

Specialising Simulator Generators for High-Performance Monte-Carlo Methods*

Gabriele Keller¹, Hugh Chaffey-Millar², Manuel M. T. Chakravarty¹,
Don Stewart¹, and Christopher Barner-Kowollik²

¹ Programming Languages and Systems, School of Computer Science and
Engineering, University of New South Wales, {keller,chak,dons}@cse.unsw.edu.au

² Centre for Advanced Macromolecular Design, School of Chemical Sciences and
Engineering, University of New South Wales,
h.chaffey-millar@student.unsw.edu.au, c.barner-kowollik@unsw.edu.au

Abstract. We address the tension between software generality and performance in the domain of simulations based on Monte-Carlo methods. We simultaneously achieve generality and high performance by a novel development methodology and software architecture centred around the concept of a *specialising simulator generator*. Our approach combines and extends methods from functional programming, generative programming, partial evaluation, and runtime code generation. We also show how to generate parallelised simulators.

We evaluated our approach by implementing a simulator for advanced forms of polymerisation kinetics. We achieved unprecedented performance, making Monte-Carlo methods practically useful in an area that was previously dominated by deterministic PDE solvers. This is of high practical relevance, as Monte-Carlo simulations can provide detailed microscopic information that cannot be obtained with deterministic solvers.

1 Introduction

The tension between software generality and performance is especially strong in computationally intensive software, such as scientific and financial simulations. Software designers usually aim to produce applications with a wide range of functionality, which in the case of simulations means that they are highly parameterisable and, ideally, target different types of high-performance hardware. Scientific software is often used for new research tasks, and financial software is often employed for new products and new markets. In both cases, there is a high likelihood that the boundary of previous uses will be stretched. However, generality often comes with a performance penalty, as computations become more interpretive, require more runtime checks, and use less efficient data structures.

As an example, consider a computational chemistry simulation, using a Monte-Carlo method, that in its innermost loop repeatedly selects a random chemical

* This work was funded by the UNSW FRGP *High Performance Parallel Computing for Complex Polymer Architecture Design*.

reaction from a set of possible reactions. The probabilities of the reactions are determined by the reactions' empirical rate coefficients and the reactants' concentrations. If we aim for generality, the code will have the capability to handle a wide range of reactions. These reactions and their probabilities will be stored in a data structure that the code will have to traverse over and over when making a selection and when updating the concentrations of the various reactants. This is an interpretive process whose structure is not unlike that of an interpreter applying the rules of a term rewriting system repeatedly to the redexes of a term. The more general the rules handled by the interpreter, the higher the interpretive overhead, and the fewer rewrites will be executed per second.

To maximise performance, we need to eliminate all interpretive overhead. In the extreme, we have a program that hard-codes a single term rewriting system; i.e., we compile the term rewriting system instead of interpreting it. We can transfer that idea to the chemistry simulation by specialising the simulator so that it only applies a fixed set of reactions. Such specialisation can have a very significant impact on the performance of simulators that execute a relatively small part of the code very often; i.e., even minor inefficiencies of a few additional CPU cycles can add significantly to the overall running time. Giving up on generality is not an option. We also cannot expect the user to manually specialise and optimise the simulation code for the exact reactions and input parameters of a particular simulation; instead, we automate the generation and compilation of specialised code.

In programming languages, the move between interpreter and compiler is well known from the work on partial evaluation [1]. More generally, research on generative programming [2] and self-optimising libraries [3] introduced approaches to code specialisation in a range of application areas, including numerically intensive applications [4, 5]. Much of this work is concerned with providing general libraries that are specialised at compile time. In this paper, we transfer these ideas from libraries to applications and combine them with runtime code generation and a development methodology based on prototyping.

More precisely, we introduce a novel software architecture for simulators using Monte-Carlo methods. This architecture uses *generative code specialisation* to reconcile generality and performance in a way that is transparent to the end user. Specifically, instead of an interpretive simulator in a low-level language, we implement a *specialising simulator generator* in a functional language and use it to generate optimised C code specialised for a particular simulator configuration. Moreover, we outline how a specialising simulator generator can be developed by way of prototyping the simulator in a functional language.

We discuss the design of specialising simulator generators and the parallelisation of the generated simulators for Monte-Carlo methods. Moreover, we demonstrate the practical relevance of our approach by a concrete application from computational chemistry, namely a simulator for polymerisation kinetics using a Markov-chain Monte-Carlo method.

We achieved unprecedented performance for the polymerisation simulator. For the first time, it makes Monte-Carlo methods practically useful in an area

that is dominated by deterministic PDE solvers. This is of high practical relevance, as Monte-Carlo simulations can provide detailed microscopic information about generated polymeric species. Such microscopic information—which is not available from deterministic simulators, specifically those using the h-p-Galerkin method [6, 7]—includes (yet is not limited to) information on polymer species with more than one chain length index (i.e., star polymer systems often applied in polymeric drug, gene, and vaccine delivery systems [8]), cross-linking densities, and branching in complex polymer networks as well as detailed information on copolymer compositions. Finally, we demonstrate good parallel speedups for our Monte-Carlo method, whereas no parallel h-p-Galerkin solvers for polymerisation exist to the best of our knowledge. Parallelisation is a pressing practical problem, as multicore counts have replaced clock rates as the main parameter increased in new processor generations.

In summary, our main contributions are the following:

- A development methodology and software architecture using generative code specialisation for Monte-Carlo simulations (Section 2 & Section 3).
- A parallelisation strategy for Markov-chain Monte-Carlo methods (Section 4).
- A detailed performance evaluation of our Monte-Carlo simulator for polymerisation kinetics, which shows that the application of methods from functional programming can lead to code that is significantly more efficient than what can be achieved with traditional methods (Section 5).

As aforementioned, we build on a host of previous work from generative programming, partial evaluation, and runtime code generation. We discuss this related work as well as other work on polymerisation kinetics in Section 6.

2 A generative code specialisation architecture

Simulations based on Monte-Carlo methods are popular in the study of complex systems with a large number of coupled degrees of freedom, this includes applications ranging from computational physics (e.g., high energy particle physics) to financial mathematics (e.g., option pricing). The underlying principle is the *law of large numbers*; that is, we can estimate the probability of an event with increasing accuracy as we repeat a stochastic experiment over and over. This principle applies to any system that we can model in terms of *probability density functions (PDFs)*. This includes the numerical approximation of purely mathematical constructs with no apparent stochasticity or randomness, such as approximating the value of π or the numerical integration of complex functions [9].

Monte-Carlo methods use probability density functions to drive sampling during a simulation. This can be the repeated evaluation of a function at random points, e.g. to integrate it numerically, or it can be a sequence of system state changes, each of which occurs with a certain probability—e.g., a solution of chemical reactants changes depending on the likelihood of the reactions.

To exploit the law of large numbers, all Monte-Carlo simulations repeat one or more stochastic experiments a large number of times, while tallying the results,

and possibly continuously evolving some system state and computing variance reduction information. With increasing complexity of the system and increasing need for precision, more and more stochastic experiments need to be performed. This highly repetitive nature of Monte-Carlo simulations is one of the two key points underlying the software architecture that we are about to discuss.

The second key point is that, in many application areas, Monte-Carlo simulations should admit a wide variety of different simulations. For example, in reaction kinetics, we would like to handle many different chemical reactions and, in financial modelling, we would like to model many different financial products. We call this the *configuration space*. The more general a simulator, the larger its configuration space. To explore new chemical processes and new financial products, we need to have short turn-arounds in an interactive system to explore a design space by repeatedly altering configurations. In contrast, when the user finds a point in the design space that they want to simulate in more detail, a simulation may run for hours or even days.

In summary, the two crucial properties of Monte-Carlo simulations guiding the following discussion are thus:

- *Property 1*: The simulation repeats one or more stochastic experiments and associated bookkeeping a large number of times, to achieve numeric accuracy.
- *Property 2*: It simulates complex systems with a large number of degrees of freedom and a rich configuration space.

2.1 The classical approach: a simulator in C, C++, or Fortran

Property 1 makes Monte-Carlo simulators very computationally intensive—e.g., sophisticated simulations in the domain of polymerisation kinetics can run for hours or even days. Hence, manually optimised simulator code in low-level languages like C, C++, and Fortran is the state of the art, and the use of functional languages is out of the question, unless the same level of performance can be achieved.

While Property 1 encourages the use of a low-level language, the number of optimisations that can be performed in such a language is limited by Property 2. A simulator in a low-level language must be sufficiently generic to handle a large configuration space in which it has to evaluate functions with a large number of inputs. In other words, the code in the repeatedly executed inner loop will be complex and possibly traverse sophisticated data structures. However, given the number of repetitions, each CPU cycle counts significantly towards the final running time. Additional instructions required to implement a more general solution lead to notable inefficiencies compared to specialised implementations.

To illustrate this situation, consider reaction kinetics again. Each reaction occurs with a probability that depends on the relative concentration of the various reactants. If it occurs, it will consume one or more reactants and release new reactants into the solution. A Monte-Carlo simulator will have to keep track of these concentrations and the associated reaction probabilities. It has to select reactions according to the implied probability density function. The reactions

have to be modelled in a data structure and the more variations we allow, the more interpretive the process in the inner loop will be. In other words, the larger the configuration space, the slower the simulator.

2.2 A generative approach

This situation calls for a generative approach. The simulator has a large configuration space and its inner loop will be executed many times for a single configuration, making it worthwhile to specialise the inner loop for one configuration, thus, effectively giving us a custom simulator for one problem. This specialisation has to be transparent to the user and has to occur in an interactive environment to admit exploratory uses. Hence, we propose the use of online generative code specialisation: that is, depending on user input, the application specialises its inner core to produce highly optimised code, and then, dynamically loads and links that code into the running application.

2.3 From Haskell to C to a C generator

We propose the following development methodology:

1. Implement a *prototype simulator* in a functional language like Haskell as an executable specification, to explore alternative designs.
2. Implement a *specialised simulator* in a low-level language like C by specialising the prototype simulator for one or more concrete simulator configurations. Use it to explore possible low-level optimisations including selecting appropriate imperative data structures.
3. Replace the simulator core of the prototype simulator with a *simulator generator* that, when executed on the same configuration, produces the specialised simulator we manually implemented in the previous step.

Both the specialised simulator and the simulator generator are validated against the prototype simulator, which is much more compact and easier to reason about. This development methodology is especially worthwhile when extending the boundaries of existing simulators, as was the case in the project in which we developed it. We had undertaken to implement the first simulator to compute detailed microscopic information for reactions of star polymers and to achieve higher levels of efficiency than existing simulators. Existing systems either oversimplified complex molecular structures [10] or lacked performance and generality [6]. Moreover, no existing system was parallelised, and we aimed for good scalability on high-latency networks, such as Ethernet-based PC clusters.

Consequently, we developed a new simulator from scratch in Haskell and placed particular emphasis on data structures for the system state that are sufficiently small to enable cheap network transmission, while still allowing for a highly efficient innermost loop of the simulator. Subsequently, we parallelised the prototype simulator using a generic master/worker skeleton based on the standard network library distributed with the Glasgow Haskell Compiler (GHC).

Only after we convinced ourselves of the efficiency of the data structures and algorithms implemented in the prototype using GHC’s heap profiler and scalability benchmarks on a PC cluster, did we turn to Step 2 of our development methodology and implement a specialised simulator in C. This enabled problem-oriented, explorative development without too much attention to low-level details and avoiding premature optimisations.

The concrete specialisation opportunities in Step 2 are domain-specific and to find them specialisation-time data must be separated from runtime data; i.e., the variables in the inner loop of the simulator that are fixed by the simulator configuration and do not change between loop iterations must be identified. In our case, this was the number and type of reactions and reactants; i.e., we can pre-calculate and hardcode all possible one-step changes of the system state.

In Step 3, we exploit the fact that the performance-critical code is in the inner loop of the simulator. Hence, the simulator generator does not need to generate the C code for an entire simulator. Instead, it inserts into a simulator skeleton only system-state initialisation code, state-changing code of the inner loops, and other configuration-depended code, such as some I/O. For example, in the simulator for polymerisation kinetics, the most important piece of specialised and generated code is the body of one C `switch` statement that, depending on the randomly selected reaction, effects the update of the system state—c.f., Section 3.2.

2.4 Runtime compilation & loading

The use of a simulator generator implies runtime code generation and compilation. In our case, the latter consists of invoking a standard C compiler, such as GNU’s `gcc` or Intel’s `icc` to compile the generated specialised simulator. Given the long running times of typical Monte-Carlo simulations, we can even amortise time-consuming compilation with costly optimisations enabled. In addition to the simplification of data and control structures explicitly performed by the specialising simulator generator, the C compiler can exploit the fact that many of the variables of the generic simulator are now embedded as constants. This leads to additional constant folding, loop unrolling, etc.

After code generation, the simulator executable can be executed in a separate process or, as in our implementation, loaded into the main application using dynamic code loading [11]. The latter is attractive for interactive applications that, for example, animate the simulation graphically.

3 Generative Monte-Carlo Methods

We will now discuss two examples of Monte-Carlo methods. The first is a toy example to illustrate the basic structure of Monte-Carlo methods and the second is a real-world application, namely the aforementioned polymerisation simulator.

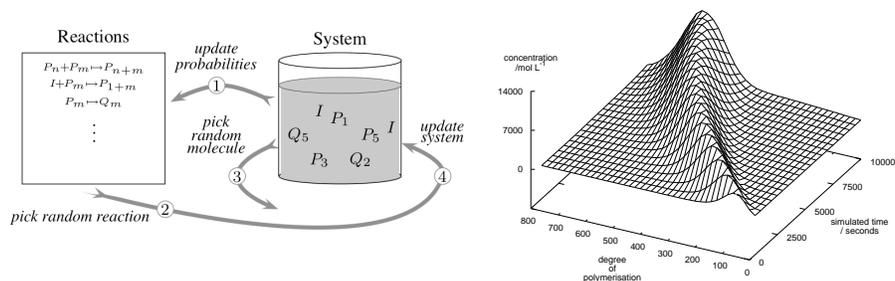


Fig. 1. Structure of polymerisation simulator (left) and computed molecular weight distribution (right)

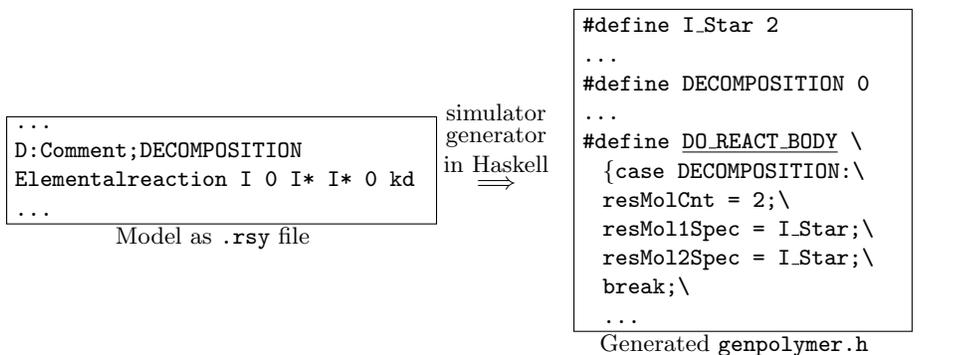
3.1 Computing π

The probably simplest Monte-Carlo method is the one to compute an approximation of π . We know that the area of a circle is $A = \pi r^2$. Hence, $\pi = A/r^2$; i.e., $\pi/4$ is the *probability* that a point picked at random out of a square of side length $2r$ is within the circle enclosed by that square. As explained in the previous section, the fundamental idea underlying Monte-Carlo methods is to *estimate the probability of an event with increasing accuracy by repeating a stochastic experiment over and over*. Here the stochastic experiment is to pick a point in the square at random, and we use that experiment to approximate the probability that picked points lie inside the circle. By multiplying that approximated probability with 4, we approximate π .

3.2 Modelling polymerisation kinetics

Our main example is a simulator for polymerisation kinetics. This is a complete application incorporating a significant amount of domain knowledge; hence, we cannot sensibly display and explain its source code in a paper. However, the code is publicly available³ for inspection and use—in fact, we have two versions of the application, the prototype simulator (entirely in Haskell) and the specialising simulator generator (with the generator in Haskell and the simulator skeleton in C). In the following, we will discuss the innermost loop of the simulator, containing all the performance-critical code, as well as sketch the work distribution between the simulator generator (in Haskell) and the simulator skeleton (in C). Due to space constraints and to avoid having to explain too much of the chemistry, we will abstract over many of the simulator data structures; more details, from a chemist’s perspective, are in a companion paper [8].

³ <http://www.cse.unsw.edu.au/~chak/project/polysim/>



```
#include "genpolymer.h"
void oneReaction () { // updates global system state
    int reactIndex, mol1Len, mol2Len;           // consumed molecules
    int resMol1Spec, resMol2Spec,             // produced...
        resMol1Len[CHAINS], resMol2Len[CHAINS]; // ... molecules
    int resMolCnt = 1;                         // number of produced; default is one

    ① Compute reaction probabilities as product of the reaction's statically determined
        relative probability and the current concentration of the reactants involved.
    updateProbabilities ();

    ② Randomly pick a reaction according to the current reaction probabilities; e.g., from
        the list in Figure 1 (left), we might pick  $P_n + P_m \mapsto P_{n+m}$ .
    reactIndex = pickRndReact ();

    ③ Randomly pick the molecules involved in the reaction. In some systems polymers
        with different chain lengths react with different probability. For the reaction  $P_n +
        P_m \mapsto P_{n+m}$ , we have to pick two random chain lengths  $n$  and  $m$ .
    mol1Len = pickRndMol (reactToSpecInd1 (reactIndex));
    if (consumesTwoMols (reactIndex))
        mol2Len = pickRndMol (reactToSpecInd2 (reactIndex));

    ④ Compute reaction products, and update the concentration of molecules accord-
        ingly; for our example, we add  $P_{n+m}$ . The consumed molecules,  $P_n$  and  $P_m$ , were
        already removed in Step ③. Also, the system clock is incremented.
    switch (reactIndex) // compute reaction products
        DO_REACT_BODY // defined by simulator generator; sets resMol1Spec etc.
    incrementMolCnt (resMol1Spec, resMol1Len);
    if (resMolCnt == 2)
        incrementMolCnt (resMol2Spec, resMol2Len);
    advanceSystemTime (); // compute  $\Delta t$  of this reaction
}
```

Fig. 2. Task of the specialising generator (top) and simulator skeleton (bottom)

Chemical reactions in four steps. The four steps performed by the innermost loop of the simulator are illustrated in Figure 1 (left): ① computation of the reaction probabilities; ② random selection of a reaction; ③ random selection of the consumed molecules; and ④ update of the system with the produced molecules. These steps are further explained in Figure 2 (bottom), where the corresponding C code of the simulator skeleton, in form of the function `oneReaction()`, is also given. In fact, the C code of `oneReaction()` is almost the same for a generic simulator and a specialised simulator. The main difference is the `switch` statement and its body `DO_REACT_BODY`, which is a placeholder for code inserted by the specialising simulator generator—we will come back to this below.

Chemistry basics. Generally, polymerisation kinetics features two kinds of molecules: *simple molecules* that have no chain length (such as the I in the example) and *polymers* that have a chain length specified by a suffix (such as the P_n in the example). Polymers with multiple chains, are called *star polymers*, which are often applied in polymeric drug, gene, and vaccine delivery systems. Our simulator is the first to compute detailed microscopic information for this important class of polymers. In our simulator, a reaction consumes one or two molecules and also produces one or two molecules; this is encoded in the conditionals in Step ③ and ④, respectively. Reactions involving polymers are specific w.r.t. the type of molecules involved, but are parametrised over the chain length; e.g., $P_n + P_m \mapsto P_{n+m}$ consumes two polymers with lengths n and m and produces one with length $n + m$. In Step ③, the probability always depends on the current concentration of the molecules of varying chain lengths, but may be adjusted by a factor that models how the chain length influences the reactivity of a molecule.

Computing molecular weight distributions. Each invocation of `oneReaction()` corresponds to one chemical reaction and to one stochastic experiment of the Monte-Carlo method. These reactions slowly alter the concentration of polymer molecules with particular chain lengths. An indirect measure of chain length is molecular weight, and Chemists like to see the evaluation of polymer concentrations in the form of *molecular weight distributions*, as in Figure 1 (right), which was computed by our simulator.

Specialisation. Figure 2 (top) illustrates the task of the specialising simulator generator: it reads a reaction specification, as an `.rsy` file, and compiles it into a C header file `genpolymer.h`. This header file contains all reaction-specific *data and code*, and is `#included` by the simulator skeleton. To avoid overheads due to sophisticated data structures, the different types of molecules and reactions are simply encoded as integers (e.g., the `#defines` for `I_Star` and `DECOMPOSITION` in the displayed code fragment). This, most importantly, enables the use of a simple `switch` statement to compute the produced molecules in Step ④. The body of that switch statement is one of the most important pieces of generated code and `#defined` by `DO_REACT_BODY` (underlined). The code fragment in Figure 2 (top) gives the switch `case` for a simple decomposition reaction. The cases for polymers are somewhat more involved, as chain lengths have to be computed.

In Section 1, we discussed that the aim of a specialising simulator generator is to eliminate interpretive overhead. In the polymerisation simulator, we achieve this by (a) hardcoding the reactants involved in each reaction and (b) using an array with fixed size and layout that maps molecule types (including chain length for polymers) to the number of that type of molecule in the system. Point (a) is crucial to be able to use a `switch` statement with a few simple operations per `case`/reaction in Step ④. In contrast, a generic simulator needs to consult a dynamic reaction table to achieve the same. For a complex reaction, this specialisation reduces the number of executed assembly instructions in our code from 31 (including 5 branches) to 6 (including one branch and one indirect jump) when compiled with the Intel C compiler.

3.3 A specialiser in Haskell

In general, the reaction specifications are significantly more involved than the `.rsy` file fragment in Figure 2 (top). The file format originates from the PREDICI system [12, 13] and, by using it, we can create chemical models with PREDICI’s graphical frontend. Parsing reaction specifications, extracting the information necessary for the simulator generation, and generating the specialised C data structures and code fragments makes heavy use of algebraic data types, pattern-matching, list manipulation, and higher-order functions; i.e., it is the type of code where functional languages excel.

As an example, take the fragment of the specialisation code in Figure 3, which is a simplified version of the actual code used in the generator. We model chemical reactions with the type `Reaction`, which specifies the involved kinds of molecules and the reaction’s rate coefficient (i.e., the probability of that reaction happening in dependence on the concentrations of the involved reactants). Molecules can be of three kinds determined by the data type `Kind` (i.e., simple molecules, linear polymer chains, and star polymers). Moreover, the variant `NoSpec` is used to when any of the two reactant or product slots in a `Reaction` are not used (e.g., reactions where two reactant molecules result in a single product molecule). In addition to `Reactions`, we have `ReactionSchemas` that determine the length of polymers produced by a reaction using values of type `ResLen`. Figure 3 only shows part of the definition of `ResLen`; in general, it models arithmetic expressions over the two input chain lengths with support for inclusion of random variables. The latter is required to represent the splitting of a polymer chain at a random position. For star polymers, `ResLen` calculations become slightly more complicated, as we need to express a sequence of calculations on a star’s chains. As an example, take the reaction discussed previously: $P_n + P_m \mapsto P_{n+m}$. Its representation as a `ReactionSchema` is by the value `RS (Poly, Poly) (Poly, NoSpec) (AddLen FstLen SndLen, NoLen)`.

Figure 3 also gives the code for the function `specialiseReacts`, which uses a list comprehension to combine each reaction with its matching schema, as determined by the auxiliary function `matchesSchema`. The function then derives from each matching reaction-schema pair a specialised reaction `SpecReaction`. These

```

— Concrete reaction:
data Reaction =
  Reaction
    Name
    RateCoef — rate coefficient
    Kind Kind — reactants
    Kind Kind — products

— Various kinds of molecules:
data Kind
  = Simple — regular molecule
  | Poly — linear polymer
  | Star Arms — star polymer
  | NoSpec — not present

— How to compute polymer lengths:
data ResLen — Reaction product length:
  = FstLen — length of 1st reactant
  | SndLen — length of 2nd reactant
  | AddLen ResLen ResLen — sum of 1st & 2nd
  | ConstLen Int — constant length
— some further variants

type RateCoef = Double
type Arms = Int

— Desc. of a specialised reaction:
data SpecReaction =
  SpecReaction
    Name
    Kind Kind — reactants
    Kind ResLen — product #1
    KInd ResLen — product #2

specialiseReacts :: [Reaction] -> [ReactionSchema] -> [SpecReaction]
specialiseReacts reactions schemata = map specialiseReaction reactSchema
  where
    reactSchema = [(r,s) | r <- reactions, s <- schemata, matchesSchema r s]
    specialiseReaction ((Reaction name _ react1 react2 prod1 prod2),
      (RS name _ _ (resLen1, resLen2)))
      = SpecReaction react1 react2 prod1 resLen1 prod2 resLen2

matchesSchema :: Reaction -> ReactionSchema -> Bool
matchesSchema r s = (check whether reaction r fits schema s)

```

Fig. 3. Fragment of the specialisation code

specialised reactions are subsequently fed into the code generator to produce the code for `DO_REACT_BODY`.

The specialiser generates all C code that is dependent on the type of molecules and reactions in the system. In the C skeleton, these code fragments are represented by C pre-processor macros, such as `DO_REACT_BODY`. Similarly, all system parameters (e.g., the maximum number of chains per molecule `CHAINS`) and tag values to represent molecules and reactions (e.g., `I_Star` and `DECOMPOSITION`) are computed by the specialiser and emitted as macro declarations. All parameters of reactions, such as rate coefficients, are hardcoded into the macros that form the system initialisation code. After macro expansion, the C compiler is presented with a program manipulating integers, floats, and arrays that has a

simple control structure and is littered with constant values—i.e., it has ample opportunities to apply standard compiler optimisations.

4 Parallelisation

Monte-Carlo methods where the individual stochastic experiments are independent—such as the approximation of π —are almost trivial to parallelise. The large number of independent stochastic experiments can be easily distributed over multiple processors, as long as we ensure that the statistic properties of our source of randomness are robust with respect to parallel execution. The only communication between the parallel threads is at the end of the simulation when the local results need to be combined into a global result. This can be efficiently achieved with standard parallel reduction operations (e.g., parallel folds).

The parallelisation of the polymerisation simulator is more involved. The probability of the various types of reactions changes over time with the changing concentration of the molecules involved in a reaction. In other words, the stochastic experiments are dependent and we have a *Markov-chain* Monte-Carlo method. Previous proposals to parallelise Monte-Carlo polymerisation involve running the same simulation several times and calculating the average [6]. Different instances of the same simulation can run in parallel, and the average over the individual results will be more accurate than any single simulation. However, it is important to keep in mind that running the same simulation ten times will not, in general, lead to a result of the same quality as a single simulation of a system ten times the size, since the concentration of some of the reactants is so low, that, for small systems there would be less than one molecule available and the fact that the simulation is discrete would distort the result. This is not just a theoretical problem, we have observed it in production-sized simulations [8].

We solved this problem by exploiting the following observation: a common simplification in simulations of reaction kinetics is to abstract over the spatial location of the molecules in the system. If two molecules are far apart, they would, in reality, be less likely to react. We can use this for parallelisation by splitting the system into several subsystems, running the simulation of the subsystems independently in parallel, but ensuring that we *mix*—i.e., gather, average, and re-distribute—the states of the subsystems with sufficient frequency to model the Brownian motion of the molecules. Thus, we parallelise the application without compromising the quality of the result. The speed up is slightly less than for a trivially parallel Monte-Carlo simulation, as mixing triggers communication. However, as the following benchmarks show, the parallelisation is still very good.

Although, our approach of regularly averaging over a set of Monte-Carlo simulations running in parallel was motivated by the physical intuition of spatial separation and Brownian motion in a liquid, we conjecture that the same approach to parallelisation is more generally applicable to Markov-chain Monte-Carlo methods. Regular averaging over a set of parallel simulations will improve the accuracy of the intermediate results and so usually accelerate convergence. However, the ideal frequency for mixing will depend on the concrete application

5 Performance

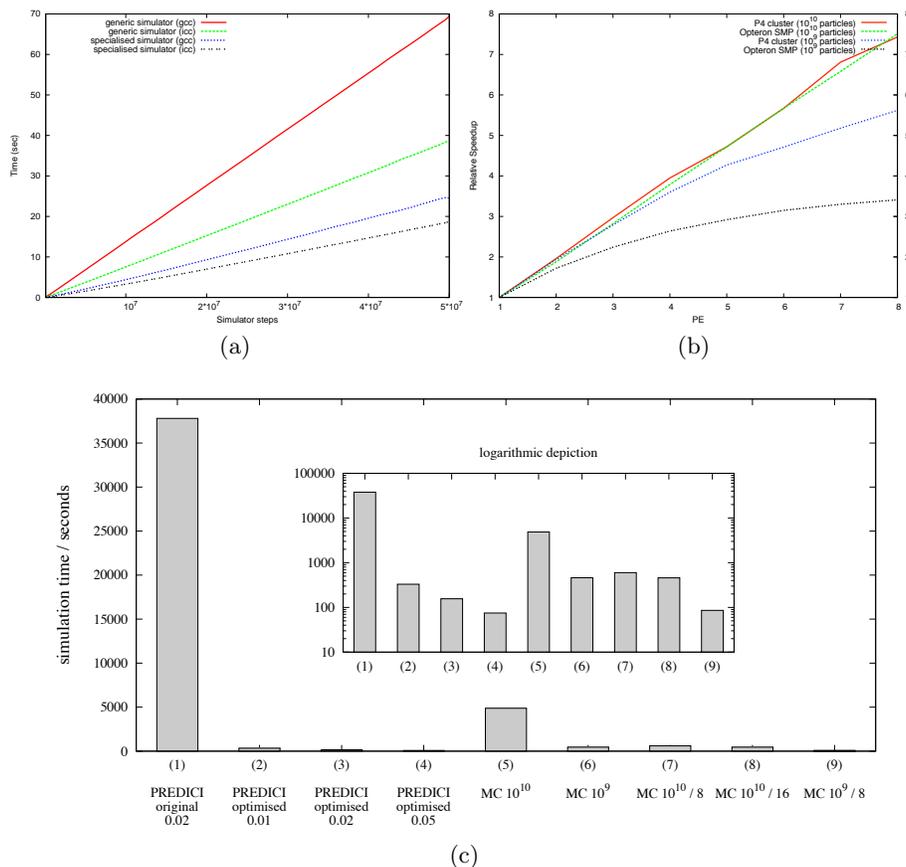


Fig. 4. (a) Generic versus specialised polymerisation simulator; (b) parallel speedup on shared memory and cluster hardware; (c) deterministic versus Monte-Carlo simulator

To quantify the performance benefits of our approach, we will now discuss three aspects of the performance of our system: (a) the performance improvement due to specialisation, (b) the speedup due to parallelisation, and (c) the performance relative to existing systems. We used two types of hardware for benchmarking the Monte-Carlo code: (1) a PC cluster with Intel Pentium 4, 3.2GHz, processors connected by GigaBit Ethernet (called *P4 cluster* in the following) and (2) a shared-memory computer containing 8x dualcore AMD Athlon 64 3200+, 2.2GHz, processors connected by a HyperTransport 1.0 bus (called *Athlon SMP* in the following). Both systems ran GNU/Linux. All C code was

compiled with the Intel C compiler `icc`, version 9.1, except where we explicitly noted the use of the GNU C compiler `gcc`, version 4.1. We generated random numbers using Matsumoto & Nishimura’s Mersenne Twister MT19937, version 2002/1/26. All communication between parallel processes was via MPICH2. We only used one core per processor of the Athlon SMP system to avoid skewing the benchmarks by bus contention on the processor-memory interface.

Generic versus specialised simulator. We start by quantifying the performance benefit gained from specialisation. Figure 4(a) plots the running time of a generic simulator, manually implemented in C, and a simulator produced by our specialising simulator generator. For this particular simulation (a simple model of the polymerisation of bulk styrene [8, Fig. 4]), the specialisation leads to a performance improvement between a factor of 2 and 2.5, depending on whether we compile with `gcc` or `icc`. The benchmark was executed on one node of the P4 cluster—`icc` produces only marginally better code than `gcc` on AMD processors.

Accurate comparisons with other Monte-Carlo simulators for polymerisation kinetics are difficult, as published data is scarce and no software is available for benchmarking. However, we reproduced the methyl acrylate model of Drache et al. [6] to the best of our knowledge and the performance of the processors in our Athlon SMP system is very similar to the hardware used by Drache et al. The results suggest that the performance of our generic simulator, at least for Drache’s methyl acrylate model, is essentially the same as that of Drache’s simulator. In summary, our novel specialising simulator generators advance the state-of-the-art in uniprocessor performance of Monte-Carlo simulators for polymerisation kinetics by around a factor of two.

Parallel speedup. Figure 4(b) graphs the speedup of the specialised simulator for the simple styrene model for both the P4 cluster and the Athlon SMP. It does so for a solution containing 10^9 and a solution containing 10^{10} particles. With 10^{10} particles, we get very good scalability (close to 7.5 for 8 PEs) on both architectures. Given the rather simple commodity hardware of the P4 cluster, this is a very positive result. For 10^9 particles, scalability is clearly limited. In essence, the 10^9 particle system is too small to efficiently utilise more than 4 PEs in the cluster and 5 PEs in the SMP system.

Deterministic versus Monte-Carlo simulator. From an end-user perspective, it is irrelevant whether a simulator uses a Monte-Carlo or a deterministic method, such as the popular h-p-Galerkin method to compute a molecular weight distribution by solving partial differential equations. What counts is (a) the amount of detail in the information produced by the simulator and (b) the speed with which the information is computed. Monte-Carlo methods are attractive as they can compute information detail that is not available from deterministic simulators, such as information on polymer species with more than one chain length index, cross-linking densities, and branching in complex polymer networks as well as information on copolymer compositions [6, 7]. However, Monte-Carlo methods are

rarely used in practice as they have until now required much longer simulation times.

We already showed that our use of specialising simulator generators improves the performance over generic Monte-Carlo simulators, such as Drache’s system [6], even for simple polymerisation models. However, the acid test for the practical usefulness of our approach is a comparison with the fastest available deterministic solvers. The undisputed leader in this space is the commercial package PREDICI (of CiT GmbH) [12, 13]. We originally benchmarked the distributed version 6.36.1 of PREDICI, but after supplying CiT with a draft of our companion paper [8] and after some discussion about the reasons for the poor performance of PREDICI between CiT and us, CiT supplied us with a custom optimised version of PREDICI, which performs much better on the complex styrene model [8, Fig. 5] used for the benchmarks. All PREDICI benchmarks were on a 3.4GHz Xeon machine running Windows; i.e., slightly faster hardware than the P4 cluster, which we measured our code on.

The results are depicted in Figure 4(c). The bars labelled “MC 10^9 ” and “MC 10^{10} ” are for the uniprocessor performance of our specialised Monte-Carlo simulator for 10^9 and 10^{10} particles, respectively, on the P4 cluster. “MC $10^9/8$ ” and “MC $10^{10}/8$ ” are for the same code running in parallel on 8 PEs. “MC $10^{10}/16$ ” is also *only on 8 PEs*, but using 16 processes to gain some further slight speedup by using the HyperThreading capabilities of the processors of the P4 cluster.

Our uniprocessor performance is several times better than the original performance of PREDICI. However, after the optimisation, PREDICI improved by two orders of magnitude⁴ and we need to use 4 PEs to achieve roughly the same performance with Monte-Carlo. (At this stage, it is not entirely clear how general the optimisation of PREDICI is.) The number of particles for Monte-Carlo and the accuracy value of PREDICI that give comparable results depend on the simulated model. In this benchmark, Monte-Carlo with 10^9 particles is comparable to PREDICI with an accuracy of 0.02; so, with 8 PEs, we are nearly twice as fast—note that this is on a cheap Ethernet-based cluster.

In summary, our combined performance improvement by specialisation and parallelisation has made Monte-Carlo methods for polymerisation kinetics a practical alternative to deterministic solvers for complex models; especially so, when microscopic detail is of interest that deterministic solvers cannot produce. Moreover, with the increasing number of cores in commodity processors, the lack of parallelisation will be an ever more serious obstacle for deterministic solvers.

6 Related Work

Generative programming. FFTW [4] was clearly an inspiration for us and shares with our approach the use a functional language to generate optimised low-level code. This is, however, where the similarity ends. FFTW provides a library of

⁴ PREDICI’s dramatic improvement is due to an algorithmic change, after studying the behaviour of our complex styrene model with our Monte-Carlo simulator.

composable solvers, whereas we presented an application architecture. FFTW heavily relies on dynamic optimisations, whereas our approach is purely static.

ATLAS [14] applies techniques similar to FFTW to the implementation of optimised BLAS algorithms. In particular, it runs benchmark kernels at installation time to determine important architecture parameters and compose optimised BLAS routines from a range of kernels using a code generator.

Partial evaluation. Partial evaluators for C, such as C-Mix [15] and Tempo [16], share with our work the objective of high-performing, yet easily maintainable code, and they also rely on the C compiler to apply standard optimisations that have been enabled by specialisation. However, there are also significant differences: We changed the core data structures when moving from generic to specialised simulator. It is unlikely that the same kind of changes could have been achieved automatically, as we relied on domain knowledge. Our specialising simulator generator makes heavy use of higher-order functions, pattern matching and algebraic data types. These language features are not or not particularly well supported in C, and so would not be available when implementing a fully generic simulator in C (for specialisation by a partial evaluator tool); i.e., we would lose the benefit of doing most of the algorithmic work in a declarative language.

C++ templates as a substrate for partial evaluation, as in [17], allow the addition of domain specific information, and it would be interesting to investigate if it is possible to get similar results as we have with our approach. However, it would definitely be necessary to push the limits of C++ template programming, a technique which can be fairly tricky and error prone.

Polymerisation kinetics. We based the development of our Monte-Carlo method on [18–20, 6]. Although Drache et al. [6] use multiple processors to run independent simulations in parallel, to increase the accuracy of the result, we are the first to implement a parallel version of a *single* simulation. We discussed this and compared our performance with Drache’s in Section 5.

7 Conclusion

The classic approach to high-performance Monte-Carlo simulations is to design, implement, and optimise a generic simulator in an imperative language. Using a functional language, we outperformed the classic approach in generality (we fully simulate star polymers) and execution time (we are more than twice as fast on a uniprocessor and scale well in parallel) by exploring the design space with a functional prototype and generating specialised data structures and performance critical code. For the first time, Monte-Carlo methods are now a viable alternative to deterministic solvers for polymerisation kinetics.

Acknowledgements. We would like to thank Dr. Michael Wulkow from CiT GmbH for the stimulating and interesting discussions that have benefited both the PREDICI and parallel Monte Carlo approach. We also thank the anonymous reviewers for their helpful comments.

References

1. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice Hall International (1993)
2. Czarnecki, K., Eisenecker, U.W.: *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley (2000)
3. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: *Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. (1998)
4. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. *Proceedings of the IEEE* **93**(2) (2005) 216–231
5. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: *Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, SIAM Press (1998)
6. Drache, M., Schmidt-Naake, G., Buback, M., Vana, P.: Modeling RAFT polymerization kinetics via Monte Carlo methods: cumyl dithiobenzoate mediated methyl acrylate polymerization. *Polymer* (2004)
7. Tobita, H., Yanase, F.: Monte Carlo simulation of controlled/living radical polymerization in emulsified systems. *Macromolecular Theory and Simulation* (2007)
8. Chaffey-Millar, H., Stewart, D.B., Chakravarty, M., Keller, G., Barner-Kowollik, C.: A parallelised high performance Monte Carlo simulation approach for complex polymerization kinetics. *Macromolecular Theory and Simulations* **16**(6) (2007) 575–592
9. Robert, C., Casella, G.: *Monte Carlo Statistical Methods*. Springer Verlag (2004)
10. Chaffey-Millar, H., Busch, M., Davis, T.P., Stenzel, M.H., Barner-Kowollik, C.: Advanced computational strategies for modelling the evolution of full molecular weight distributions formed during multiarmed (star) polymerisations. **14** (2005)
11. Pang, A., Stewart, D., Seefried, S., Chakravarty, M.M.T.: Plugging Haskell in. In: *Proc. of the ACM SIGPLAN Workshop on Haskell*, ACM Press (2004) 10–21
12. Wulkow, M.: Predici. <http://www.cit-wulkow.de/tbapred.htm> (2007)
13. Wulkow, M.: The simulation of molecular weight distributions in polyreaction kinetics by discrete galerkin methods. *Macromolecular Theory Simulation* **5** (1996)
14. Whaley, R.C., Petitot, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. *Parallel Computing* **27**(1–2) (2001) 3–35
15. Arne J. Glenstrup, H.M., Secher, J.P.: C-Mix – specialization of C programs. In: *Partial Evaluation: Practice and Theory*. (1999)
16. Consel, C., Hornof, L., Lawall, J.L., Marlet, R., Muller, G., Noy, J., Thibault, S., Volanschi, E.N.: Tempo: Specializing systems applications and beyond. *ACM Computing Surveys* **vol 30**(no 3) (September 1998)
17. Veldhuizen, T.L.: C++ templates as partial evaluation. In: *Partial Evaluation and Semantic-Based Program Manipulation*. (1999) 13–18
18. Lu, J., Zhang, H., Yang, Y.: Monte carlo simulation of kinetics and chain-length distribution in radical polymerization. *Macromolecular Theory and Simulation* **2** (1993) 747–760
19. He, J., Zhang, H., Yang, Y.: Monte carlo simulation of chain length distribution in radical polymerization with transfer reaction. *Macromolecular Theory and Simulation* **4** (1995) 811–819
20. Prescott, S.W.: Chain-length dependence in living/controlled free-radical polymerizations: Physical manifestation and Monte Carlo simulation of reversible transfer agents. *Macromolecules* **36** (2003) 9608–9621