Automata-theoretic protocol programming

Jongmans, Sung-Shik Theodorus Quirinus

Chapter 9

Conclusion

9.1 Summary

In the early 2000s, hardware manufacturers ran into a number of obstacles that prevented them from directing the still exponential increase in transistors toward faster *unicore processors* [ABC+06]. In 2005, this trend led to the introduction of *multicore processors*, capable of processing multiple instruction streams in parallel. As a consequence, software engineers needed to start writing parallel programs. For optimal performance, regardless of how many cores a processor consists of—24 today or 1024 tomorrow—the same parallel program should divide instructions as evenly as possible over all its available cores. Conceptually, every parallel program consists of *workers*, which perform the actual computation, and *protocols*, which state the rules of interaction that workers must abide by. To achieve good performance on multicore processors, by *Amdahl's Law* [Amd67], software engineers should minimize the inherently sequential fraction of computation performed by workers. Additionally, and of increasing importance as this sequential fraction decreases [YMG14], software engineers should minimize the amount of resources spent on enforcing protocols. [Section 1.1]

A decade after the advent of multicore processors, many software engineers still use largely the same abstractions for parallel programming in general, and implementing protocols in particular, as forty years ago: threads, shared memory, and concurrency constructs for mutual exclusion [Dij02, Hoa74]. Even when programming with higher-level abstractions, such as *thread pools* [DM98, GPB+06, LSB09, Rei07, Rob13] or *actors* [AVWW96, Hal12, HO09], the implementation of nontrivial protocols often requires software engineers to use these relatively old techniques [TDJ13]. However, implementing protocols by manually managing shared memory with concurrency constructs for mutual exclusion has three major issues: (i) it complicates correctly implementing protocols, because it complicates reasoning about programs' behavior, (ii) it complicates implementing protocols with high performance, because it fails to preserve

crucial *intention information* that compilers need for automatically optimizing interaction code, and (iii) it complicates implementing protocols in a modular fashion, because it neither enforces nor encourages syntactic separation of computation code from interaction code. One way to resolve these issues consists of imposing another level of abstraction on top of shared memory, thereby eliminating the need for concurrency constructs for mutual exclusion for implementing protocols. Incarnations of this solution include *transactional memory* [HM93, Kni86, ST97] and *algorithmic skeletons* [Col88, GL10], but both these solutions have their limitations. [Section 1.2]

In another incarnation of this solution, software engineers implement their protocols at a new, high, *intention-expressing level of abstraction*, which provides constructs for implementing protocols while preserving as much information about software engineers' intention behind those implementations as possible. Not only should this make correctly implementing protocols easier for software engineers (thereby resolving the first issue, above), but it also makes automatically optimizing their resulting protocol implementations easier for compilers (thereby resolving the second issue). *Domain-specific languages* (DSL) for interaction seem a particularly well-suited vehicle for providing software engineers such intention-expressing levels of abstraction, through intuitive and protocol-tailored syntax. DSLs for interaction also force software engineers to syntactically separate protocol implementations from worker implementations (thereby resolving the third issue), and they naturally complement existing *general-purpose languages* (GPL): software engineers can continue implementing their workers in a GPL, but their protocols in a DSL, after which a DSL compiler translates those DSL-coded protocol implementations into GPL code. Together, hand-written GPL-coded worker implementations and compiler-generated GPL-coded protocol implementations constitute full GPL-coded parallel programs, whose grand integration can happen automatically, by the DSL compiler, completely transparent to software engineers. [Section 1.3]

In the programming model for one concrete instance of the previous approach, every worker has access to a number of conceptual *ports*. Through such ports, workers can interact with their "environment", by performing blocking I/O operations on those ports. Crucially, however, a worker never knows which other workers constitute its environment—every worker sees only its own ports and cannot directly address other workers or their ports. Instead, whenever a worker performs an I/O operation, a separate entity responsible for enforcing the protocols among workers determines whether this I/O operation may immediately complete—perhaps synchronously with already pending I/O operations performed by other workers—and if so, how data flow between ports; if not, the worker becomes *suspended* and remains suspended until its I/O operation completes at some future time. A true intention-expressing DSL for interaction, then, must provide constructs for expressing "a set of rules that control the way data is [exchanged through ports]", which, in fact, constitutes a dictionary definition of "protocol" (i.e., a generally accepted interpretation, and closest approximation of, the intention that people have when they use the word "protocol"). [Section 1.3]

Every set of atomic data-flows between ports yields one *instance of interaction* among workers. A simple formal model of instances of interaction, then, consists of functions from ports (involved in an instance of interaction) to data (observable on those ports in that instance). An infinite sequence of such functions subsequently models one *chain of interaction* admitted by a protocol; a set of such infinite sequences models all that protocol's admissible infinite chains. Interpreting the latter kind of sets (and their elements) as automata-theoretic *languages* (and infinite *words*), a concise formal model of protocols consists of automata over such languages. Essentially, every transition of such an automaton represents one rule "that control[s] the way data is [exchanged through ports]", and as such, this kind of automata truly captures the intention behind the word "protocol". [Section 2.1]

In one rather naive automaton model of protocols, transitions explicitly carry functions from ports to data as their labels. Often, however, this gives rise to infinitely many transitions. Therefore, in a more advanced automaton model of protocols—*constraint automata*—transition labels symbolically represent (possibly infinite) sets of functions from ports to data as two constraints: a *synchronization constraint*, which consists of a set of ports, and a *data constraint*, which consists of a first-order logic formula. Conceptually, a synchronization constraint specifies which ports synchronize in an admissible instance of interaction (i.e., through which ports data synchronously flow); a data constraint specifies the particular data observable on those ports. By definition, *behaviorally equivalent* constraint automata accept the same language (i.e., their corresponding protocols admit exactly the same chains of interaction). Analogously, *behaviorally congruent* constraint automata also accept the same language, but moreover, they also have coincident transition relations (in a *bisimulation* kind of way [BSAR06, Mil89]). Although only behavioral equivalence matters in the end, behavioral congruence often simplifies proofs and reasoning about constraint automata. While individual constraint automata model individual protocols, their *multiplication* and *subtraction* model protocol composition (i.e., superimposing different sets "of rules that control the way data is [exchanged through ports]") and abstraction (i.e., removing ports not of interest). [Section 2.1]

A Java library for constraint automata allows software engineers to represent constraint automata, their multiplication, and their subtraction in Java, as nonexecutable data structures. [Section 2.2]

Because constraint automata model (the intention behind) protocols, they constitute a well-suited semantic domain for an intention-expressing DSL for interaction. As in any kind of engineering, rather than providing software engineers syntactic constructs for directly constructing constraint automata, a more scalable approach consists of providing software engineers constructs for representing *multiplication expressions* of constraint automata, thereby exploiting their compositionality. In a graphical syntax for this approach, software engineers draw multiplication expressions as data-flow diagrams. This graphical syntax essentially yields *Reo* [Arb04, Arb11], an existing graphical language

for compositional construction of protocols. In an alternative textual syntax, called FOCAML, software engineers write multiplication expressions as declarative pieces of code. Reo and FOCAML have complementary use cases: Reo visualizes data-flows, which makes seeing—at a glance—which rules of interaction a protocol enforces easier, while FOCAML has more expressive power in terms of parametrization. [Section 3.1]

The FOCAML editor/parser/interpreter plugin for Eclipse 4.x enables software engineers to write FOCAML programs. This plugin can also translate Reo diagrams, drawn using an existing collection of plugins for Reo development, into FOCAML code. [Section 3.2]

To compile FOCAML code (and, using the Reo-to-FOCAML translator, also Reo diagrams) into GPL code to get a full GPL-coded program, several approaches exist. In the *Distributed Approach*, a FOCAML compiler translates every primitive constraint automaton in a multiplication expression into its own thread. These threads use an expensive consensus algorithm to compose their local behavior into consistent global behavior, effectively computing their product at run-time. In the *Centralized Approach*, in contrast, a FOCAML compiler first computes the product of the primitive constraint automata in a multiplication expression, then subtracts all the resulting *internal ports* (i.e., ports that serve as an *input port* in one constraint automaton and as an *output port* in another one, which nobody can access anymore in their product), and finally translates the resulting composite constraint automaton into a single thread. [Section 4.1]

GPL code generated from FOCAML code under the Distributed Approach exhibits maximal parallelism (with respect to the primitive constraint automata in a multiplication expression) and, therefore, achieves relatively high *throughput*. The expensive *consensus algorithm* required in the Distributed Approach, however, inflicts serious overhead. Code generated under the Distributed Approach, therefore, suffers from relatively high *latency*. In contrast, code generated under the Centralized Approach exhibits maximal sequentiality and, therefore, suffers from relatively low throughput. By avoiding the need for expensive consensus as in the Distributed Approach, however, the Centralized Approach eliminates a source of serious overhead. Code generated under the Centralized Approach, therefore, achieves relatively low latency. [Section 4.1]

Without additional compiler flags, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code under the Centralized Approach. Built on top of the FOCAML editor/parser/interpreter plugin, it also supports compiling Reo into Java. [Section 4.2]

The Centralized Approach has two scalability problems, one of which manifests at compile-time, the other of which manifests at run-time. At compile-time, computing the product of all primitive constraint automata in a multiplication expression may give rise to *state space explosion*; at run-time, the single thread for a computed product may give rise to *oversequentialization*. In oversequentialized compiler-generated code for a constraint automaton, two *independent transitions*, originating from two independent primitive constraint

automata in a multiplication expression (i.e., constraint automata that share no ports), cannot fire simultaneously but only consecutively, thereby unjustifiably reducing throughput. To solve these two problems, while avoiding the need for expensive consensus as in the Distributed Approach, a FOCAML compiler can apply the *Hybrid Approach*. This third approach sits somewhere between the Distributed Approach and the Centralized Approach, by serializing *useless parallelism* (i.e., the kind of parallelism that requires heavy synchronization) to preserve only *useful parallelism* (i.e., the kind of parallelism that requires light synchronization). [Section 5.1]

In the Hybrid Approach, a FOCAML compiler first distributes the primitive constraint automata in a multiplication expression over disjoint subsets. Subsequently, as in the Centralized Approach, the compiler computes *per-subset products* by multiplying, for every subset, the constraint automata in that subset (and by subsequently subtracting internal ports from those per-subset products). Finally, as in the Distributed Approach, the compiler translates the resulting per-subset composite constraint automata into as many threads. Contrasting the Distributed Approach, however, these threads require only a cheap consensus algorithm for synchronizing their behavior. [Section 5.1]

Just as the expensive consensus algorithm required in the Distributed Approach, the cheap consensus algorithm required in the Hybrid Approach corresponds to run-time multiplication of constraint automata, but under a new definition of multiplication. This new multiplication, always called *l(ocal)-multiplication*, generally does not coincide with the old multiplication, sometimes called *g(lobal)-multiplication*. For instance, l-multiplication does not exhibit associativity, whereas g-multiplication does. Only g-multiplication matters in the end: a FOCAML compiler always starts with a g-multiplication expression, which forms an absolute reference point for compilation soundness. In the worst case, then, the threads generated in the Hybrid Approach do not behave as their g-multiplication expression (because they use the cheap consensus algorithm, which corresponds to l-multiplication), making their behavior unsound. However, by carefully *partitioning* the primitive constraint automata in a g-multiplication expression into subsets (i.e., the first step in the Hybrid Approach), a FOCAML compiler can guarantee that the behavior of their corresponding threads corresponds to g-multiplication, even though those threads use the cheap consensus algorithm. In Reo terminology, such partitioning corresponds to computing *synchronous and asynchronous regions* [CP12, **J**CP12, **J**CP16, PCdVA11, PCdVA12, Pro11]. [Section 5.1]

With compiler flag `PARTITION` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code under the Hybrid Approach. [Section 5.2]

One problem that the Hybrid Approach, as the Centralized approach, still suffers from concerns the size of data constraints, symptomized by the fact that the neutral element for multiplication negatively affects performance of compiler-generated code: multiplying this neutral element any number of times with other constraint automata behaviorally makes no difference, but performance-

wise, it does. This problem results from the definition of subtraction, which a FOCAML compiler uses for removing internal ports (from per-subset products in the Hybrid Approach or from the full product in the Centralized Approach). In particular, the original definition of subtraction removes ports from data constraints in constraint automata only *semantically*, by enveloping each of its data constraints in an existential quantification for the port-to-remove. By doing so, subtracting a port from a constraint automaton actually increases the sizes of its data constraints. Larger data constraints generally require more computational resources to handle at run-time, which requires constraint solving over a finite discrete domain. This, then, explains why the neutral element for multiplication negatively affects performance: it yields equivalent, yet substantially larger, data constraints. [Section 6.1]

Syntactic subtraction, a new subtraction on constraint automata, avoids this problem. Instead of enveloping data constraints in existential quantifications, it tries to find a suitable—semantically neutral—substitute for the port-to-remove in every data constraint. If such substitutes exist, syntactic subtraction subsequently replaces every occurrence of the port-to-remove in a data constraint with its substitute in that data constraint. By subsequently removing obvious tautologies, syntactic subtraction actually decreases data constraint sizes. The previous problem with the neutral element for multiplication then also goes away. [Section 6.1]

Syntactic subtraction works effectively only when applied to *normalized constraint automata*. Normalization, applicable to every well-formed constraint automaton, therefore constitutes an important operation. [Section 6.1]

With compiler flag `SUBTRACT_SYNTACTICALLY` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code using syntactic subtraction (instead of semantic subtraction). [Section 6.2]

Notwithstanding syntactic subtraction, handling data constraints with general-purpose *constraint solving* techniques inflicts a significant amount of overhead at run-time—not only overhead proportional to the size of a data constraint but also a constant overhead for preparing, making, and processing the result of every constraint solver invocation—especially for relatively simple data constraints. Many such data constraints, however, essentially constitute declarative specifications of data-flows between ports. When provided such specifications, then, most—if not all–software engineers would probably just write direct imperative implementations instead of indirect invocations to a constraint solver. [Section 7.1]

Commandification, an operation on constraint automata, aims to automate just that: it translates every data constraint in a constraint automaton into a piece of imperative code, called a *data command*, in a behavior-preserving way (proved by equipping data commands with a transition system semantics, notions of partial/total correctness, and Hoare logic [AdBO09, Hoa69]). The construction of data commands for data constraints uses hypergraph representations of the data-flows represented by those data constraints, paying special attention to cycles in such hypergraphs. [Section 7.1]

With compiler flag `COMMANDIFY` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code using commandification. [Section 7.2]

To fire any transition in a constraint automaton at run-time, its corresponding thread first needs to check all transitions out of the current state for enabledness (regardless of whether this thread came about through the Distributed, Centralized, or Hybrid Approach). This requires $\mathcal{O}(k)$ time, where k denotes the number of transitions. Often, the number of transitions increases at least linearly as the number of workers increases. Consequently, often, the time required to check transitions for enabledness also increases at least linearly as the number of workers increases. This may cause the performance of compiler-generated code to scale poorly. [Section 8.1]

Queue-inference aims to solve this problem in cases where the transitions to check for enabledness differ from each other in a "well-behaved way". For instance, suppose that all k transitions involve only two ports: the same output port but a different input port in each transition. Originally, to check these transitions for enabledness, a thread checks both of the two ports involved in each of these transition for a pending I/O operation. But now, suppose that whenever a worker performs an I/O operation on a port, it also offers that port into a special *queue*. To *simultaneously* check all k transitions for enabledness, then, a thread needs to check only this queue for nonemptiness (in addition to checking the output port for a pending I/O operation); if so, this thread knows that at least one of the k transitions can fire. Consequently, k checks for enabledness collapse into just one. Automata-theoretically, such collapsing corresponds to *combining* multiple transitions into a single transition with a special transition label, to express that this combined transition has an efficient queue-based implementation. With queue-inference, then, a FOCAML compiler first analyzes constraint automata for transitions amenable to combination, subsequently actually combines such transitions, and finally generates code with queues for combined transitions. [Section 8.1]

With compiler flag `INFER_QUEUES` raised, the FOCAML-to-Java compiler plugin for Eclipse 4.x compiles FOCAML code into Java code using queue-inference. [Section 8.2]

Figures 9.2 and 9.3 show experimental results (absolute performance and relative speedups) for eight *families* of constraint automata, each parametric in a natural number k. Blue lines represent code generated under the Centralized Approach without further improvements; red lines represent code generated under the Hybrid Approach; yellow lines represent code generated under the Hybrid Approach, plus syntactic subtraction; green lines represent code generated under the Hybrid Approach, plus syntactic subtraction, plus commandification; purple lines represent code generated under the Hybrid Approach, plus syntactic subtraction, plus commandification, plus queue-inference. If a chart excludes one of these lines, applying the improvement to which this missing line corresponds does not actually change the generated code (i.e., the

Centralized Approach	[Chapter 4]
Hybrid Approach	[Chapter 5]
+ syntactic subtraction	[Chapter 6]
+ commandification	[Chapter 7]
+ queue-inference	[Chapter 8]

Legend

- In performance charts, every *solid colored line* represents the performance of compiler-generated code, in millions of rounds (more = better).
- In performance charts, every *dotted colored line* represents inverse-proportional growth relative to its solid colored line at $k = 1$.
- In speedup charts in Chapters 4–8, every *solid black line* represents the speedup of compiler-generated code in the current chapter relative to the dotted black line in the same chart.
- In every speedup chart in Chapters 4–8, every *dotted black line* represents the speedup of compiler-generated code in the previous chapter relative to itself (i.e., 0% speedup, i.e., a baseline).
- In speedup charts in Chapter 9, every *solid colored line* whose color corresponds to Chapter i represents the speedup of compiler-generated code in Chapter i relative to the dotted colored line in the same chart.
- In speedup charts in Chapter 9, every *dotted colored line* whose color corresponds to Chapter i represents the speedup of compiler-generated code in Chapter i relative to itself (i.e., 0% speedup, i.e., a baseline).

Description

Figure 9.1: Legend for charts about protocol experiments

improvement has no effect). [Sections 4.2, 5.2, 6.2, and 7.2]

Figures 9.2 and 9.3 show that starting from the Centralized Approach, overall, performance gets better with every successive improvement, while software engineers no longer need to worry about manually applying these kinds of optimizations to their protocol implementations. For instance, in the Hybrid Approach, only the compiler decides about how to parallelize protocol implementations, completely transparent to software engineers. Orthogonally, syntactic subtraction automatically simplifies data-flows, essentially by eliminating intermediate vertices in a data-flow graph, while commandification linearizes declarative specifications of data-flows into efficient imperative implementations, as carefully ordered sequences of instructions. Perhaps most advanced, with queue-inference, the compiler can—all by itself—identify cases

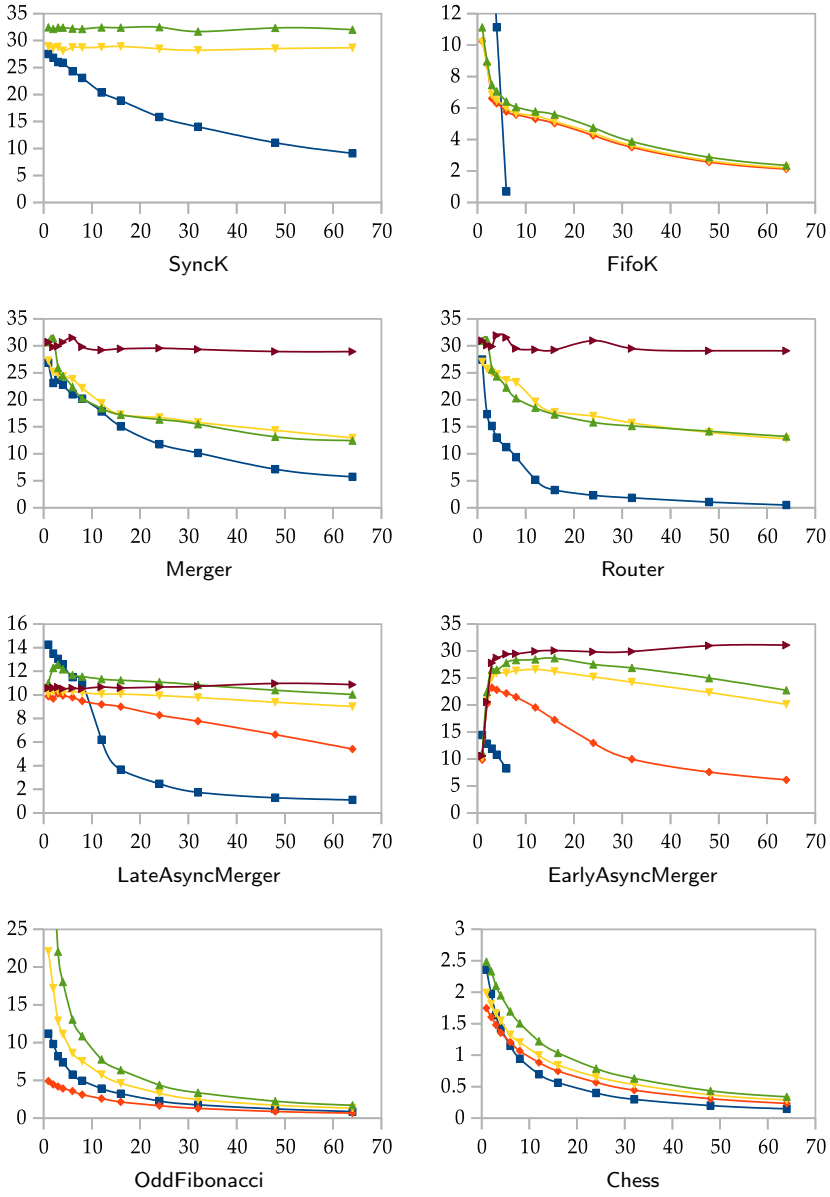

Figure 9.2: Performance (in number of completed rounds per four minutes) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See also Figure 9.1.

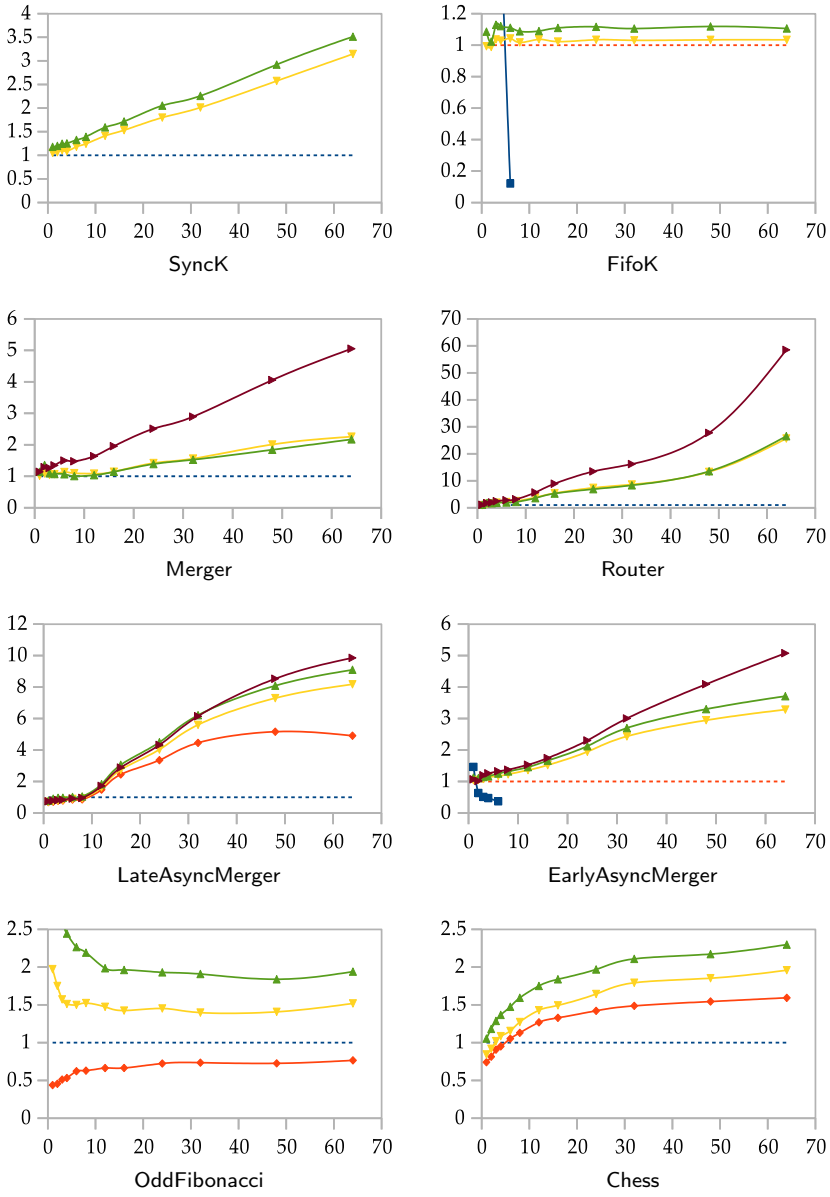

Figure 9.3: Speedup (relative to compiler-generated code in Chapter 4 and, for FifoK and EarlyAsyncMerger, relative to compiler-generated code in Chapter 5) as a function of the number of Syncs/Fifos/producers/consumers/chess engines, denoted by k. See the legend in Figure 9.1.

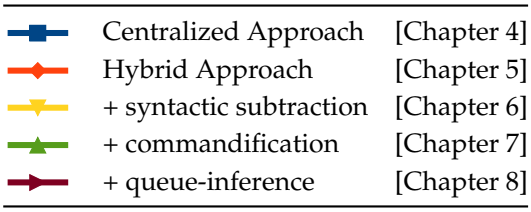

Figure 9.4: Legend for program experiment charts

where the use of queue data structures yields more scalable protocol implementations than individual variables would yield. Relieving software engineers from the responsibility of carrying out such optimizations—which, one way or the other, they themselves would have to carry out otherwise—simplifies their development of not only correct but also efficient protocol subprograms. And in each of these optimizations, the intention information captured by constraint automata plays an essential role in their automation, both at design-time (defining and modeling the optimization) and at compile-time (testing the applicability of the optimization for a given input). [Section 1.3]

Figures 9.5–9.11 show experimental results for FOCAML-to-Java-compiled versions of NASA's well-established *NAS Parallel Benchmarks* (NPB) [BBB+91, BBB+94], including experimental results for the Java versions of these bench-

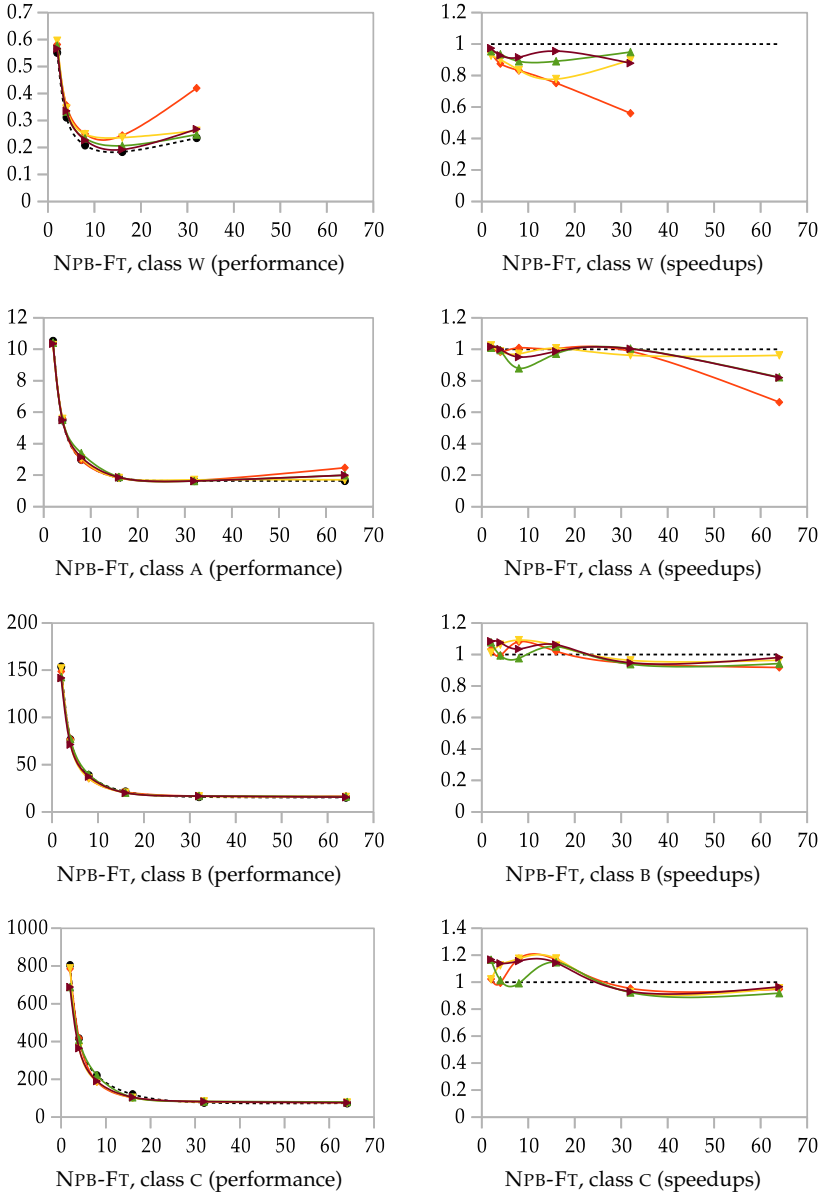

Figure 9.5: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

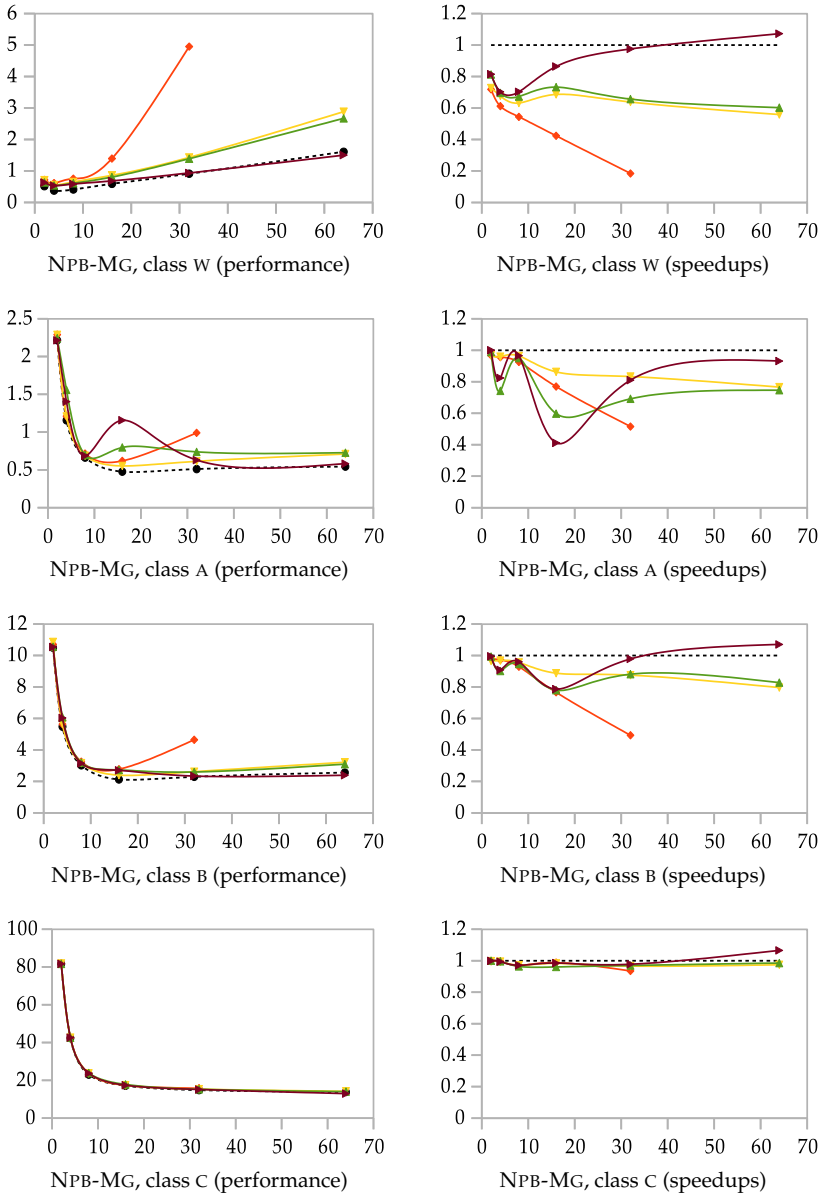

Figure 9.6: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

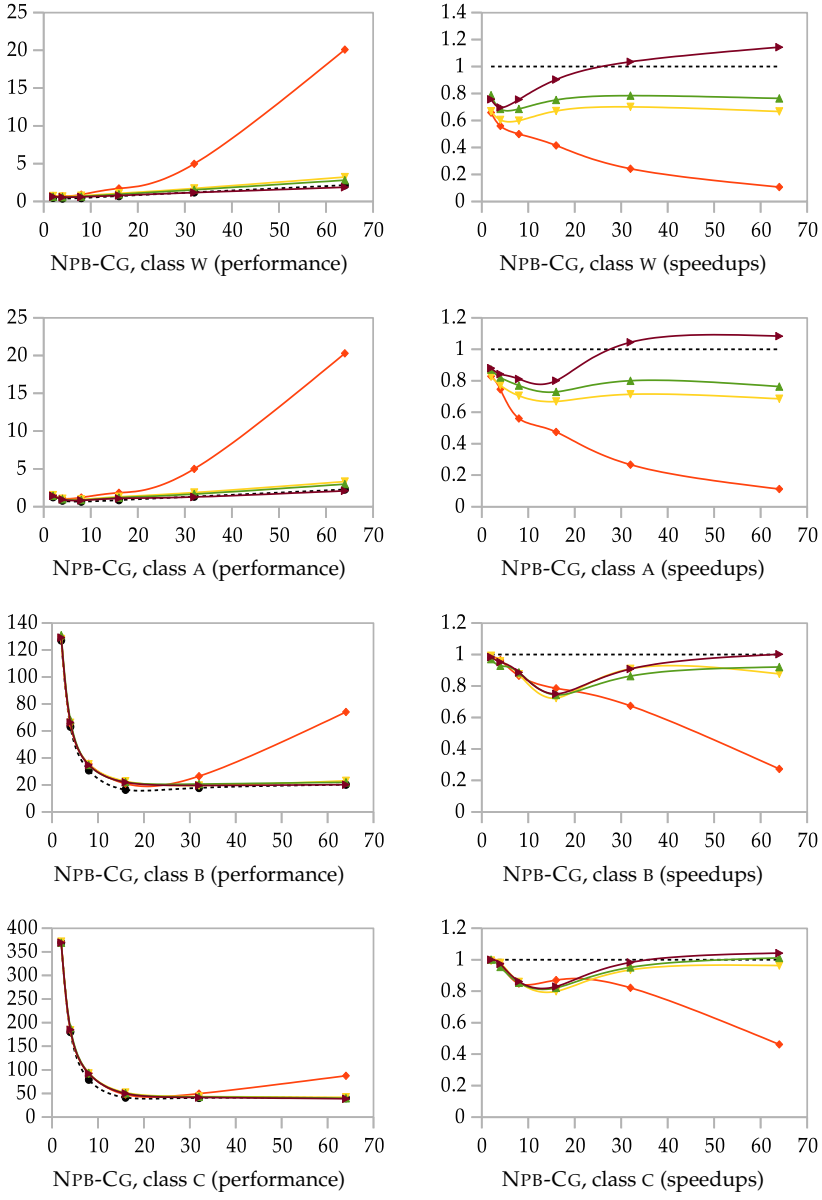

Figure 9.7: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

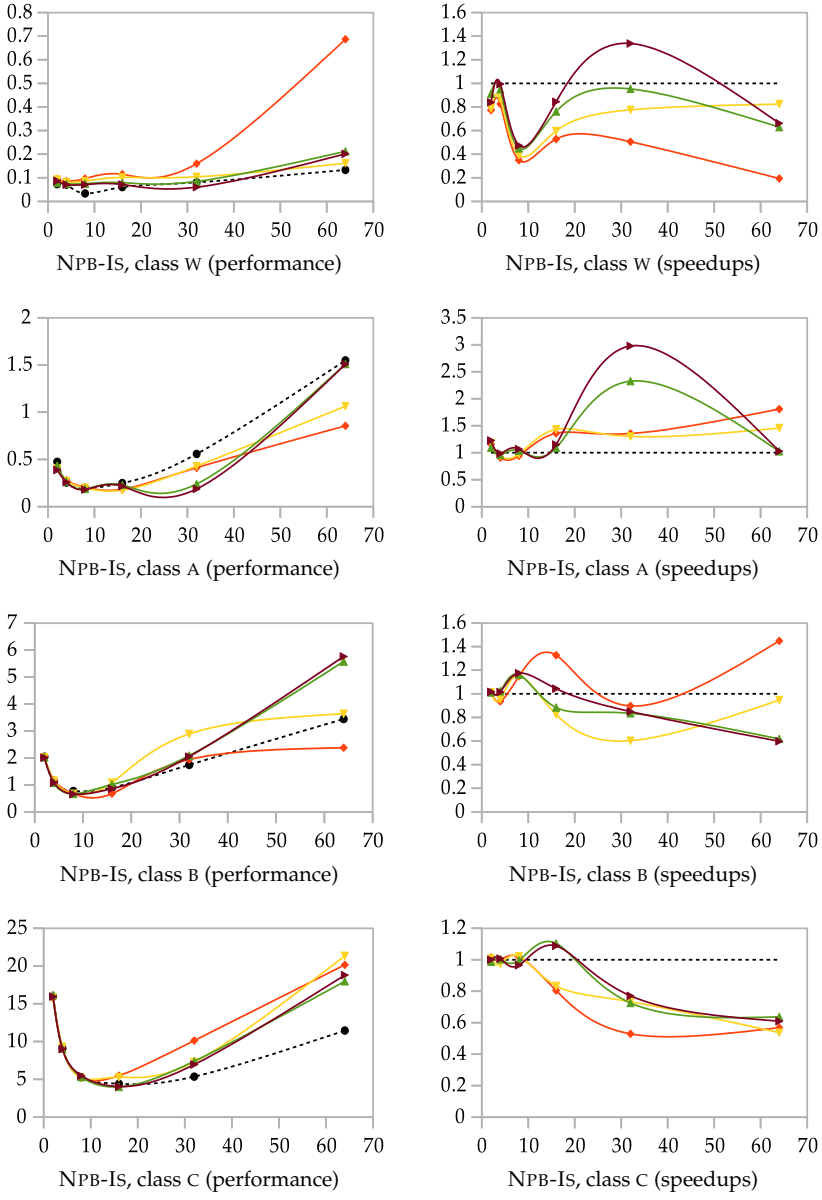

Figure 9.8: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

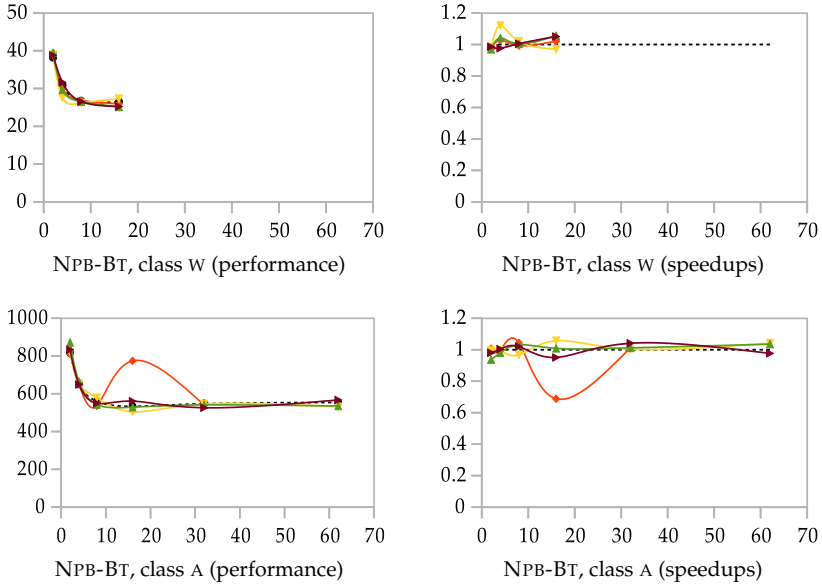

Figure 9.9: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

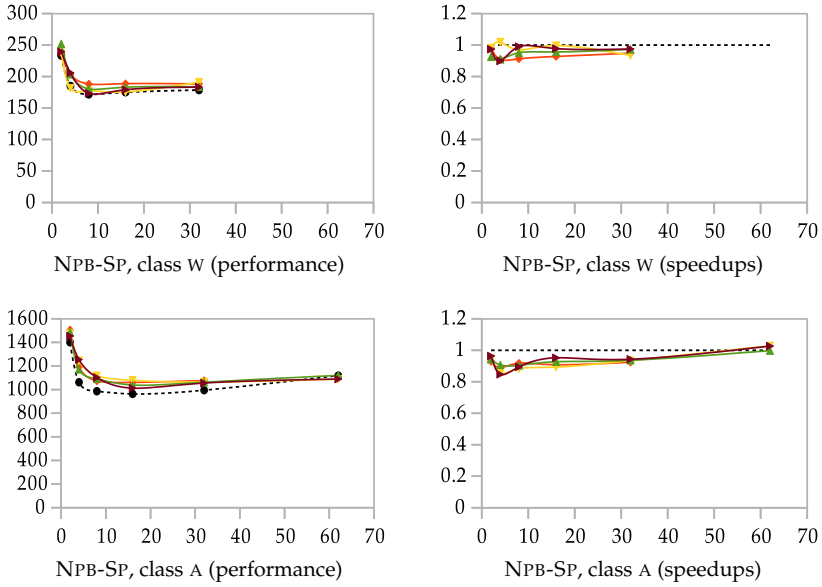

Figure 9.10: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

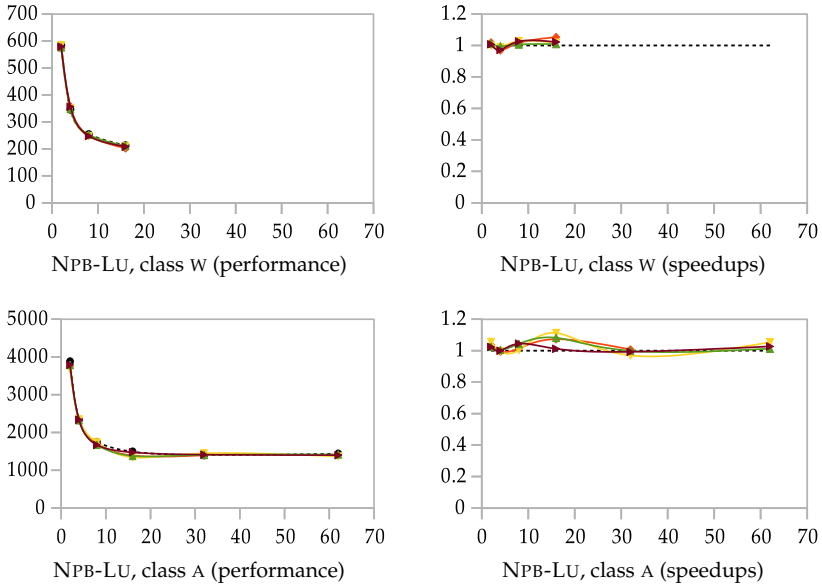

Figure 9.11: Left, performance (in seconds) as a function of the number of slaves, denoted by k. Right, speedup as a function of k. See also Figure 9.4.

marks by Frumkin et al. [FSJY02, FSJY03]. In this realistic and extensive case study consisting of full Java programs instead of protocols in isolation, overall, the FOCAML-to-Java-compiled versions perform better with every successive improvement. Moreover, with all improvements in place, the FOCAML-to-Java-compiled versions perform roughly as well as the Java versions, sometimes a bit slower, but sometimes also a bit faster. If anything, these promising results indicate the practical feasibility of using a high, intention-expressing level of abstraction for implementing protocols. [Sections 4.2, 5.2, 6.2, and 7.2]

Although the current FOCAML compiler targets Java, neither its compiler-generated code nor the corresponding run-time library uses any Java-specific features. Combined with the fact that neither the Hybrid Approach, nor syntactic subtraction, nor commandification, nor queue-inference—all formalized and proven correct at the higher level of constraint automata instead of at the lower level of GPL code—depend in any way on Java, any GPL that supports some form of threading and mutual exclusion may serve as a target language for FOCAML compilation. For instance, for his MSc thesis [vdN15], Van de Nes developed a compiler that generates C code. [Section 4.2]

9.2 Future Work

This thesis covers only the bare essentials of FOCAML compilation, leaving plenty of room for interesting and challenging future work. I mentioned some of the minor opportunities (i.e., smaller projects, perhaps good for a single publication) throughout the text. In this final section, I want to discuss three major opportunities (i.e., larger projects, probably good for multiple publications), in no particular order.

- *Further compilation*

  Explicit user-threading APIs (as defined in Chapter 1) expose just enough details of the underlying hardware—but no more—for software engineers to get reasonable performance with a reasonable amount of effort. Perhaps if such APIs would expose more details, expert software engineers would squeeze even more performance out of the hardware; average software engineers, however, would require a disproportionate amount of effort to write programs even with just reasonable performance. Consequently, explicit user-threading APIs became the minimal abstraction that software engineers use to write programs, in general.

  In my specific context of FOCAML compilation, however, it seems less obvious that also compiler-generated code should invoke explicit user-threading APIs. After all, as I need to build a compiler only once anyway, it may very well pay off to put extra effort in generating code *below* explicit user-threading APIs, and perhaps even *below* the operating system's kernel threads, directly managing cores, to further improve performance.

  In one extreme incarnation of this approach, a FOCAML compiler generates assembly code. This, however, requires a lot of effort and assembly expertise from the designers of that compiler. Moreover, portability may suffer. A better option, therefore, seems the use of portable frameworks that give more control over cores than traditional explicit user-threading APIs do but not at the cost of having to generate assembly code.

  With Halle and Arbab, I did preliminary work in this direction [**J**HA14a, **J**HA14b], based on the *Proto Runtime Toolkit* (PRT) [Hal11, HC13], developed by Halle for his PhD thesis. PRT consists, among other components, of a run-time system for C code and APIs. On its start-up, the PRT run-time system seizes control of the available cores from the operating system, thereby gaining full responsibility for scheduling instructions onto those cores. These cores remain hidden from software engineers, though, through an API for managing PRT threads and a separate API for imposing custom scheduling policies. PRT-aware C code invokes the former API to instantiate units of parallelism, which the PRT run-time system subsequently schedules onto cores, without interference by the operating system. Bypassing the operating system in this way (including its rather heavy-weight scheduler), should result in better performance.

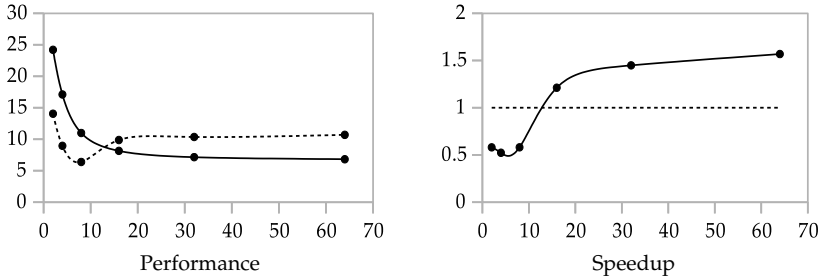

Figure 9.12: Left, performance (in thousands of cycles per `put`/`get` pair) as a function of the number of producers, denoted by k. Right, speedup relative to the dotted line as a function of k. Solid lines represent the performance/speedup of PRT-based compiler-generated code; dotted lines represent the performance/speedup of carefully optimized C+Pthreads-based hand-written code.

With Halle and Arbab, I performed preliminary experiments with a preliminary Reo-to-PRT compiler, and the preliminary results look encouraging [**J**HA14a, **J**HA14b]. For instance, Figure 9.12 shows the performance and speedup of two versions of a producers/consumer protocol for an increasing number of producers. For now, it remains unclear *exactly which* factors contribute to these promising results: perhaps PRT facilitates more efficient management of shared data structures (notably, queues) and/or context-switching between threads. Future research in this direction should target both the development of a more mature compiler and the conduction of more serious experiments to better understand and quantify the benefits of this approach.

- *Memory-centric compilation*

  In this thesis, I focused on the "algorithmic aspects" of FOCAML compilation, attempting to minimize the number of hardware instructions derived from compiler-generated code to reduce run time. As the experimental results for the NPB suite show, however, memory and caches seriously affect the run time of programs as well. Optimizing memory and cache usage constitutes a research field by itself, and for scientists working on those and related topics, this observation will not come as a surprise.

  Having built a foundation for the algorithmic aspects of FOCAML compilation, I firmly believe that future work should target improvements for optimizing memory and cache usage. Of course, at least code generated by the FOCAML-to-Java compiler should benefit from such improvements. However, I also foresee the need for a FOCAML-to-C compiler, to gain full control over memory allocation and make analyzing cache behavior easier, both of which the Java virtual machine obscures.

- *Type-aware compilation*

  In this thesis, I considered only untyped I/O operations: using the Java API for ports in Figure 1.9, every `put` sends an `Object`, while every `get` receives an `Object`. This untypedness makes worker code not only inelegant because of the many required **instanceof** checks and typecasts but also fragile: software engineers must know the type(s) of the objects returned by a `get` for every such invocation, and if they make a mistake, this goes unnoticed until their program crashes at run-time.

  Invented in the 1990s, *session types* seem a very suitable candidate for extending FOCAML with types [HVK98]. By annotating every port in a synchronization constraint in a constraint automaton with a type, this constraint automaton effectively becomes a global session type, amenable to projection on individual ports (as with multiparty session types [HYC08, CHY12]). The compiler can subsequently check the resulting per-port local session types against the actual usage of those ports in worker subprograms, to statically detect typing errors.

  Incidentally, with Santini and Arbab, I briefly sketched a different application of combining constraint automata with session types, namely projection of unprojectable choreographies [**J**SA15].

  Also incidentally, Ng et al. recently developed code generation technology for parametrized multiparty session types [NCY15], not unlike the work reported on in this thesis: using the work of Ng et al., software engineers implement protocols as global session types, after which a compiler generates MPI code and merges this code with existing computation code for workers. Though similar, the kinds of protocols supported by the DSL for interaction of Ng et al., called Pabble [NY14], seems limited compared to what FOCAML supports. For instance, FOCAML allows mixing synchronous and asynchronous interaction, which Pabble does not. Also, FOCAML has richer support for (functions and relations on) data. In contrast, Pabble better supports run-time parametrization, which FOCAML does not support whatsoever.

  In any case, the similarities between constraint automata and session types seem profound, and regardless of any practical motives, they require a better theoretical understanding. Future work in this direction should therefore target the development of constraint automata as global session types, both in theory and in practice.

The experimental results in this thesis show that compiler-generated code for intention-expressing protocol implementations can have performance comparable to hand-written code. I do not—and cannot—claim to achieve similar results in all possible scenarios, and much more experimentation and real-world case studies need to follow this initial piece of work. Nevertheless, by demonstrating promising first results, this thesis provides a justification for pursuing such future work. And that, perhaps, comprises the most valuable contribution of this thesis.