Automata-theoretic protocol programming

Jongmans, Sung-Shik Theodorus Quirinus

Chapter 6

Improved Compilation II: Syntactic Subtraction

In Chapters 4 and 5, I presented a basic FOCAML compiler and reported on a technique for solving scalability problems of the Centralized Approach. Essentially, the Hybrid Approach presented in Chapter 5 tries to balance sequentiality *inside* protocol units with parallelism *among* those units. However, every individual protocol unit still executes purely sequential code and, as such, may form a bottleneck, potentially slowing down the entire program-in-execution.

In this chapter and the next, I present improvements to one of the more computationally costly aspects of protocol units' execution: the data constraint checks involved in determining whether a transition can fire and, in particular, the expensive constraint solver calls made during such checks. More precisely, in this chapter, I study a way of reducing the size of data constraints at compile-time to reduce the overhead of constraint solving at run-time. In Section 6.1, to motivate the need for this reduction, I first discuss a so far disregarded deficiency of the current compilation approach, related to the size of data constraints. Subsequently, I introduce an auxiliary operation on constraint automata, namely *normalization*, primarily to simplify subsequent technicalities. Finally, I introduce a new subtraction to carry out data constraint reduction and formally compare it to the old subtraction in Definition 30. In Section 6.2, I present an improved version of Lykos using this reduction technique for data constraints, including new experimental results on performance.

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

6.1 Theory

(I have not yet published the material in this section.)

64 Syncs

Recall the Sync family defined in Figure 3.4. An undergraduate student of Computer Science may quite straightforwardly prove that members of Sync behave as a kind of algebraic identity of multiplication and subtraction, in the following sense. Let $\boldsymbol{a}$ range over the set of all constraint automata that have an input port p_2.

$$\mathsf{Behav}((\mathsf{Sync}(p_1;p_2)\otimes\boldsymbol{a})\ominus p_2) = \left\{ w' \;\middle|\; \begin{array}{l} w \in \mathsf{Behav}(\boldsymbol{a}) \\ \textbf{and } \left[w'(i) = w(i)|_{\mathrm{Dom}(w(i))\setminus\{p_2\}} \cup \{p_1 \mapsto w(i)(p_2)\} \textbf{ for all } i\right] \end{array} \right\}$$

In words, $(\mathsf{Sync}(p_1;p_2)\otimes\boldsymbol{a})\ominus p_2$ and $\boldsymbol{a}$ have language equivalent behavior modulo substitution of p_1 for p_2. Generally, one can "prefix" (i.e., multiply on its input ports) or "suffix" (i.e., multiply on its output ports) any number of Syncs to a constraint automaton without affecting—in the sense just described—that automaton's behavior. Given this property, it seems not unreasonable to assume that compiler-generated code for a single Sync has the same performance as a sequence of 64 Syncs. Slightly more formally, if $\sim$ means "has the same performance", one may expect:

$$\mathsf{Sync}(p_1;p_{65}) \sim (\mathsf{Sync}(p_1;p_2)\otimes\cdots\otimes\mathsf{Sync}(p_{64};p_{65}))\ominus p_2\ominus\cdots\ominus p_{64}$$

Sensible as this supposition may seem, as the experimental results in Chapter 4 show, code generated by Lykos violates this property: Sync_1 completes 27 million rounds in four minutes, whereas Sync_{64} completes only nine million rounds in the same amount of time (Figure 4.26). Indeed, although the performance of Sync_k stays above the critical threshold of inverse-proportionality, I can hardly claim that the compiler-generated code for Sync_k scales well in k. Thus, as I already stated before, inverse-proportionality forms a necessary condition for good scalability but not necessarily a sufficient one; I discuss inverse-proportionality for the other families of constraint automata with which I experimented so far in Section 6.2.

To better understand the phenomenon at hand, take another look at Definition 30 of $\ominus$. Whereas subtraction eliminates ports from synchronization constraints *syntactically*—effectively making those sets smaller—it removes ports from data constraints only *semantically*. Indeed, $\ominus$ does not reduce the size of data constraints (in terms of the number of data variables, data literals, and existential quantifications) but, in fact and in contrast, makes data constraints larger by enveloping them in existential quantifications: the transition of the single Sync has just $p_1 = p_{65}$ as its data constraint, whereas the corresponding transition in the product of 64 Syncs has $\exists p_{64}.\cdots\exists p_2.(p_1 = p_2 \wedge \cdots \wedge p_{64} = p_{65})$.

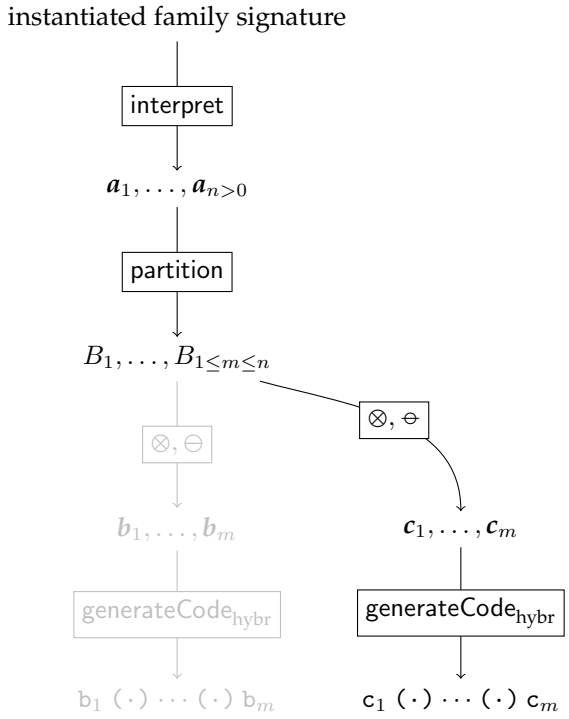

Figure 6.1: Hybrid compilation approach with syntactic subtraction

Clearly, solving the latter data constraint requires more resources than the former.

In the rest of this section, I develop a syntactic subtraction that, when applied 63 times to the product of 64 Syncs, yields the same data constraint as the one in the single Sync. Figure 6.1 shows the resulting compilation approach. First, I need a normalization for bringing constraint automata into a syntactically more convenient shape.

Normalization

Normalization consumes a constraint automaton $\boldsymbol{a}$ and produces a constraint automaton as output. Normalized constraint automata simplify proving certain properties. To normalize a constraint automaton, I manipulate only its data constraints, possibly splitting them into parts and distributing those parts over multiple transitions. Afterward, a normalized constraint automaton contains only *normal data constraints*. Every normal data constraint occurs in *prenex normal form* [Rau10a], in which zero or more existential quantifications envelop a quantifier-free *kernel*. The kernel of a normal data constraint consists of a conjunction of data literals.

Definition 38 (normal data constraints)**.** *A normal data constraint is an object generated by the following grammar:*

$$\begin{array}{lll} \ell & ::= & \textit{any data literal from Definition 15} \\ \varphi & ::= & \exists x.\varphi \mid \ell_1 \wedge \cdots \wedge \ell_{k\geq 1} \qquad \textit{(normal data constraints)} \end{array}$$

$\mathbb{DC}_{\exists,\wedge}$ *denotes the set of all normal data constraints.*

Clearly, $\mathbb{DC}$ in Definition 15 subsumes $\mathbb{DC}_{\exists,\wedge}$. Henceforth, let $\mathsf{Liter}(\varphi)$ denote the set of data literals in φ's kernel.

To normalize a constraint automaton $\boldsymbol{a}$, I must compute a new transition relation. For every transition (q, P, ϕ, q') in $\boldsymbol{a}$, I first compute a prenex normal form of ϕ, denoted by $\mathsf{pnf}(\phi)$. The goodness that ϕ exhibits by Definition 19 of $\mathbb{A}\text{UTOM}$ guarantees that such a prenex normal form always exists (i.e., the properties of goodness in Definition 17 guarantee that all inner existential quantifiers in ϕ can safely move outward). Next, I compute a *disjunctive normal form* of the kernel χ of $\mathsf{pnf}(\phi)$ [Rau10b], denoted by $\mathsf{dnf}(\chi)$. For every disjunct $\ell_1 \wedge \cdots \wedge \ell_k$ in $\mathsf{dnf}(\chi)$, I subsequently construct a new transition from state q to state q' labeled by synchronization constraint P and the normal data constraint built out of the existential quantifications in $\mathsf{pnf}(\phi)$ and $\ell_1 \wedge \cdots \wedge \ell_k$. Constructing new transitions in this way preserves the semantics of the original transition, because (i) prenex/disjunctive normal forms preserve the semantics of ϕ, (ii) existential quantification distributes over disjunction, and (iii) multiple transitions out of the same state represent a disjunction.

Definition 39 (normalization)**.** $|\cdot| : \mathbb{A}\text{UTOM} \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

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

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

$$\frac{q \xrightarrow{P,\phi} q' \ \textbf{and} \ \mathsf{pnf}(\phi) = \exists x_1.\cdots\exists x_l.\chi \ \textbf{and} \ \mathsf{dnf}(\chi) = \phi_1 \vee \cdots \vee \phi_k}{q \ |\xrightarrow{P,\exists x_1.\cdots\exists x_l.\phi_{1\leq i\leq k}}| \ q'} \tag{6.1}$$

The following theorem states the correctness of Definition 39 of $|\cdot|$: normalization of a constraint automaton yields a behaviorally congruent constraint automaton. (Consequently, because $\simeq$ implies $\approx$ by Theorem 1, a normalized constraint automaton accepts the same language as its original.)

Theorem 12. $\boldsymbol{a} \simeq |\boldsymbol{a}|$

The following theorem states the effectiveness of Definition 39: a normalized constraint automaton indeed contains only normal data constraints.

Theorem 13. $\mathsf{Dc}(|\boldsymbol{a}|) \subseteq \mathbb{DC}_{\exists,\wedge}$

Syntactic Subtraction

I now proceed by defining syntactic substitution. I consider only (constraint automata with) normal data constraints, without loss of generality (because I can normalize every constraint automaton to a behaviorally congruent one).

First, I need to introduce the concept of *determinants* of free data variables in normal data constraints. For a normal data constraint φ and one of its free data variables $x \in \mathsf{Free}(\varphi)$, the set of determinants of x consists of those terms that *precisely* determine the datum $\sigma(x)$ assigned to x in any data assignment σ that satisfies φ (i.e., $\sigma \models \varphi$). "Precisely" here means that a determinant neither overspecifies nor underspecifies $\sigma(x)$. Thus, if a set of determinants contains multiple data terms, each of those data terms evaluates to the same datum under σ. Determinants furthermore determine $\sigma(x)$ independent of x itself: no determinant of x has x among its free data variables (i.e., determinants have no recursion). In the following definition, recall that a stands for a negated data atom in Definition 15.

Definition 40 (determinants)**.** $\mathsf{Determ} : \mathbb{X} \times \mathbb{DC}_{\exists,\wedge} \to 2^{\mathbb{TERM}}$ *denotes the function defined by the following equations:*

$$
\begin{aligned}
&\mathsf{Determ}_x(\top), \mathsf{Determ}_x(\bot) = \emptyset \\
&\mathsf{Determ}_x(\mathtt{K}(M)) = \emptyset \\
&\mathsf{Determ}_x(t_1 = t_2) = \begin{cases} \{t_2\} & \textbf{if } \left[t_1 = x \textbf{ and } x \notin \mathsf{Variabl}(t_2)\right] \\ \{t_1\} & \textbf{if } \left[t_2 = x \textbf{ and } x \notin \mathsf{Variabl}(t_1)\right] \\ \emptyset & \textbf{otherwise} \end{cases} \\
&\mathsf{Determ}_x(R(t_1, \ldots, t_k)) = \emptyset \\
&\mathsf{Determ}_x(\neg a) = \emptyset \\
&\mathsf{Determ}_x(\ell_1 \wedge \cdots \wedge \ell_k) = \mathsf{Determ}_x(\ell_1) \cup \cdots \cup \mathsf{Determ}_x(\ell_k) \\
&\mathsf{Determ}_x(\exists x'.\varphi') = \begin{cases} \mathsf{Determ}_x(\varphi) & \textbf{if } x \neq x' \\ \emptyset & \textbf{otherwise} \end{cases}
\end{aligned}
$$

For instance, let ϕ_{eg} denote the data constraint defined by the following equation (which occurs in the constraint automaton for the instantiated family signature `OddFibonacciPart(A,C;F,H)` in Figure 3.27):

$$\phi_{\text{eg}} = {}^{\bullet}\mathsf{x} = \mathsf{B} \wedge \mathsf{C} = \mathsf{D} \wedge \mathtt{add}(\mathsf{B}, \mathsf{D}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})$$

Because $\mathbb{DC}_{\exists,\wedge}$ contains ϕ_{eg}, I can directly apply Determ to compute the determinants of its free variables:

$$
\begin{aligned}
&\mathsf{Determ}_{{}^{\bullet}\mathsf{x}}(\phi_{\text{eg}}) = \{\mathsf{B}\} \\
&\mathsf{Determ}_{\mathsf{B}}(\phi_{\text{eg}}) = \{{}^{\bullet}\mathsf{x}\} \\
&\mathsf{Determ}_{\mathsf{C}}(\phi_{\text{eg}}) = \{\mathsf{D}\} \\
&\mathsf{Determ}_{\mathsf{D}}(\phi_{\text{eg}}) = \{\mathsf{C}\}
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{Determ}_{\mathsf{E}}(\phi_{\text{eg}}) = \{\mathtt{add}(\mathsf{B}, \mathsf{D}), \mathsf{D}, \mathsf{D}\} \\
&\mathsf{Determ}_{\mathsf{F}}(\phi_{\text{eg}}) = \{\mathsf{E}\} \\
&\mathsf{Determ}_{\mathsf{G}}(\phi_{\text{eg}}) = \{\mathsf{E}\}
\end{aligned}
$$

Let $|\boldsymbol{a}|$ denote a normalized constraint automaton, and let φ denote one of its normal data constraints. Suppose that I subtract x from $|\boldsymbol{a}|$ with $\ominus$. By Definition 30 of $\ominus$, the transition(s) of $|\boldsymbol{a}|$ previously labeled by φ now carry $\exists x.\varphi$. However, if x has determinants, instead of enveloping φ in an existential quantification as $\ominus$ does, I can alternatively perform a syntactic substitution of one of those determinants for x. I formalize such a substitution with the following function.

Definition 41 (syntactic existential quantification)**.** $\mathsf{exists} : \mathbb{X} \times \mathbb{DC}_{\exists,\wedge} \to \mathbb{DC}_{\exists,\wedge}$ *denotes the function defined by the following equations:*

$$\mathsf{exists}_x(\varphi) = \begin{cases} \varphi[t/x] & \textbf{if } \left[\mathsf{Determ}_x(\varphi) \neq \emptyset \textbf{ and } t = \min(\mathsf{Determ}_x(\varphi))\right] \\ \exists x.\varphi & \textbf{otherwise} \end{cases}$$

In this definition, function $\min(\cdot)$ takes the least element in $\mathsf{Determ}_x(\phi)$, under the global order on data constraints $<_{\mathbb{DC}}$, to ensure that exists always produces the same output under the same input. The following equations exemplify the (nested) application of exists on ϕ_{eg} as defined above for all internal ports.

$$\begin{aligned}
&\mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\mathsf{exists}_{\mathsf{B}}(\phi_{\text{eg}})))) \\
&= \mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\mathsf{exists}_{\mathsf{B}}(\\
&\quad {}^{\bullet}\mathsf{x} = \mathsf{B} \wedge \mathsf{C} = \mathsf{D} \wedge \mathtt{add}(\mathsf{B},\mathsf{D}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})))))) \\
&= \mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\mathsf{exists}_{\mathsf{D}}(\\
&\quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{D} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{D}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})))) \\
&= \mathsf{exists}_{\mathsf{G}}(\mathsf{exists}_{\mathsf{E}}(\\
&\quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{C} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{C}) = \mathsf{E} \wedge \mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G}))) \\
&= \mathsf{exists}_{\mathsf{G}}(\\
&\quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{C} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{C}) = \mathsf{F} \wedge \mathsf{F} = \mathsf{F} \wedge \mathsf{F} = \mathsf{G} \wedge \neg\mathtt{Odd}(\mathsf{G})) \\
&= \quad {}^{\bullet}\mathsf{x} = {}^{\bullet}\mathsf{x} \wedge \mathsf{C} = \mathsf{C} \wedge \mathtt{add}({}^{\bullet}\mathsf{x},\mathsf{C}) = \mathsf{F} \wedge \mathsf{F} = \mathsf{F} \wedge \mathsf{F} = \mathsf{F} \wedge \neg\mathtt{Odd}(\mathsf{F})
\end{aligned}$$

I define syntactic subtraction in terms of exists.

Definition 42 (syntactic subtraction)**.** $\ominus : \mathbb{A}\text{UTOM} \times \mathbb{P} \to \mathbb{A}\text{UTOM}$ *denotes the function defined by the following equation:*

$$\begin{aligned}
&(Q, (P^{\text{all}}, P^{\text{in}}, P^{\text{out}}), M, \longrightarrow, (q^0, \mu^0)) \ominus p = \\
&\quad (Q, (P^{\text{all}} \setminus \{p\}, P^{\text{in}} \setminus \{p\}, P^{\text{out}} \setminus \{p\}), M, \longrightarrow_{\ominus}, (q^0, \mu^0))
\end{aligned}$$

where $\longrightarrow_{\ominus}$ *denotes the smallest relation induced by the following rules:*

$$\frac{q \xrightarrow{P,\phi} q' \textbf{ and } \phi \in \mathbb{DC}_{\exists,\wedge}}{q \xrightarrow{P\setminus\{p\},\mathsf{exists}_p(\phi)}_{\ominus} q'} \quad (6.2) \qquad \frac{q \xrightarrow{P,\phi} q' \textbf{ and } \phi \notin \mathbb{DC}_{\exists,\wedge}}{q \xrightarrow{P\setminus\{p\},\exists p.\phi}_{\ominus} q'} \quad (6.3)$$

In the previous definition, I use exists to remove ports from data constraints. Although Definition 41 of exists also allows for removing data variables for memory cells, I do not pursue such syntactic subtraction in this thesis.

Before I can actually adopt the compilation approach in Figure 6.1 in practice, I must establish the correctness and effectiveness of syntactic subtraction. I consider syntactic subtraction correct if it yields a behaviorally congruent—hence, behaviorally equivalent by Theorem 1—constraint automaton to the constraint automaton that semantic subtraction would have yielded. Before formulating this property as a theorem, the following lemma first states the equivalence of existential quantification and exists.

Lemma 16. $\exists x.\varphi \equiv \mathsf{exists}_x(\varphi)$

From Lemma 16, I conclude the following correctness theorem.

Theorem 14. $\boldsymbol{a} \ominus p \simeq \boldsymbol{a} \oplus p$

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 syntactic subtraction has its intended effect when applied to normalized constraint automata.

I consider syntactic subtraction effective if, after syntactically subtracting a port p from a normalized constraint automaton $|\boldsymbol{a}|$, that port no longer occurs in any of that automaton's data constraints. Generally, however, such unconditional effectiveness does not hold true. After all, if $|\boldsymbol{a}|$ has a data constraint φ in which p occurs, but p has no determinants in φ, syntactic subtraction has nothing to replace p with. In that case, $\mathsf{exists}_p(\varphi) = \exists p.(\varphi)$, and consequently, syntactic subtraction does not have its intended effect. Fortunately, syntactic subtraction does satisfy a weaker—but still useful—form of effectiveness. To formulate this as a theorem, I first define a function that computes *ever-determined ports*. Under a set of excluded data terms T, I consider a port p ever-determined in a constraint automaton $\boldsymbol{a}$ iff both p occurs in $\boldsymbol{a}$ and every data constraint in $\boldsymbol{a}$ has a determinant for p outside T.

Definition 43 (ever-determined ports)**.** $\mathsf{Edp} : 2^{\mathbb{TERM}} \times \mathbb{AUTOM} \to 2^{\mathbb{P}}$ *denotes the function defined by the following equation:*

$$\mathsf{Edp}_T(\boldsymbol{a}) = \{p \mid [\begin{bmatrix} p \in \mathsf{Variabl}(\phi) \\ \textbf{and} \;\; \phi \in \mathsf{Dc}(\boldsymbol{a}) \end{bmatrix} \;\; \textbf{implies} \;\; \mathsf{Determ}_p(\phi) \setminus T \neq \emptyset] \;\; \textbf{for all} \;\; \phi\}$$

For instance, p_1, p_2, and p_3 all qualify as ever-determined (under $\emptyset$) in members of Merger2 in Figure 3.4. To understand the ever-determinedness of p_1, observe that p_1 occurs in the data constraint on the top transition in Merger2 and that p_1 has a determinant outside $\emptyset$ in that data constraint (namely p_3); because p_1 does not occur in the data constraint on the bottom transition in Merger2, p_1 indeed qualifies as ever-determined. A similar explanation applies to p_2. To understand the ever-determinedness of p_3, observe that p_3 occurs in the data constraint on both transitions in Merger2 and that p_3 has a determinant outside $\emptyset$ in both these data constraints (namely p_1 and p_2). Consequently, also p_3 qualifies as ever-determined. In contrast, p_2 in members of Filter in Figure 3.4

does *not* qualify as ever-determined (under any set of excluded data terms), because p_2 occurs in the data constraint on the top transition in Filter but does not have a single determinant in that data constraint.

The following theorem states the effectiveness of syntactic subtraction, conditional on ever-determinedness: after syntactically subtracting an ever-determined port from a normalized constraint automaton, that port no long occurs in any of that automaton's data constraints.

Theorem 15.

$$p \in \mathsf{Edp}_T(|\boldsymbol{a}|) \textbf{ implies } p \notin \{x \mid \phi \in \mathsf{Dc}(|\boldsymbol{a}| \ominus p) \textbf{ and } x \in \mathsf{Variabl}(\phi)\}$$

A FOCAML compiler can check ports for ever-determinedness before applying syntactic subtraction. To reduce the number of such checks, however, I also present a conjecture about preservation of ever-determinedness by operations on constraint automata. First, note that for syntactic subtraction of port p' in constraint automaton $\boldsymbol{a}$ to preserve the ever-determinedness of a port p, every data constraint in $\boldsymbol{a}$ should have a non-p' determinant for p. For instance, in E = F, port E has port F as its determinant but not so in $\mathsf{exists}_{\mathsf{F}}(\mathsf{E} = \mathsf{F}) = \mathsf{E} = \mathsf{E}$. Thus, syntactic subtraction of F in a constraint automaton with a transition labeled by E = F does not preserve the ever-determinedness of E. In contrast, syntactic subtraction of F in a constraint automaton with a transition labeled by $\mathsf{E} = \mathsf{F} \wedge \mathsf{E} = \mathsf{G}$ *does* preserve the ever-determinedness of E, because E has a non-F determinant in that data constraint, namely G. The following conjecture makes this precise.

Conjecture 1.

- $p \in \mathsf{Edp}_T(|\boldsymbol{a}_1|) \textbf{ implies } p \in \mathsf{Edp}_T(||\boldsymbol{a}_1| \otimes |\boldsymbol{a}_2||)$
- $p \in \mathsf{Edp}_T(|\boldsymbol{a}|) \textbf{ implies } p \in \mathsf{Edp}_T(|\boldsymbol{a}| \ominus p')$
- $p \in \mathsf{Edp}_{T \cup \{p'\}}(|\boldsymbol{a}|) \textbf{ implies } p \in \mathsf{Edp}_T(|\boldsymbol{a}| \ominus p')$

The first item of this conjecture states that multiplication preserves ever-determinedness; its second and its third item state that also subtraction preserves ever-determinedness under a suitable set T.

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 syntactic subtraction, practical consequences remain insignificant. After all, this conjecture helps only in *predicting* when syntactic subtraction may have its intended effect; it does not affect syntactic subtraction's correctness whatsoever. Still, if this conjecture indeed holds true as I strongly suspect, a FOCAML compiler can reduce its number of checks for ever-determinedness as explained next.

All instances of the primitives in Figure 3.4 have only ever-determined output ports and only ever-determined internal ports (vacuously, because none of these constraint automata has any internal ports). Regardless of whether these

primitives have ever-determined input ports—some of them do, others do not—the ever-determinedness of their output and internal ports already suffices for syntactic subtraction to have its intended effect whenever a FOCAML compiler subtracts only internal ports (as it always does). To see this, recall from Definition 29 of $\otimes$ that new internal ports arise only through multiplication. In particular, the set of internal ports in a product contains (i) the internal ports in its multiplicands and (ii) those multiplicands' shared ports, where every shared port *must* serve as an output port in exactly one multiplicand. Now, if two multiplicands indeed have only ever-determined output ports and only ever-determined internal ports (as all instances of the primitives in Figure 3.4 do), this means two things. First, by Conjecture 1, the product has only ever-determined internal ports, namely the multiplicands' shared output ports and their internal ports. Consequently, syntactic subtraction has its intended effect on all these ports. Second, as the multiplicands, also the product has only ever-determined output ports, namely the multiplicands' unshared output ports. Consequently, this reasoning can recur. Thus, if a FOCAML compiler has established that all primitives in a core set—not necessarily the same core set as in Figure 3.4—have only ever-determined output ports and only ever-determined internal ports, it does not need to check those ports for ever-determinedness again in any of those primitives' products. Symmetrically, this also works for input ports instead of output ports.

Incidentally, if I apply syntactic subtraction to the sequence of 64 Syncs as in the beginning of this chapter, and after removing $x = x$ literals (each of which trivially equates to $\top$), I get *exactly* the same data constraint as the one in the single Sync. Using syntactic subtraction, thus, the sequence of 64 Sync has the same performance as the single Sync, as shown in more detail in Section 6.2.

6.2 Practice

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

Compiler

I extended Lykos with the ability to apply syntactic subtraction as in Figure 6.1, controllable through flag `SUBTRACT_SYNTACTICALLY`. When raised, Lykos first checks whether all primitives in the core set have either only ever-determined output ports or, symmetrically, only ever-determined input ports. If so, by Conjecture 1, Lykos does not need to check ports in products for ever-determinedness during the compilation process. Note that even if Conjecture 1 turns out not to hold true, Lykos does not generate faulty code: in the worst case, Lykos unsuccessfully tries to syntactically subtract p without first checking p for ever-determinedness, but once such subtraction fails (in which case it cannot find a determinant for p), Lykos simply defaults to semantic subtraction, as in Definition 41. If this happens, compilation just takes a little longer. In contrast to Lykos' internals, the run-time library requires no modifications. Also,

code generated with syntactic subtraction looks very similar to code generated without syntactic subtraction, differing only in the size of data constraints (but the constraint solver in the run-time library solves them in exactly the same way).

Experiments I: Protocols

I repeated the same experiments as in Chapter 5 (and Chapter 4), generating code for members of families SyncK, FifoK, Merger, Router, LateAsyncMerger, EarlyAsyncMerger, OddFibonacci, and Chess with the `SUBTRACT_SYNTACTICALLY`-flag raised, but otherwise under the same conditions as in Chapter 5 (except for OddFibonacci, whose members I compiled under the Centralized Approach to manually avoid overparallelization). Figure 6.2 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 yellow lines represent the new results; the blue and red lines represent the results from Chapters 4 and 5. I compare the new results for OddFibonacci with its results in Chapter 4 instead of those in Chapter 5 because of the overparallelization issue discussed at the end of Chapter 5.

Figure 6.3 shows per-family speedup charts corresponding to the measurements in Figure 6.2; the dotted lines represent equal performance. For all constraint automata with which I experimented, syntactic subtraction indeed improves performance, to a lesser or to a greater extent. For members of SyncK, with syntactic subtraction, performance becomes constant in k (i.e., the number of Syncs that constitute a SyncK), exactly as one would expect of a neutral element for multiplication. Speedups grow linearly in k, up to 200% for $k = 64$. For members of FifoK, speedup stays roughly constant in k, at about 3%. Performance itself, in contrast, does not stay constant in k but degrades more or less linearly (from $k = 3$ onward). As for members of SyncK, inverse-proportional growth does not imply good scalability for members of FifoK: in principle, the size of a buffer (controlled by k) should not affect the speed with which data can pass through this buffer. Thus, for compiler-generated code for members of Fifo to scale well, its performance should stay constant in k instead of degrading linearly. In perhaps the simplest approach to achieve such constant performance, a compiler recognizes sequences of k consecutive Fifos in multiplication expressions and subsequently gives such sequences a special treatment, effectively generating special optimized code for k-capacity buffers. Although not a general method, sequences of Fifos occur frequently enough to justify an optimization as this. Nevertheless, I do not pursue this optimization in this thesis—it may constitute an interesting BSc project though.

For members of of LateAsyncMerger and EarlyAsyncMerger (to greater extent) and of Merger (to lesser extent), performance seems to approach constant in k (i.e., the number of producers). Members of those families do not require the producers to synchronize with each other nor with the consumer. Consequently, in principle, the performance of compiler-generated code for members of those families should not depend on the number of producers: a per-

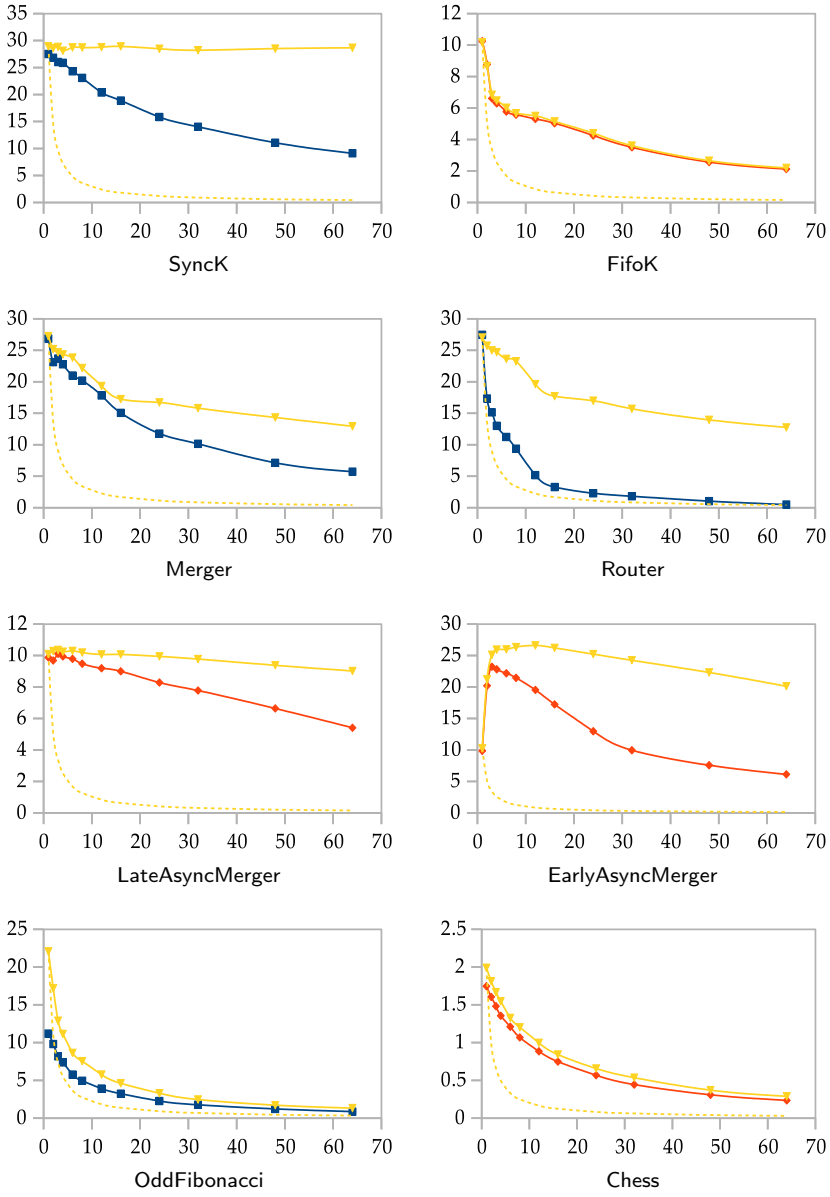

Figure 6.2: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

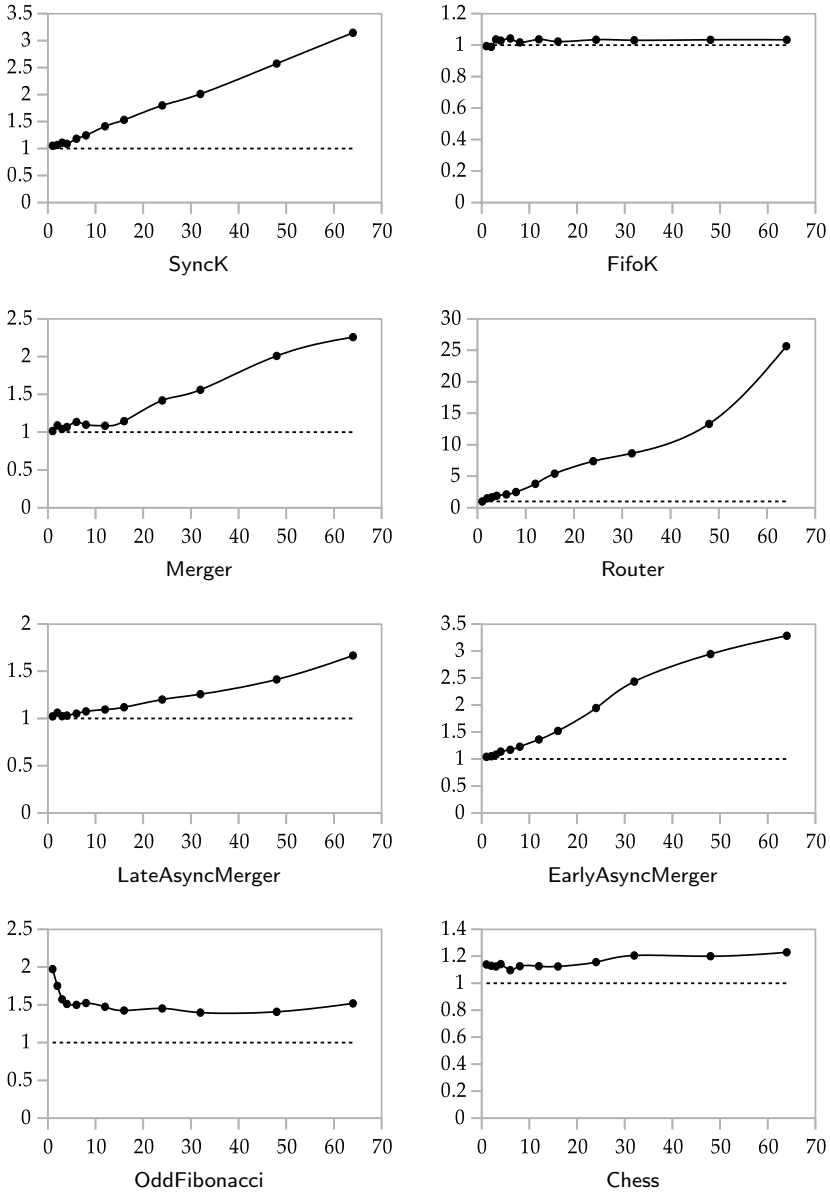

Figure 6.3: Speedup (relative to compiler-generated code in Chapter 5) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

fect compiler generates code with performance constant in k. As Sync and Fifo, thus, Merger, LateAsyncMerger, and EarlyAsyncMerger exemplify that inverse-proportionality forms a necessary condition for good scalability but not necessarily a sufficient one. In practice, however, the number of producers *does* influence the performance of code generated for members of Merger, LateAsyncMerger, and EarlyAsyncMerger, because those producers contend for the same "resource"—the consumer. Therefore, near-constant growth in k seems fair enough. My extension of Lykos with syntactic subtraction makes a significant step in achieving that goal, providing speedup of up to 125% for members of Merger, 67% for members of LateAsyncMerger and up to 228% for members of EarlyAsyncMerger. The improvements that I introduce in Chapters 7 and 8 push scalability even further toward a flat line.

In principle, a similar analysis as for members of Merger, LateAsyncMerger, and EarlyAsyncMerger applies to members of Router. However, Routers achieve truly spectacular speedup, of up to 2464%. This shows that without syntactic subtraction, Routers have complex data constraints, with many free data variables, whose constraint solving requires a very substantial amount of computational resources. Moreover, with syntactic subtraction, code generated for Mergers and Routers has similar performance (i.e., compare the yellow lines for Merger and Router in Figure 6.2), whereas without syntactic subtraction, code generated for Mergers outperforms code generated for Routers (i.e., compare the blue lines for Merger and Router in Figure 6.2). As Routers form just the inverse—in terms of port directions—of Mergers as explained in Chapter 3, it makes sense for code generated for Routers to have similar performance as code generated for Mergers. Their previously measured differences in performance therefore actually indicated a significant deficiency of Lykos (similar to its previous inability to properly handle the neutrality of Syncs). Syntactic subtraction solves this problem, providing similar performance for code generated for Mergers and Routers, as one may expect.

In contrast to members of Merger, Router, LateAsyncMerger, and EarlyAsyncMerger, members of OddFibonacci and Chess require workers to globally synchronize with everybody (instead of pairwise synchronous or asynchronous interaction). For those families, thus, one may reasonably expect the number of workers to affect performance, as long as performance does not degrade below inverse-proportionality. Although not clearly visible in Figure 6.2 (because of the scale on the y-axis), Figure 6.3 shows that syntactic subtraction leads to significant speedup also for members of OddFibonacci and Chess.

Experiments II: Programs

I repeated the same experiments as in Chapter 5 (and Chapter 4), generating code for the NPB benchmarks with the `SUBTRACT_SYNTACTICALLY`-flag raised, but otherwise under the same conditions as in Chapter 5. Using syntactic subtraction, in contrast to semantic subtraction as used in Chapter 5, Lykos succeeded in generating code for all values of k for all NPB benchmarks.

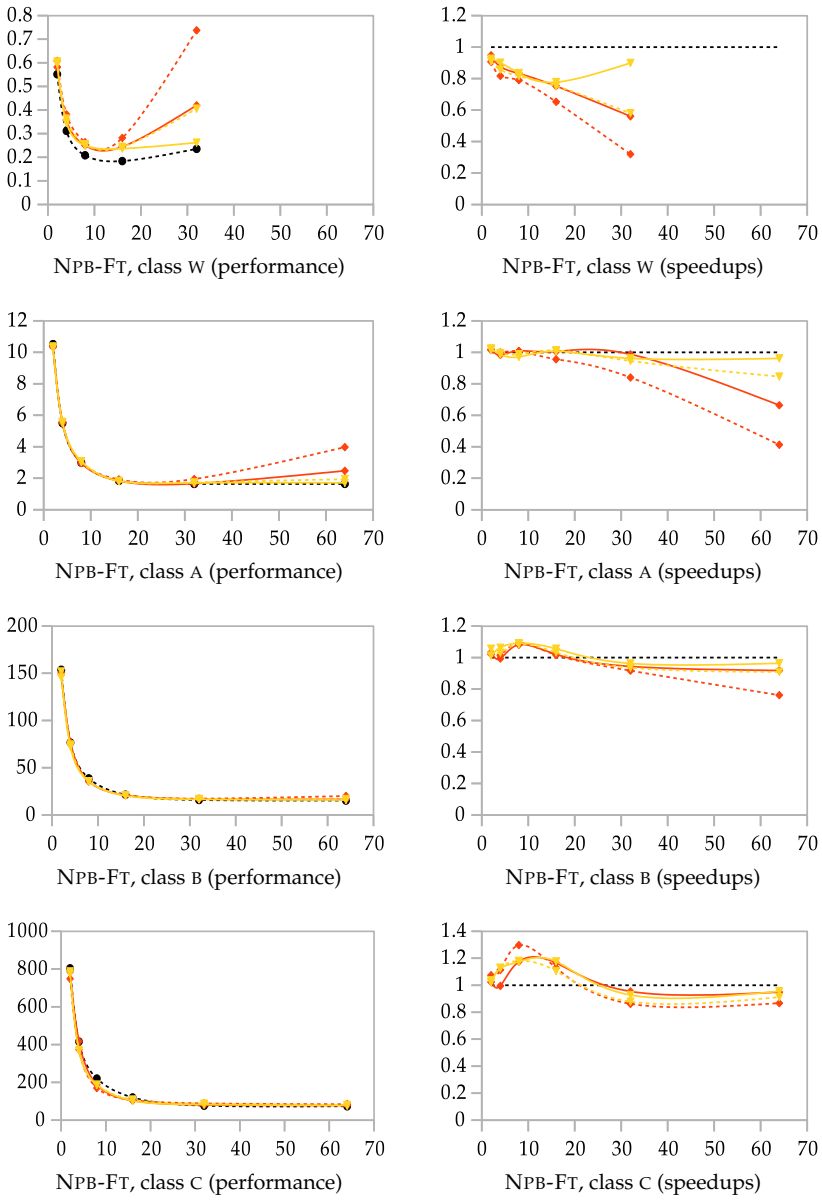

Figure 6.4: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

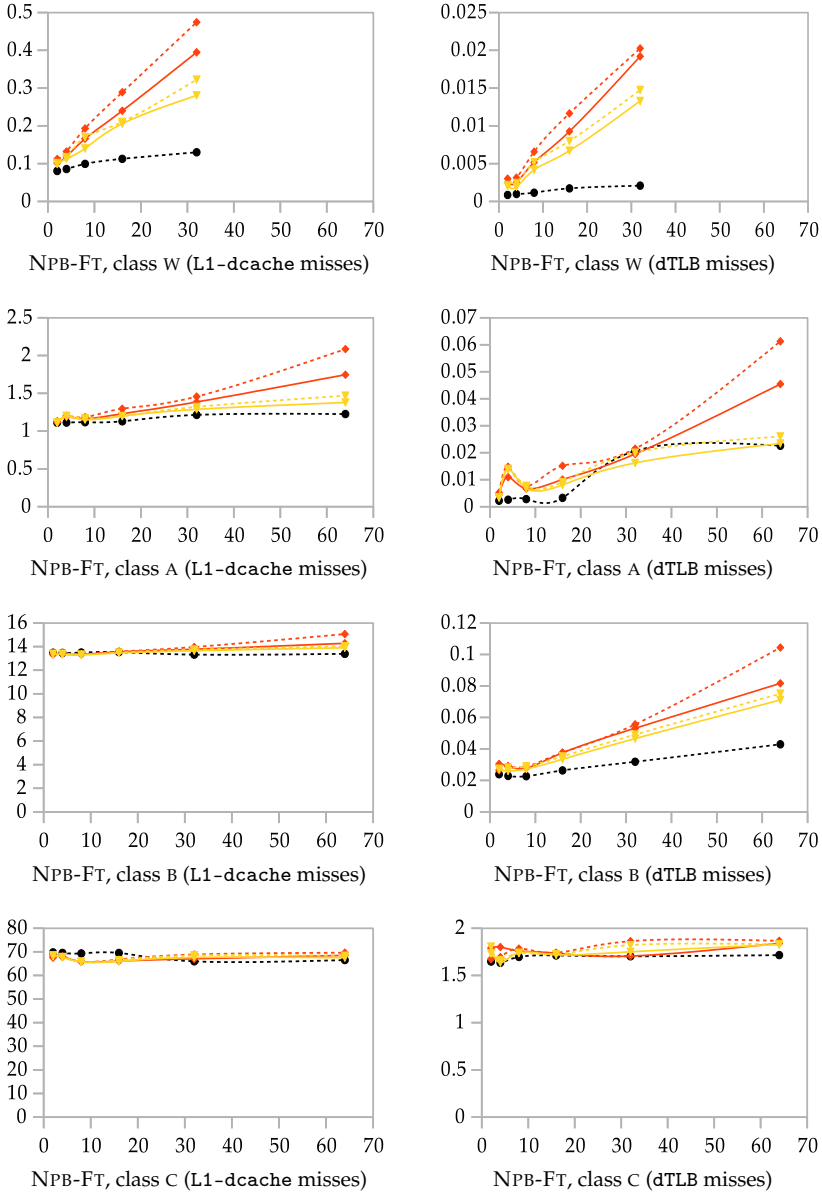

Figure 6.5: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

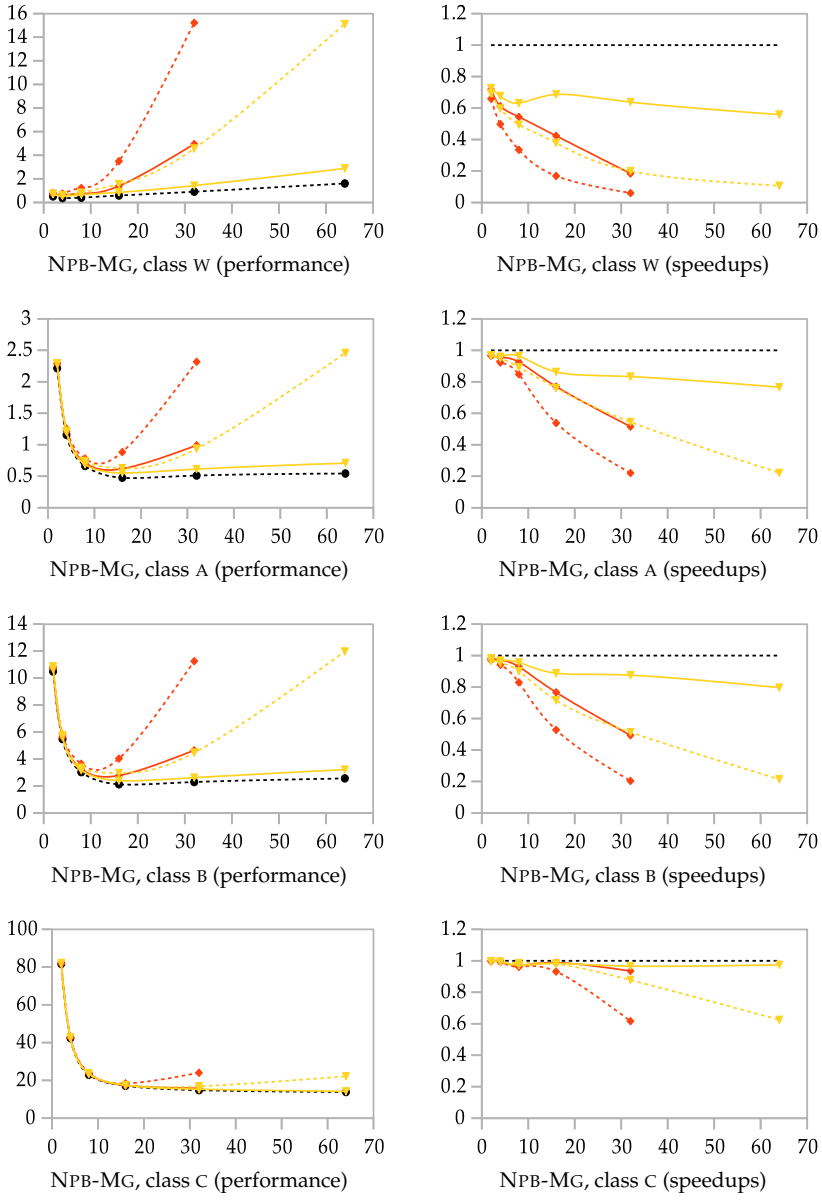

Figure 6.6: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

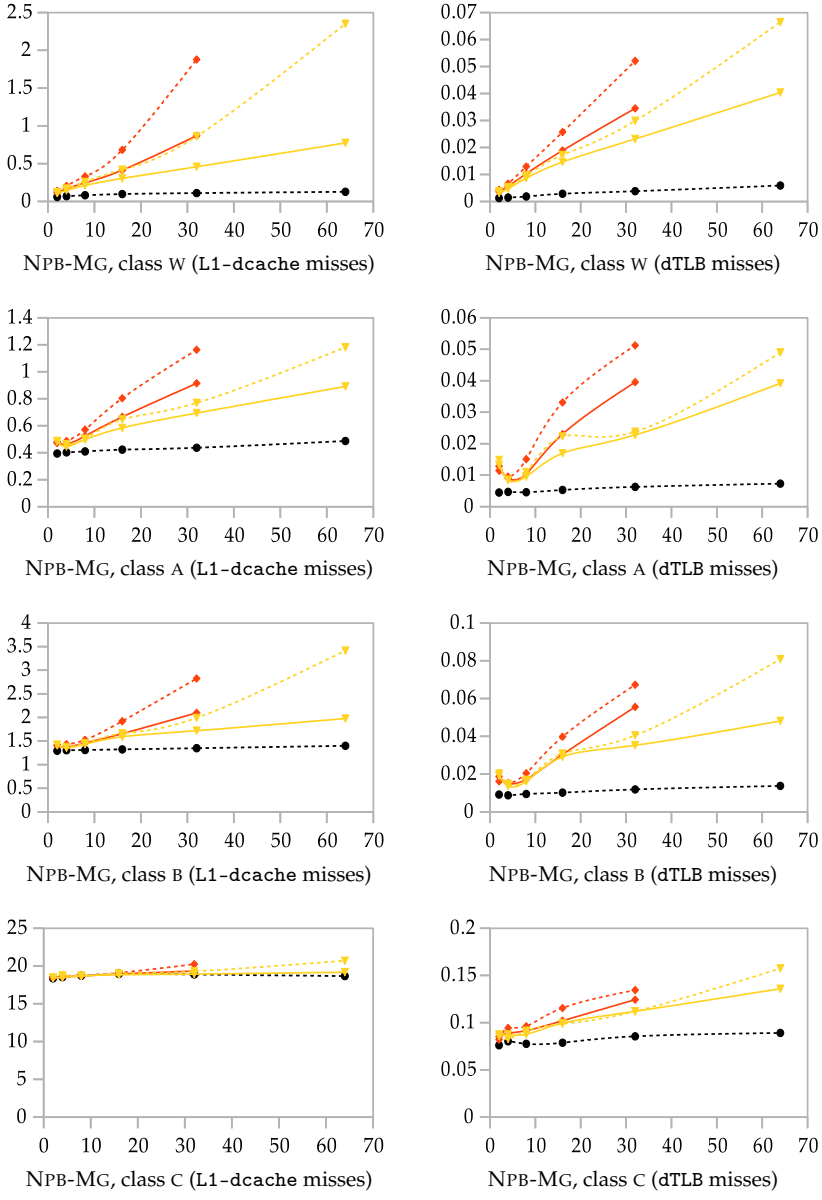

Figure 6.7: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

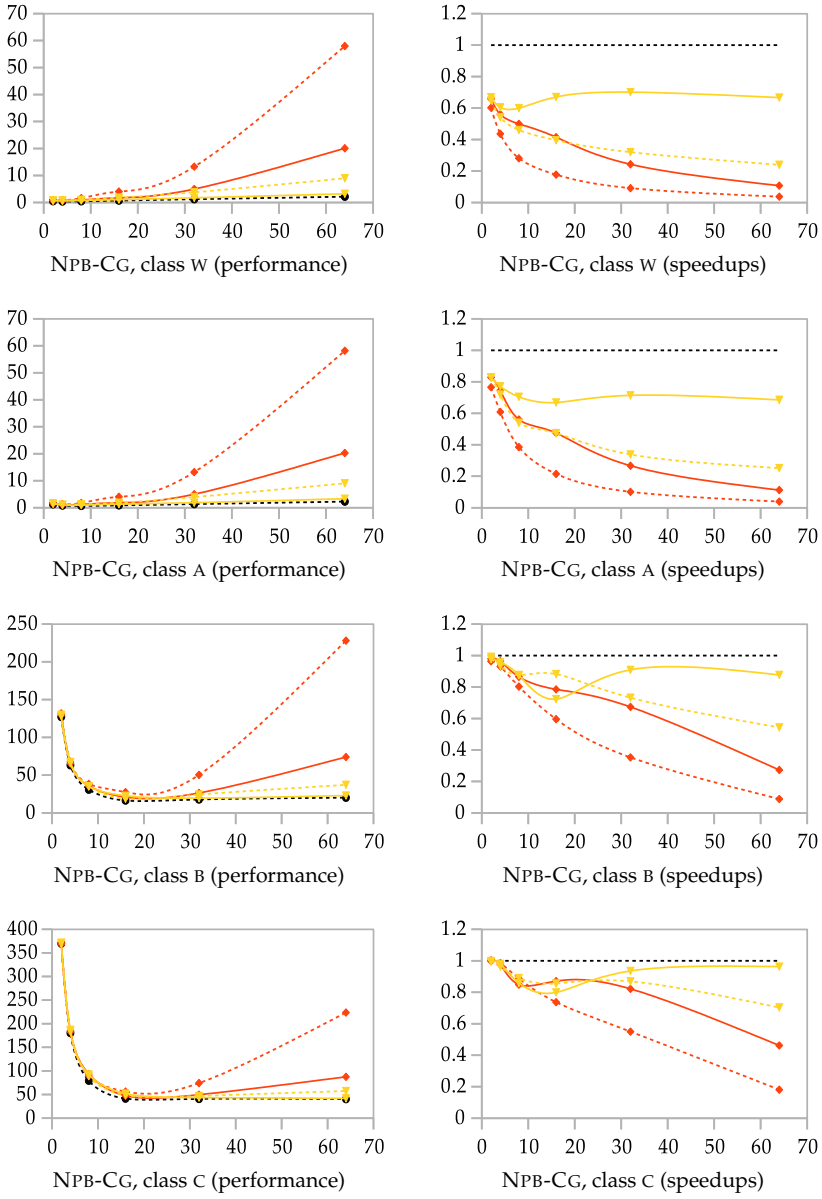

Figure 6.8: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

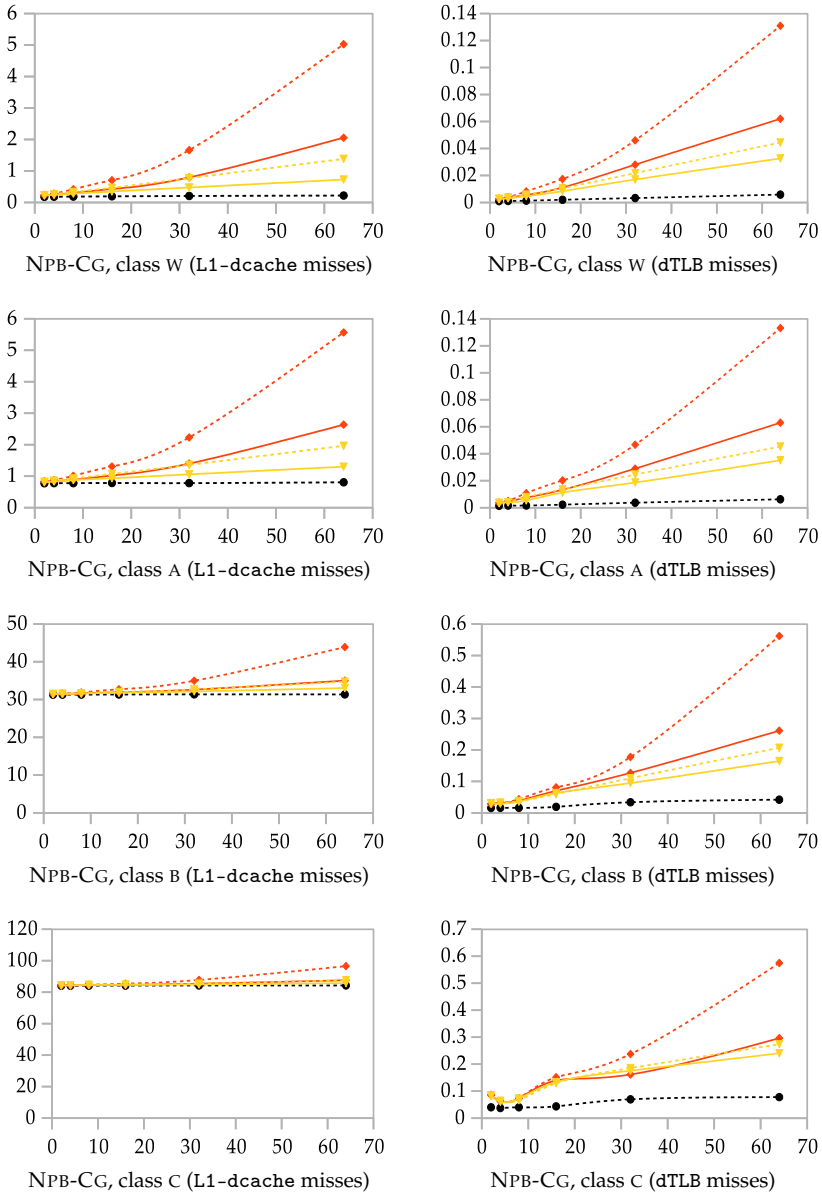

Figure 6.9: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

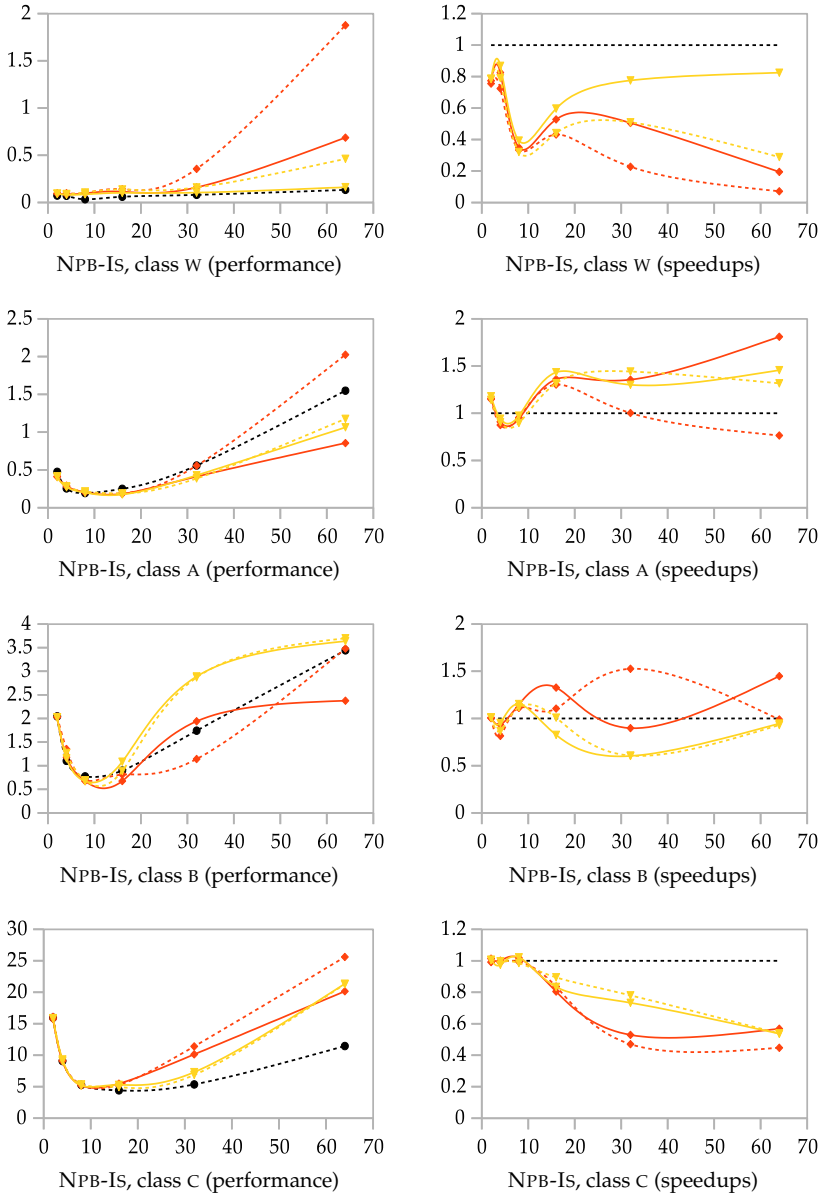

Figure 6.10: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

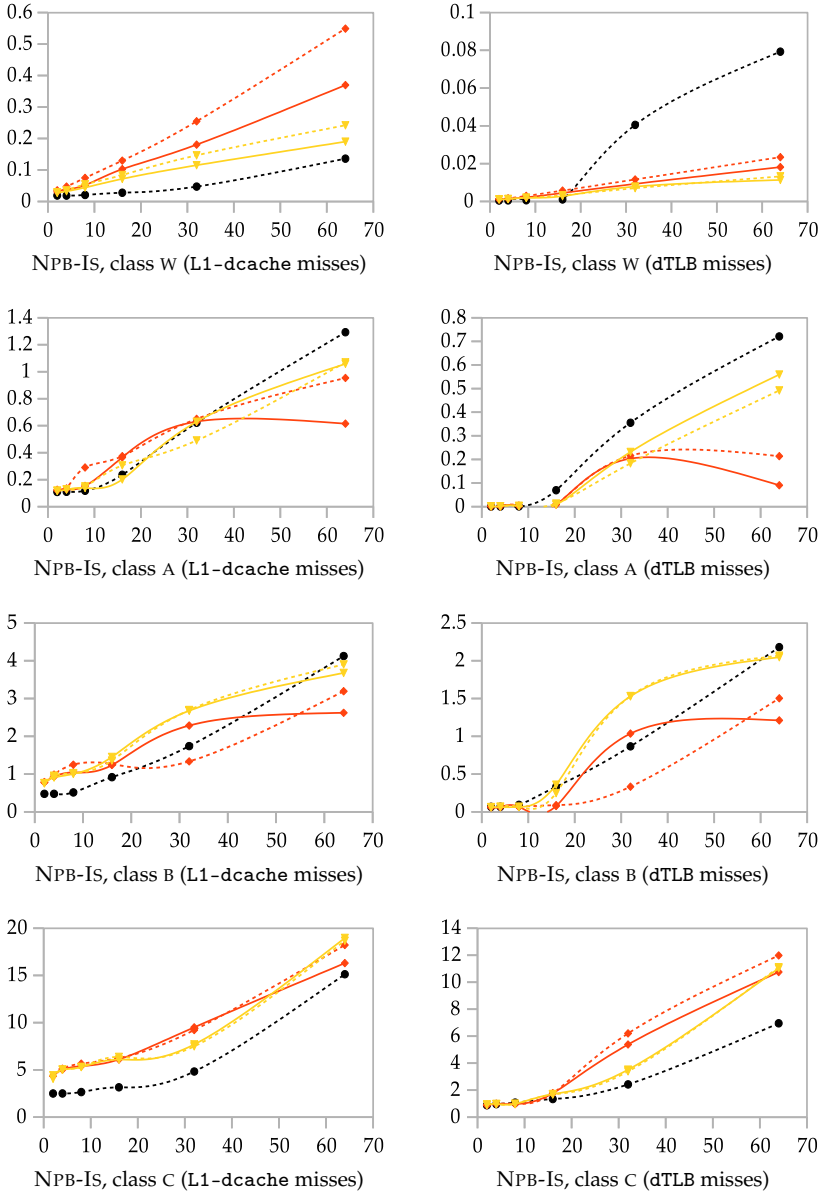

Figure 6.11: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

Figures 6.4–6.11 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 yellow/red lines represent the MasterSlavesInteractionPatternA-based FOCAML-to-Java-compiled versions of the NPB kernel benchmarks with and without syntactic subtraction (i.e., the yellow lines represent the new results, while the red lines represent the results in Chapter 5); the solid yellow/red 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 Chapter 5. Furthermore, overall, the FOCAML-to-Java-compiled versions with syntactic subtraction perform at least as well as the FOCAML-to-Java-compiled versions without syntactic subtraction (yellow lines versus red lines), and often better.

- 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 decreased substantially (compared to the results in Chapter 5): syntactic subtraction really makes an impact in these NPB kernel benchmarks.

- 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 Chapter 5, differences in numbers of cache misses explain why the FOCAML-to-Java-compiled versions of NPB-IS without syntactic subtraction outperform their supposedly improved FOCAML-to-Java-compiled versions with syntactic subtraction in class A, $k = 64$, class B, $k \in \{32, 64\}$, and class C, $k = 64$.

Figures 6.12–6.17 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 in class W, 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, I compiled

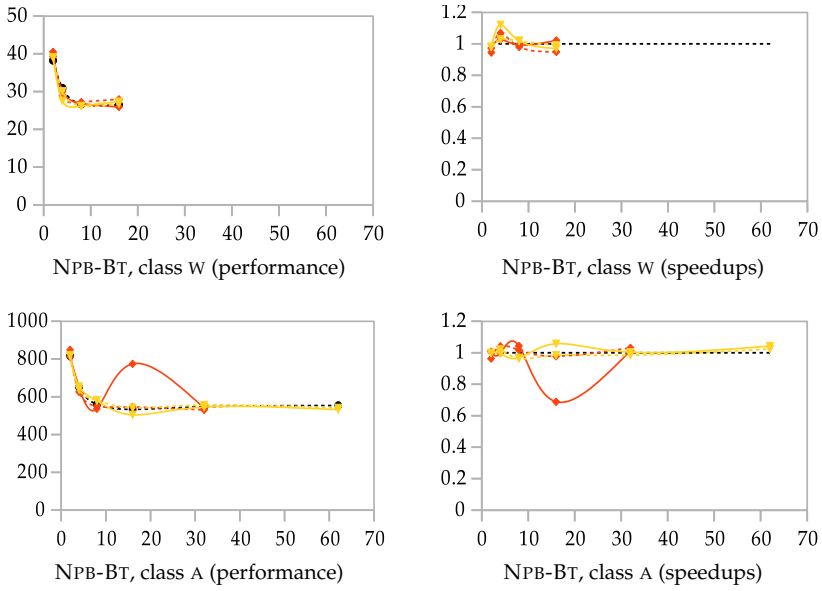

Figure 6.12: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

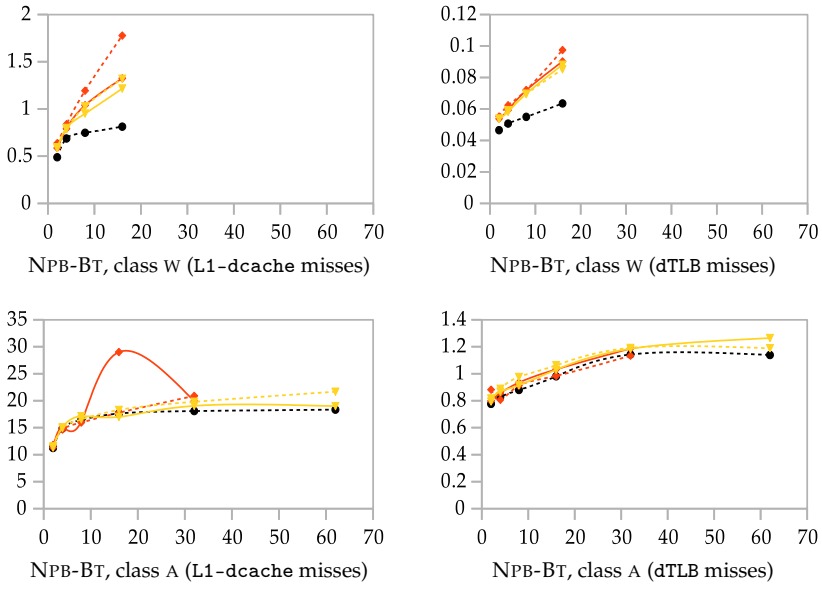

Figure 6.13: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

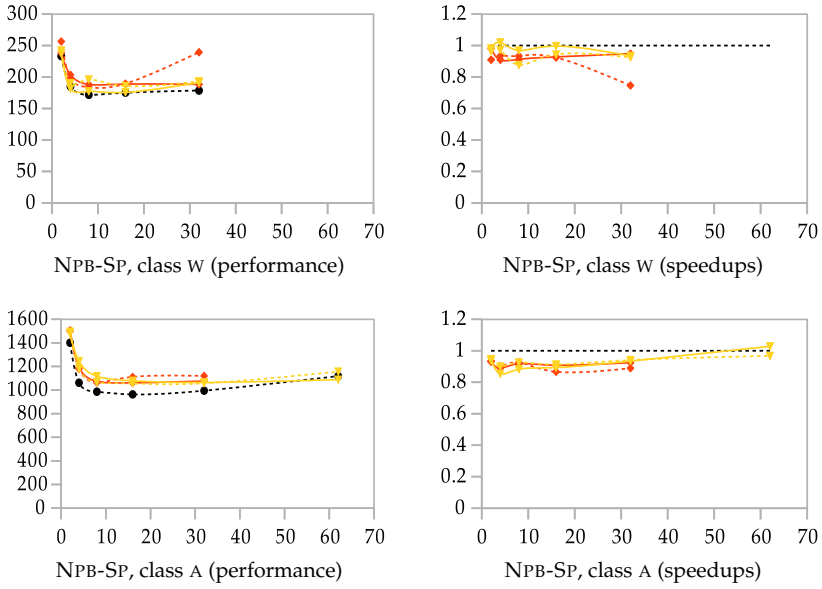

Figure 6.14: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

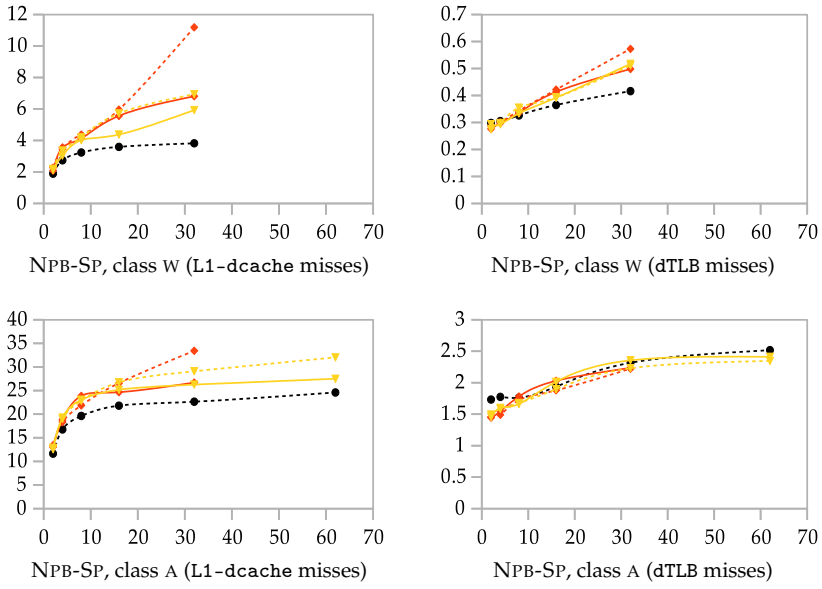

Figure 6.15: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

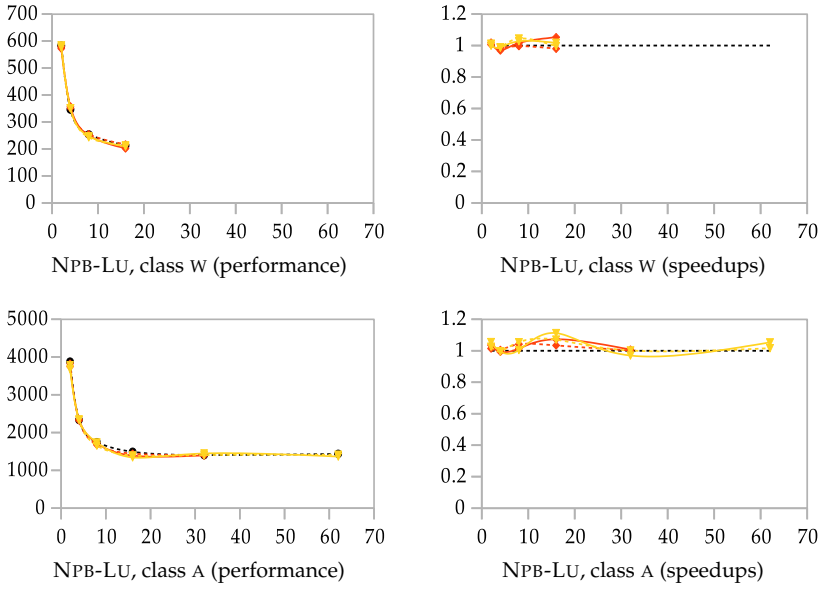

Figure 6.16: Left, performance (in seconds of run-time) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See the legend in Figure 9.4.

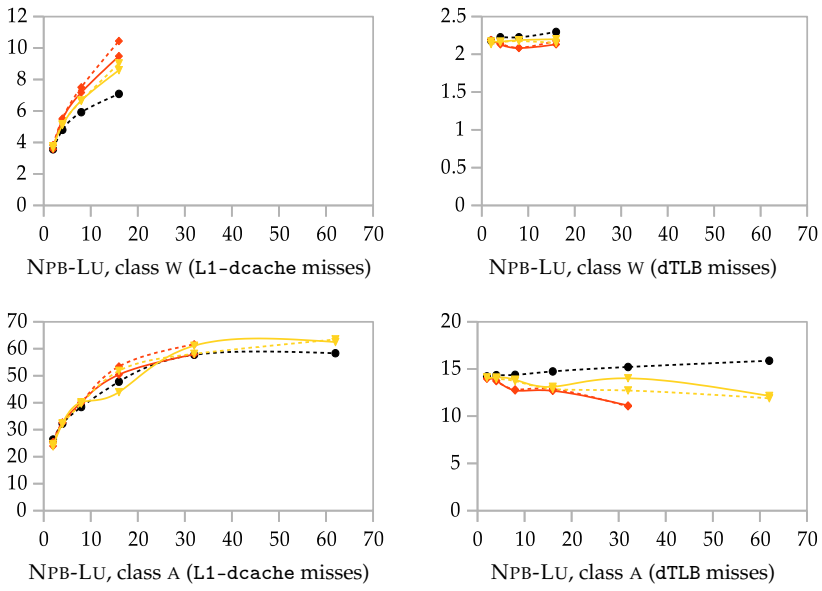

Figure 6.17: Left, `L1-dcache` misses as a function of the number of slaves, denoted by k. Right, `dTLB` misses as a function of k. See the legend in Figure 9.4.

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.