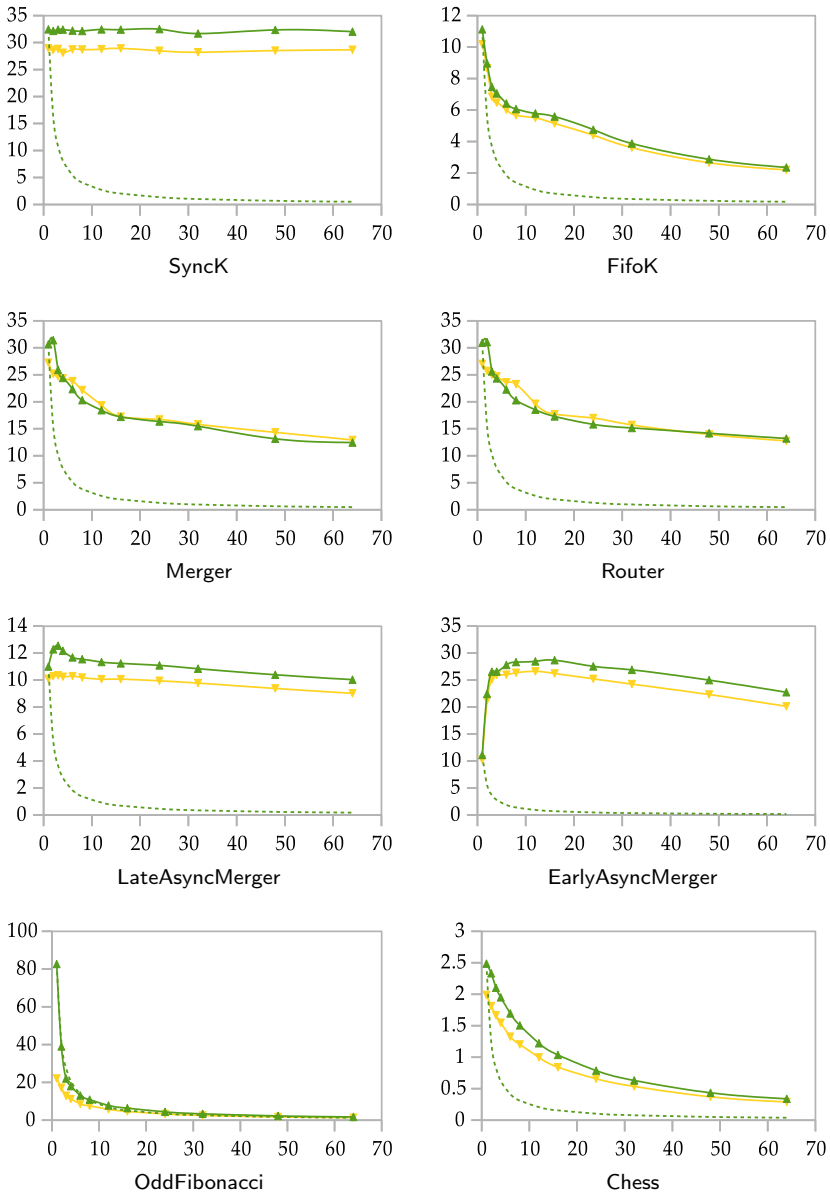

Figure 7.10: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

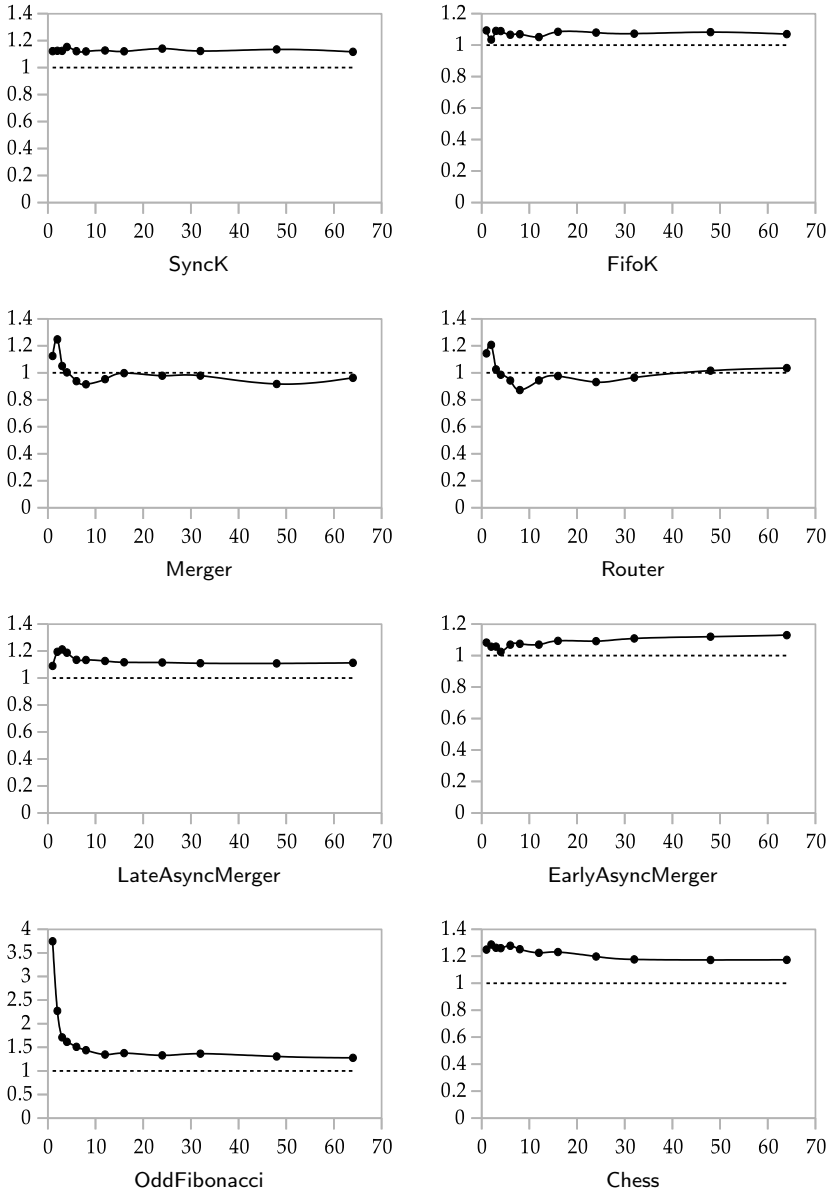

Figure 7.11: Speedup (relative to compiler-generated code in Chapter 6) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

(for $k = 1$). This makes commandification actually most effective for OddFibonacci (because members of OddFibonacci require relatively complex data processing).

Experiments II: Programs

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

Figures 7.12–7.19 show performance charts for the FOCAML-to-Java-compiled versions of the NPB kernel benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The dotted green/yellow lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks with and without commandification (i.e., the green lines represent the new results, while the yellow lines represent the results in Chapter 6); the solid green/yellow lines represent the MasterSlavesInteractionPatternB-based FOCAML-to-Java-compiled versions; the dotted black lines represent the Java versions by Frumkin et al.

I make the following main observations about these 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 lines versus dotted lines), as in Chapters 5 and 6. Furthermore, the FOCAML-to-Java-compiled versions with commandification perform at least as well as the FOCAML-to-Java-compiled versions without commandification (green lines versus yellow lines), and often better by a small margin.

- Although the Java versions of the NPB kernel benchmarks by Frumkin et al. still slightly outperform many of their FOCAML-to-Java-compiled versions, the margin has further decreased (compared to the results in Chapter 6), albeit by only a little: after syntactic subtraction, commandification does not make as big an impact.

- The same point about cache misses made in Chapter 5 applies here too: numbers of cache misses seem a fair indicator of performance.

- The same point about increasing problem sizes made in Chapter 5 applies here too: as the problem size increases, the speedup generally improves.

- As in Chapters 5 and 6, differences in numbers of cache misses explain why the FOCAML-to-Java-compiled versions of NPB-IS without commandification outperform their supposedly improved FOCAML-to-Java-compiled versions with commandification for $k = 64$ in class W, class A, and class B.

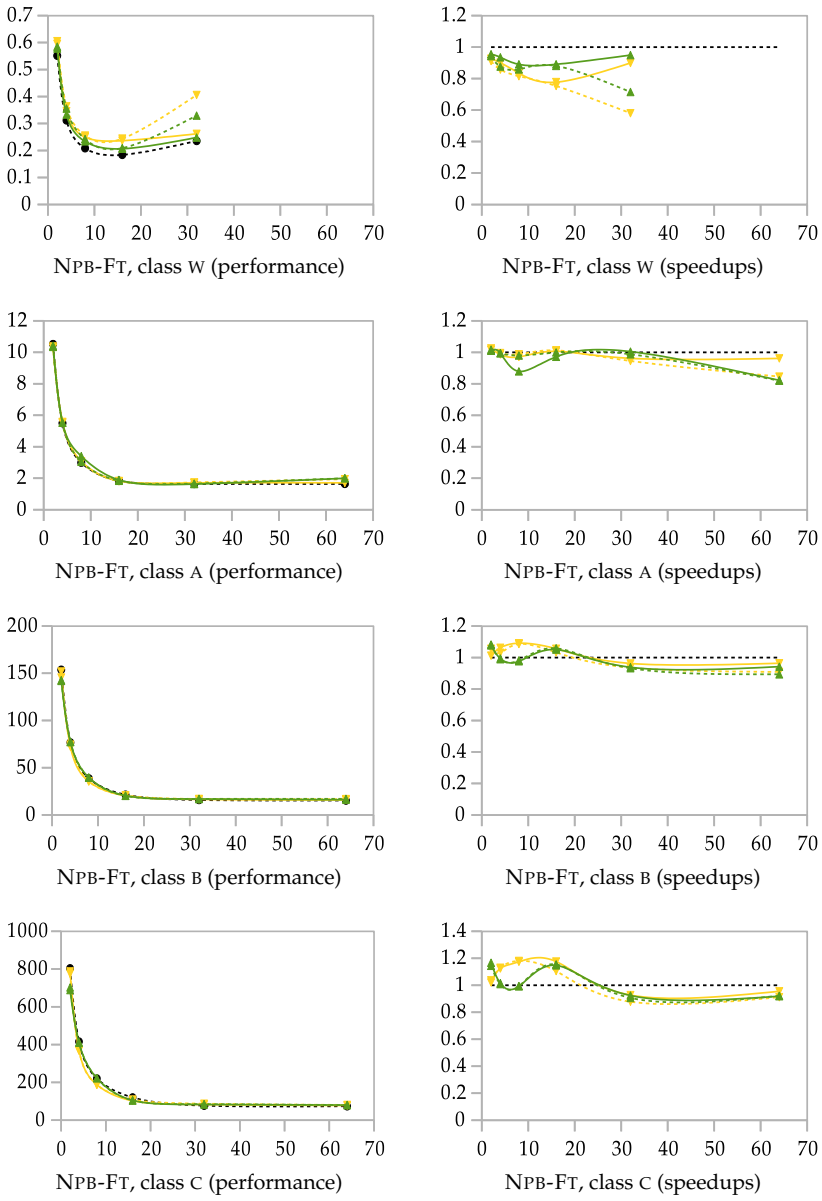

Figure 7.12: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

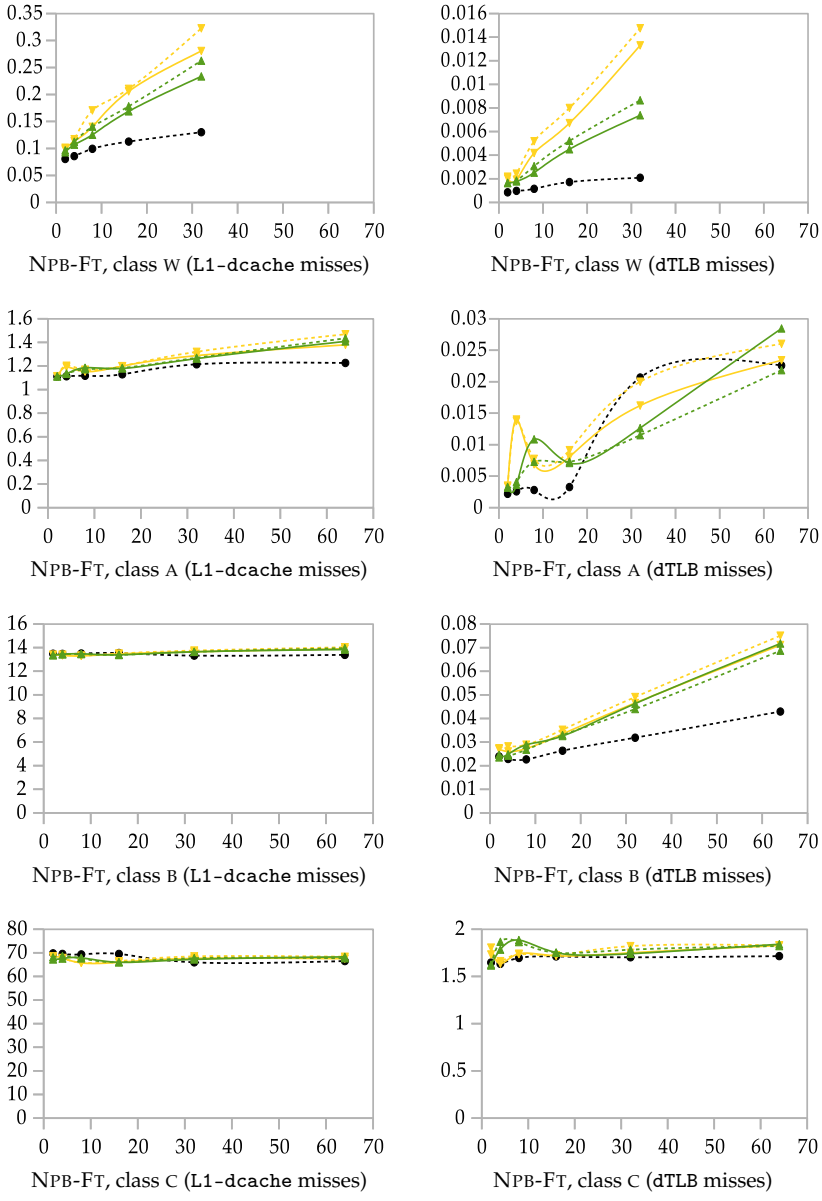

Figure 7.13: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

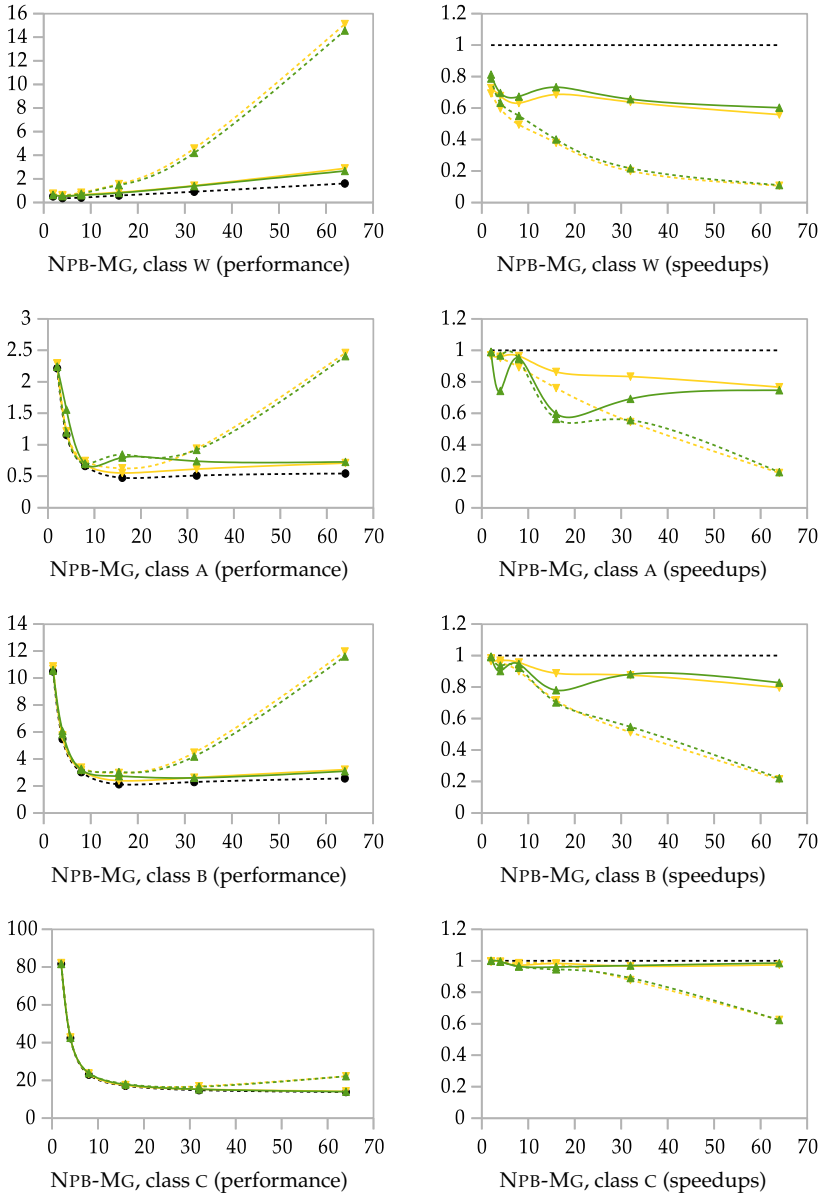

Figure 7.14: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

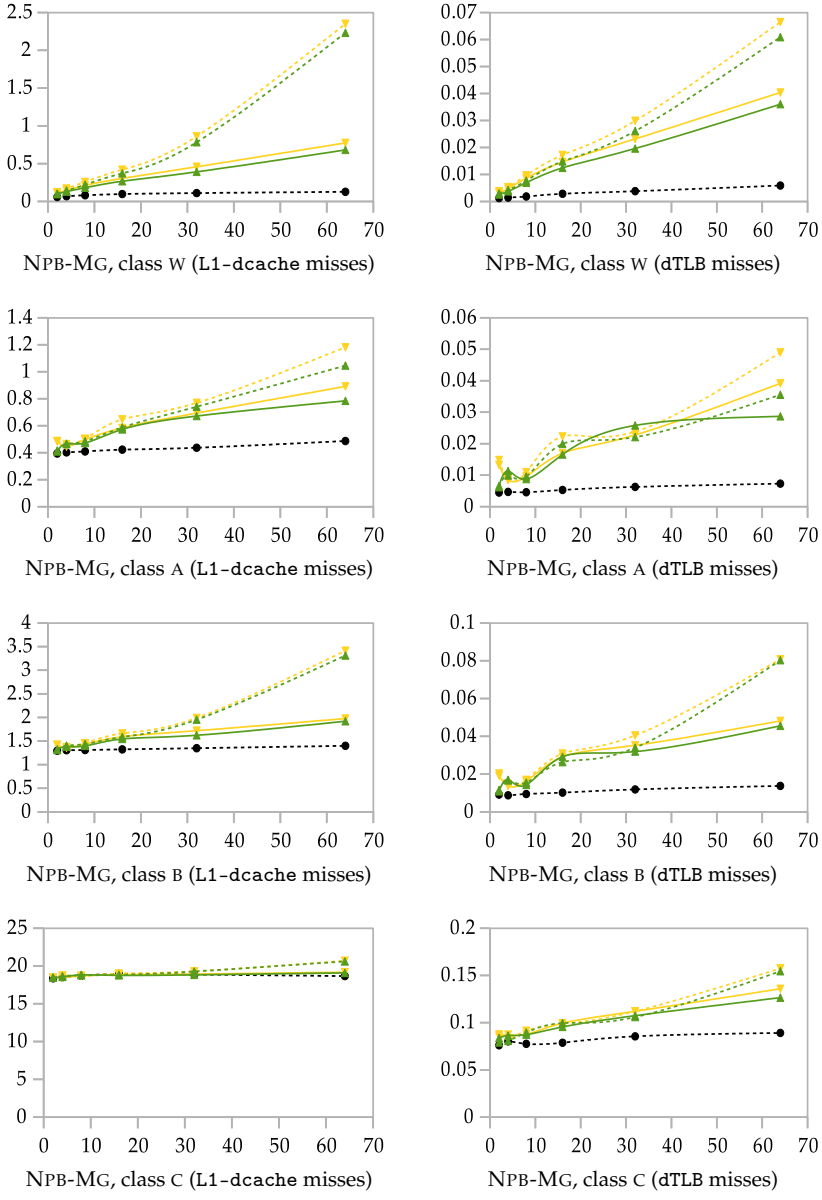

Figure 7.15: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

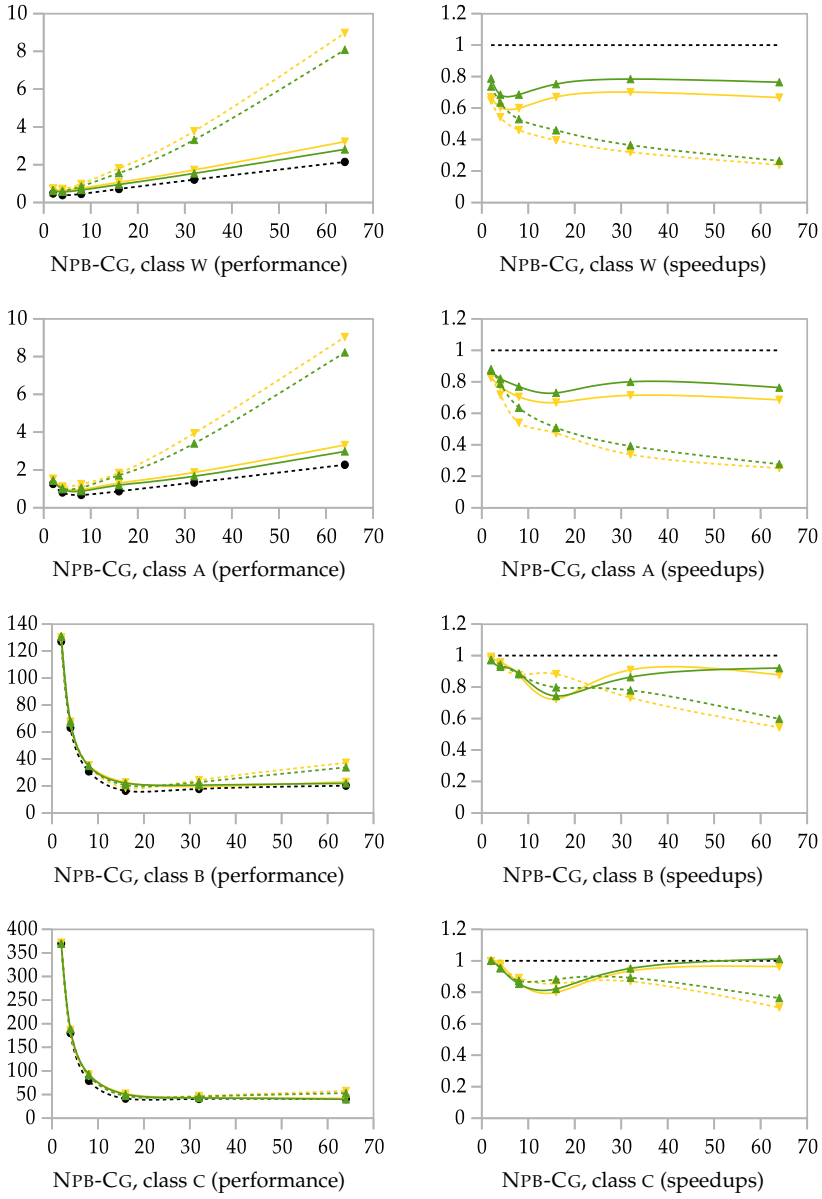

Figure 7.16: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

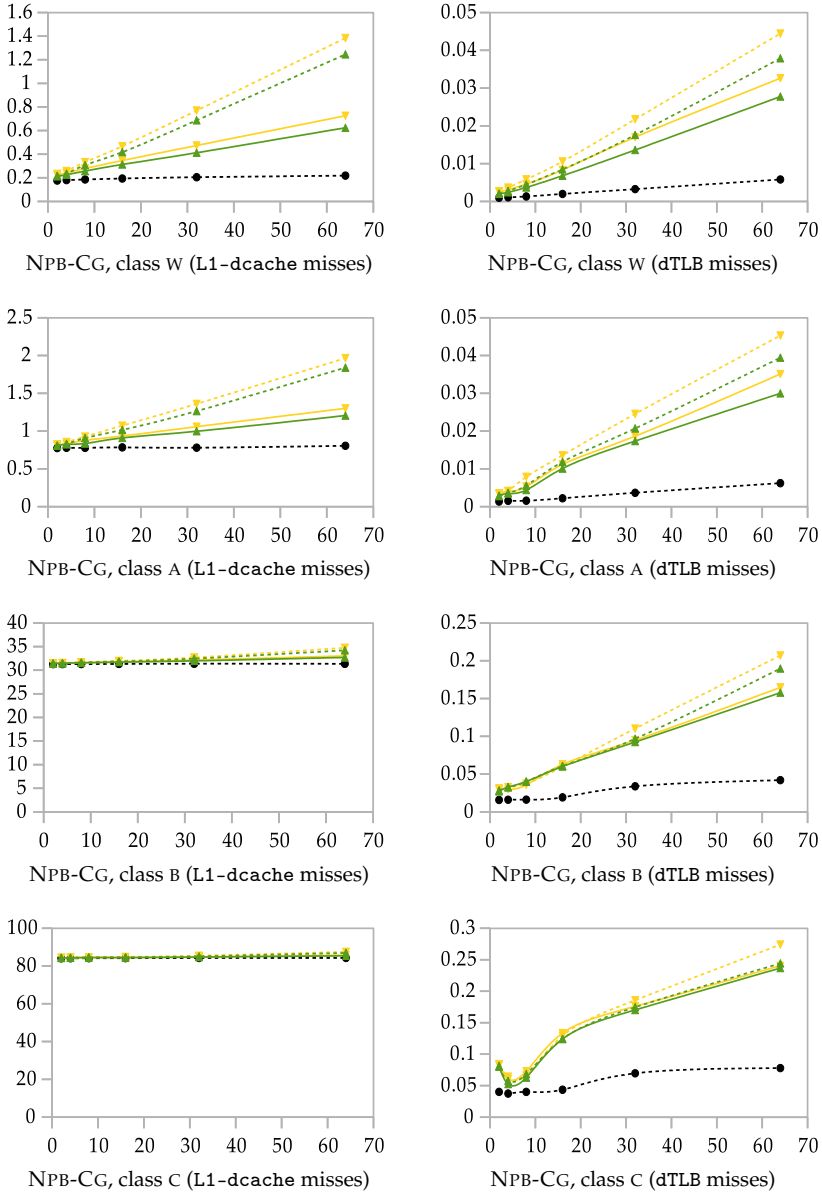

Figure 7.17: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

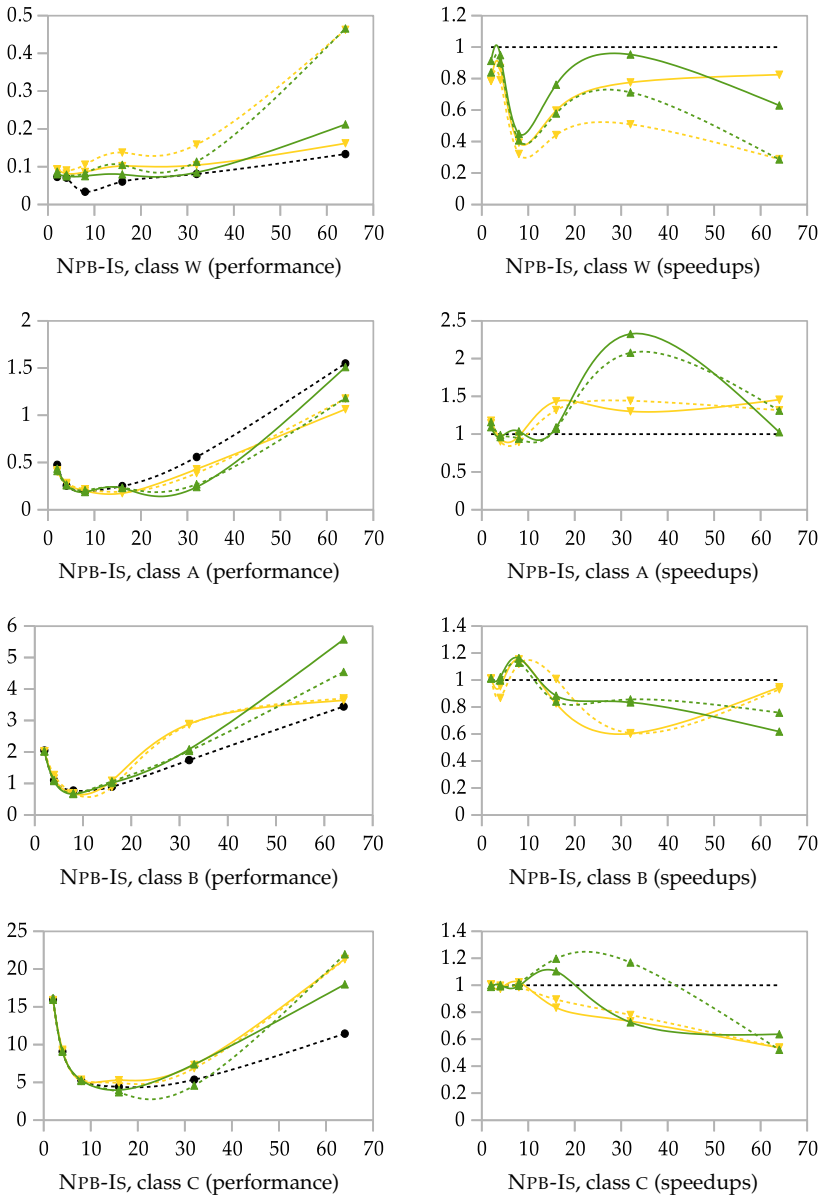

Figure 7.18: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

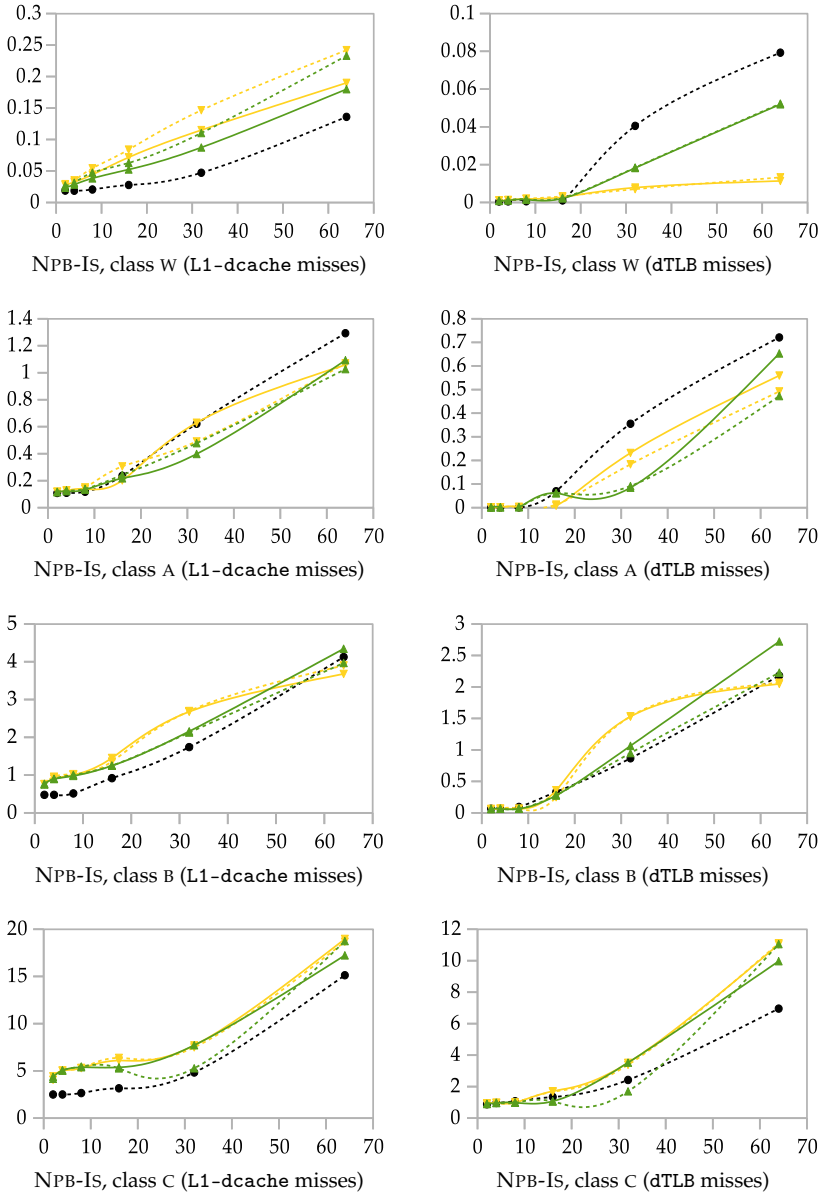

Figure 7.19: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

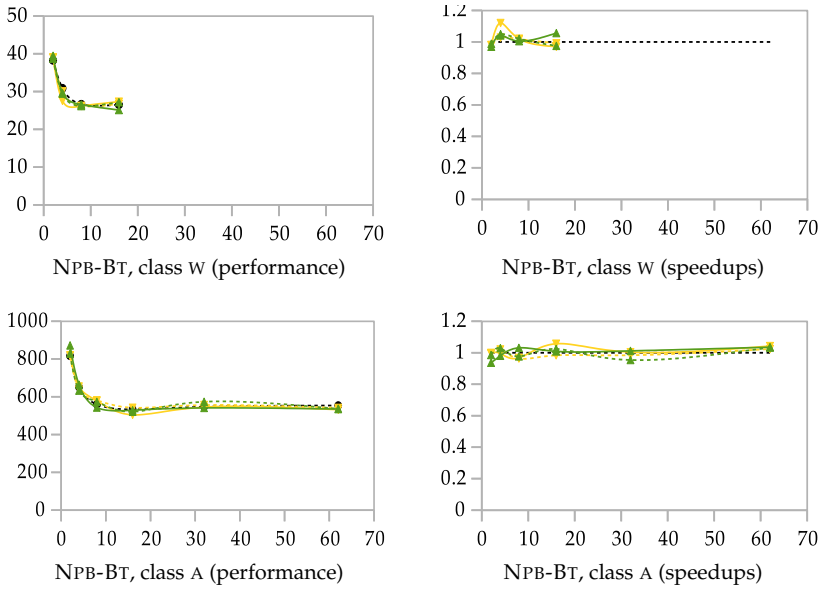

Figure 7.20: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

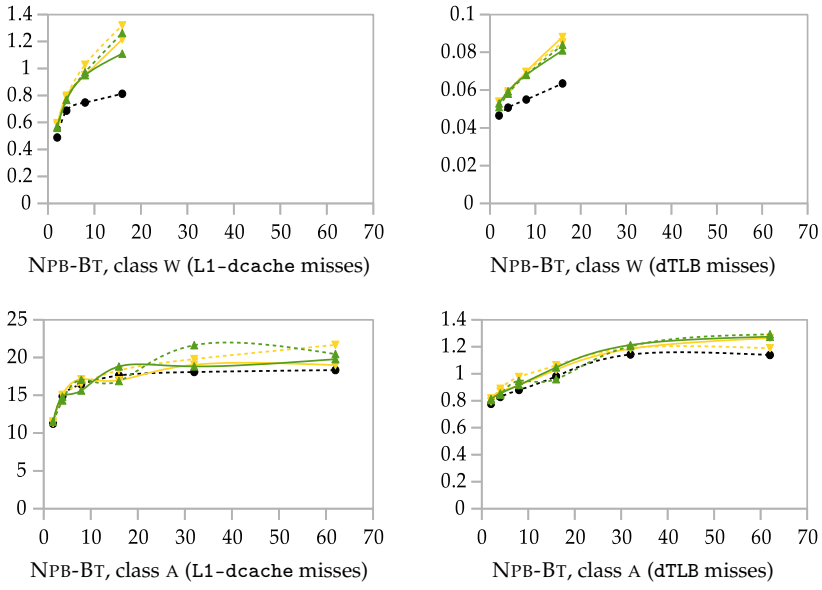

Figure 7.21: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

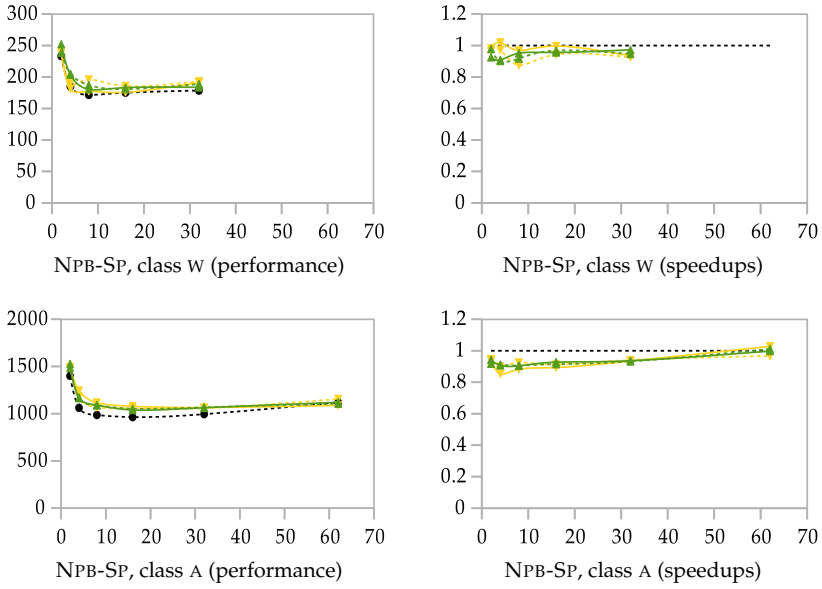

Figure 7.22: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

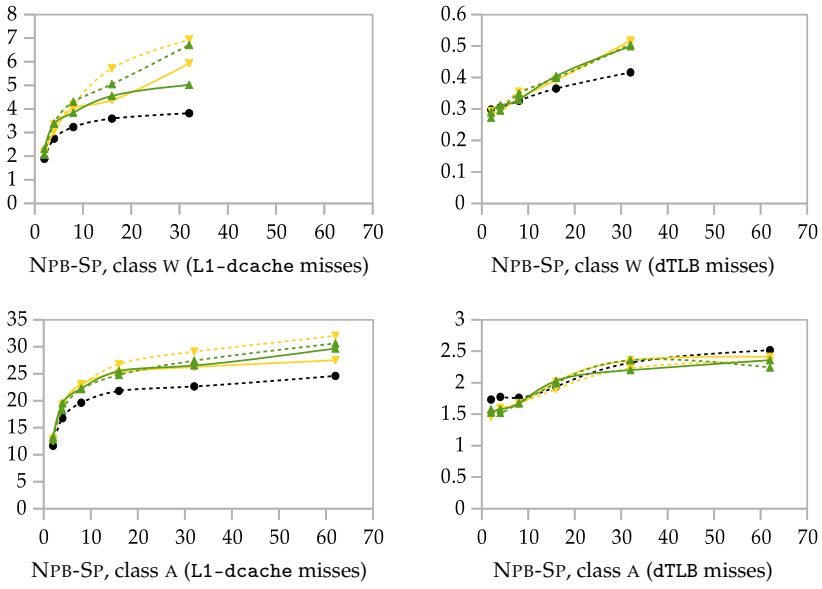

Figure 7.23: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

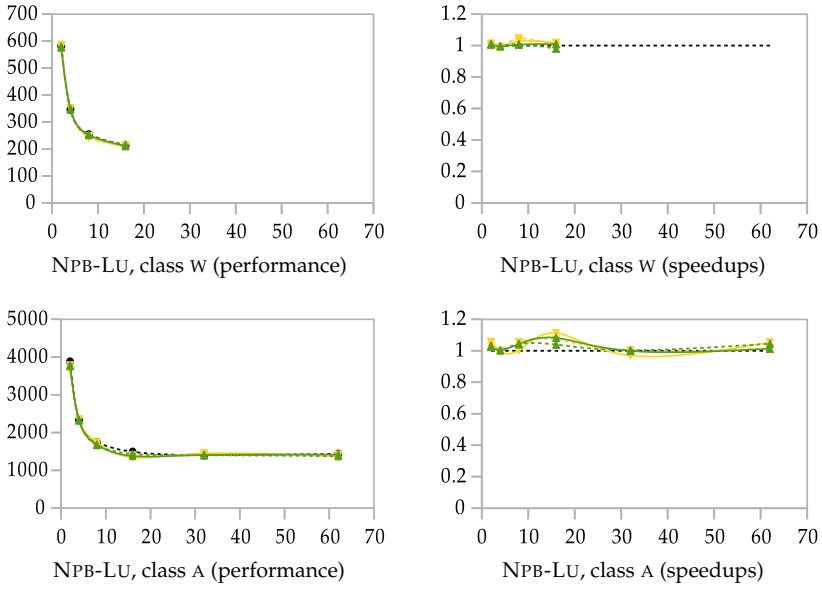

Figure 7.24: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

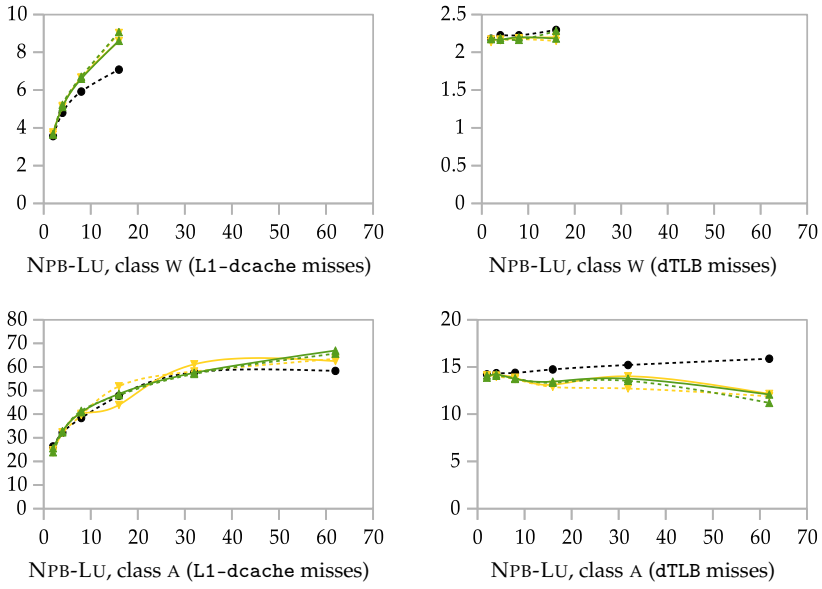

Figure 7.25: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

Figures 7.20–7.25 show performance charts for the FOCAML-to-Java-compiled versions of the NPB application benchmarks (averaged over five runs), speedup charts (with respect to their Java versions by Frumkin et al.), and charts about cache misses. The lines have the same meaning as in the figures with experimental results for the NPB kernel benchmarks. Recall from Figure 5.12 that NPB-BT and NPB-LU do not support more than 22 and 31 slaves, for which reason I have no measurements beyond $k = 16$ in class W for those benchmarks. In the same figure, note that NPB-BT, NPB-SP, and NPB-LU support at most 62 workers in class A. For that reason, as in Chapter 6, I compiled the FOCAML versions of those benchmarks for $k = 62$ instead of $k = 64$. Essentially, the same observations apply here as for the previous experimental results of the NPB kernel benchmarks.