Optimizing Space of Parallel Processes

Manfred Schmidt-Schauß*
Goethe-University
Frankfurt am Main
schauss@ki.cs.uni-frankfurt.de

Nils Dallmeyer*
Goethe-University
Frankfurt am Main
dallmeyer@ki.cs.uni-frankfurt.de

This paper is a contribution to exploring and analyzing space-improvements in concurrent programming languages, in particular in the functional process-calculus CHF. Space-improvements are defined as a generalization of the corresponding notion in deterministic pure functional languages. The main part of the paper is the $O(n \cdot \log n)$ algorithm SPOPTN for offline space optimization of several parallel independent processes. Applications of this algorithm are: (i) affirmation of space improving transformations for particular classes of program transformations; (ii) support of an interpreter-based method for refuting space-improvements; and (iii) as a stand-alone offline-optimizer for space (or similar resources) of parallel processes.

Keywords. space optimization, parallel processes, space improvements, call-by-need evaluation, concurrency

1 Introduction

The main motivation for investigating the common space consumption of parallel processes is our investigation into space optimizations and space improvements in concurrent languages. A special but important subcase are parallel processes (threads) which are independent or have only rare interactions by a controllable form of synchronization. An algorithm to compute a space-minimal execution sequence of a set of given parallel and independent processes would be a first step in space optimizations and of great help for the analysis of space-improvements and -optimizations of programs.

The space consumption of threads that are evaluated in parallel is as follows. We assume that there is a common memory, where the state of every process is stored. In addition we assume that the storage occupation of processes is independent of each other. The model for processes is rather abstract insofar as it only models the thread-local space as a sequence of numbers. Note that even in the case of only two independent threads the naive computation of the minimally necessary (thread-local) space to run the two threads leads to an exponential number of different schedules, which cannot be checked by a brute force search. As we will demonstrate in this paper, a deeper analysis shows that for independent processes (without communication, with the exception of the start and end), this minimum can be computed with an offline-algorithm in time $O((N+n)\log N)$ where N is the number of processes and n the size of the input (Theorem 5.6). The prerequisite for the algorithm is that the complete space trace of every single sub-process is already given, insofar the optimization can be classified as offline. Our abstract model can be applied if all processes have a common start and end time.

This simplicity of our model invites applications of the space-optimization algorithm also for

- industrial processes (jobs) where the number of machines can be optimized since it is similar to required space (resource-restricted scheduling). It can be used in problem settings similar to job-shop-scheduling problems [3], where the number of machines has to be minimized and where the time is not relevant (see e.g. [4]).

*supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1.

J. Niehren & D. Sabel (eds.): Rewriting Techniques
for Program Transformations and Evaluation (WPTE 2018)
EPTCS 289, 2019, pp. 53–67, doi:10.4204/EPTCS.289.4

- (independent) concurrent threads, independent of a programming language.

Our model is also extended to synchronization constraints in the form of a Boolean combination of conditions on simultaneous and/or relative time points of two threads. The results for the space optimization for synchronization-free processes can be transferred to processes with synchronizations and permits polynomial algorithms for a fixed number of synchronization constructs (see Theorem 6.2) and therefore allows further analyses of space in more concrete scenarios. In general, i.e. for arbitrary Boolean constraints, finding the minimum is NP-complete (Theorem 6.4).

The concrete programming language model that we investigate is the functional process calculus CHF, a variant of Concurrent Haskell, which permits pure and declarative functional modelling in combination with sequential (monadic) execution of processes with synchronization and which employs lazy evaluation [2, 7, 8]. Related work on space improvements in deterministic call-by-need functional languages is [5, 6, 10].

An application of results and algorithms for the space-minimization task in special cases is on the one hand to identify program transformation as space improvements (in CHF) and on the other hand to accelerate an automated search for potential counterexamples to conjectures of space-improvements. Space optimization of parallel processes can sometimes be also applied to CHF-programs. For example for processes that are deterministically parallel, i.e. there is no sharing between processes, no free variables and the computation terminates. In these special cases the notion of space improvement is the same as space optimization.

The **structure of the paper** is first to informally explain the functional process calculus *CHF***GC* and a definition of a space improvement in Section 2. A process-model and the interleaving is defined in Section 3. Then the computation of a standard form as a preparation of space optimization is given in Section 4. The optimization algorithm SpOptN is defined in Section 5, where also the correctness and complexity are determined in Theorem 5.6. Extensions for synchronization constructs are in Section 6. Section 7 illustrates a relation to other scheduling methods and reports on an implementation and use of the algorithm. The paper concludes with Section 8.

2 The Process Calculus *CHF* and Space Improvements

In this section we present sufficient information to understand the role of the space optimization method of parallel processes in the next sections for CHF as our example programming language. Therefore we first give an informal presentation of the concurrent program calculus *CHF* that combines distributed processes, synchronization, and shared memory with a purely functional expression language. We will also informally explain how our space optimization can contribute to the space behavior of program transformations (so-called space improvements) in the calculus *CHF*.

2.1 The Process Calculus *CHF*

CHF models a core language of Concurrent Haskell extended by futures, where the exact syntax, contexts, structural congruence rules, and reductions rules can be found for example in [8, 9].
A CHF-program as well as the program state after some reductions can always be represented by

$$x \overset{\mathsf{main}}{\Longleftarrow} e \mid x_1 \Leftarrow e_1 \mid \ldots \mid x_n \Leftarrow e_n \mid y_1 \,\mathbf{m}\, e'_1 \mid \ldots \mid y_m \,\mathbf{m}\, e'_m \mid z_1 = e''_1 \mid \ldots \mid z_k = e''_k$$

A thread $x_i \Leftarrow e_i$ is a sequentially executed process, where e_i is the thread-program,which finally binds its return value to x_i. The main thread $x \overset{\mathsf{main}}{\Longleftarrow} \ldots$ is a thread, with the special task to signal whether the

whole computation is finished. $y_i \,\mathbf{m}\, e_i'$ is a storage device that behaves as a one-place buffer, and $z_i = e_i''$ are shared memory cells containing the expression e_i''. The expressions e_i, e_i' and e_i'' are CHF-expressions, i.e. they are monadic expressions (sequential and side-effecting) which may contain pure expressions as in Haskell as subexpressions. The difference between $x_i \Leftarrow e_i$ and $x_i = e_i$ is that $x_i \Leftarrow e_i$ will execute, whereas $x_i = e_i$ is like a pointer for sharing the expression e_i.

The execution is defined through a standard reduction sequence on the syntactic description of the program (the state), which is a non-deterministic small-step reduction, where the non-determinism comes only from the competing processes. Every thread $x_i \Leftarrow e_i$ can be seen as a process that performs (controlled by the standard reduction) the computation defined by expression e_i. The parallel combination of the threads performs a distributed evaluation, where also new threads may be started.

As an example of a CHF-program consider the following definition of a program, where we use the do-notation as in Haskell with the same meaning in CHF.

$$x \overset{\mathsf{main}}{\Longleftarrow} \quad \texttt{do} \quad \begin{array}{l} z_1 \leftarrow (\texttt{future}\ e_1) \\ z_2 \leftarrow (\texttt{future}\ e_2) \\ \texttt{seq}\ (z_1 + z_2)\ (\texttt{return}(z_1, z_2)) \end{array}$$

After two reductions of the main thread, the state is

$$\begin{array}{l} x \overset{\mathsf{main}}{\Longleftarrow} \quad \texttt{do} \quad (\texttt{seq}\ (z_1 + z_2)\ (\texttt{return}(z_1, z_2))) \\ \mid z_1 \Leftarrow e_1 \\ \mid z_2 \Leftarrow e_2 \end{array}$$

which consists of three threads. The main thread now has to wait for the delivery of the values for z_1, z_2, which will be the result after the threads for z_1, z_2 terminate their computation and return something.

If the expressions e_1, e_2 use common variables, for example if e_2 demands the value of z_1, then the processes are not independent, and the sequence of executions is restricted. There may even be deadlocks, if e_1 requires z_2 as a value, and e_2 requires z_1 as a value.

In the case that the expressions e_1, e_2 do not use common variables (even not indirectly), the processes can be evaluated independently, which means that every interleaving of the executions of e_1, e_2 is possible. This independent case will be considered more deeply in later sections, since it permits nice space optimizations, and an example for easy detection of space improvements.

In addition to the program executions, it is crucial to recognize (binding-)garbage and remove it, since we are interested in space improving transformations. It is shown in [11, 12] that garbage collection and the modification of the standard reduction (i.e. program execution) leaves all interesting properties (equivalence of expressions, correctness of transformations) invariant, and thus this is a correct and space-optimizing transformation.

2.2 Space Measure, Equivalence of Programs and Space Improvements

An example for a space measure is the generalization of the space measure of [10, 11], which does not count variables (see Fig. 1). The reason for the specifics is that this size measure is compatible with the variants of abstract machines for CHF as explained in [10, 11].

Definition 2.1. *The space measure $sps(Red)$ of a successful standard reduction Red of a program P is the maximum of all sizes $\texttt{size}(P_i)$ during the whole standard reduction sequence, $Red = P \xrightarrow{sr} P_1 \xrightarrow{sr} \ldots \xrightarrow{sr} P_n$, where we assume that the P_i are always garbage-reduced.*

The space measure of a CHF-program P is defined as $spmin(P) = min\{sps(Red) \mid Red$ is a successful standard reduction of $P\}$.

$$
\begin{array}{lll}
\texttt{size}(x) & = 0 & \\
\texttt{size}(e_1\ e_2) & = 1+\texttt{size}(e_1)+\texttt{size}(e_2) & \\
\texttt{size}(\lambda x.e) & = 1+\texttt{size}(e) & \\
\texttt{size}(\texttt{case}\ e\ \texttt{of}\ \texttt{alt}_1 \ldots \texttt{alt}_n) & = 1+\texttt{size}(e)\ \ +\sum_{i=1}^{n}\texttt{size}(\texttt{alt}_i) & \\
\texttt{size}((c\ x_1 \ldots x_n)\ \texttt{->}\ e) & = 1+\texttt{size}(e) & \\
\texttt{size}(f\ e_1 \ldots e_n) & = 1+\sum\texttt{size}(e_i) & \text{for constructors and operators } f \\
 & & \quad \text{such as } \texttt{future},\texttt{return},\ldots \\
\texttt{size}(\texttt{letrec}\ x_1=e_1,\ldots,x_n=e_n\ \texttt{in}\ s) & = \texttt{size}(e)+\sum\texttt{size}(e_i) & \\
\texttt{size}(P_1 \mid P_2) & = \texttt{size}(P_1)+\texttt{size}(P_2) & \\
\texttt{size}(x\ op\ e) & = 1+\texttt{size}(e) & \text{for } op \in \{=,\Leftarrow,\mathbf{m}\} \\
\texttt{size}(x\,\mathbf{m}-) & = 1 & \\
\texttt{size}(\nu x.P) & = 1+\texttt{size}(P) &
\end{array}
$$

Figure 1: Definition of `size` of expressions

As a concrete example, the size of the program $(x \overset{\mathsf{main}}{\Longleftarrow} \texttt{return}\ y \mid x\,\mathbf{m}\,1 \mid y = \texttt{Cons}\ x\ \texttt{Nil})$ is $2+2+2=6$.

The reason for not counting the sizes directly before a garbage collection is that the calculus and abstract machines may create bindings that may be garbage and would thus be immediately garbage collected after the reduction step. Taking this garbage into account would distort the reasoning about measurement in particular if these bindings have a large size (more information about this can be found in [10]). This principle of measuring space in a small-step calculus is also used in [6].

In the following $P\!\downarrow$ means that P has a successful standard reduction and $P\!\uparrow$ is its negation; $P_1 \sim_c P_2$ means that P_1, P_2 are contextually equivalent in *CHF*.

Definition 2.2. *A program transformation* $\xrightarrow{PT}$ *is a* space-improvement *if for all contextual equivalent processes* P_1, P_2*:* $P_2 \xrightarrow{PT} P_1$ *implies that* P_1 *space-improves* P_2*, i.e.* $spmin(P_1) \leq spmin(P_2)$.

In this paper we focus on a special situation, where the program P consists of several threads that, after they are started, run completely independent, without using common data structures, and then communicate and halt. In order to test or prove $P \longrightarrow P'$ to be an space-improvement, it is crucial to determine the optimal space usage of P and compare it with the optimal space usage of P'. The computation of the optimal space usage of P requires (among others) to find the space-optimal interleaving of the phase between starting the n threads until all threads finally stop and communicate.

For example, in the program $u \overset{\mathsf{main}}{\Longleftarrow} \ldots \mid \ldots \mid x \Leftarrow e_1 \mid y \Leftarrow e_2$, we consider $x \Leftarrow e_1$ and $x \Leftarrow e_2$ as the two subprocesses p_1, p_2, which can be measured separately.

For the case of independent processes, we present an algorithm for computing an optimal interleaving and the space minimum in the case of parallel evaluation possibilities (if the executions are already given), where the algorithm runs in $O(n \log n)$ time. We also analyze the impact on the runtime in the case of dependencies between processes, where synchronization points between processes are defined explicitly.

3 Abstract Model of Independent Processes and Space

The assumptions underlying the abstraction is that CHF-processes use a common memory for their local data structures, but they cannot see each others memory entries. The CHF-processes may independently

start or stop or pause at certain time points. We also assume that synchronization and communication may occur at certain time points as interaction between CHF-processes.
Every CHF-process is abstractly modeled by its trace of space usage, given as a list of integers. In addition we later add constraints expressing simultaneous occurrences of time points of different CHF-processes as well as start-points and end-points of CHF-processes.

In the following we use the notation $[a_1,\ldots,a_n]$ for a list of the elements $a_1,\ldots,a_n$. We also use $(a:l)$ for adding a first element a to list l, l_1++l_2 for appending the lists l_1 and l_2, $tail(l)$ for the tail of the list l, and $[f(x) \mid x \in L]$ for a list L denotes the list of $f(x)$ in the same sequence as that of L (i.e. it is a list comprehension).

In the following we abstract CHF-processes by a list of non-negative integers. For simplicity we call this list a *process* in the rest of the paper. A (parallel) interleaving is constructed such that from one state to the next one, each process proceeds by at most one step and at least one process proceeds.

Definition 3.1. *A* process *is a nonempty, finite list of non-negative integers. For $n > 0$ let $P_1,\ldots,P_n$ be processes where m_i is the length of p_i, and let $p_{i,j}$ for $j = 1,\ldots,m_i$ be the elements. Then an* interleaving *of $P_1,\ldots,P_n$ is a list $[q_1,\ldots,q_h]$ of n-tuples q_j constructed using the following (non-deterministic) algorithm:*

1. *Initially, let q be the empty list.*
2. *If all processes $P_1,\ldots,P_n$ are empty, then return q.*
3. *Set $q := q$++$[(p_{1,1},\ldots,p_{n,1})]$, i.e., the tuple of all first elements is added at the end of q.*
   Let $(b_1,\ldots,b_n)$ be a (nondeterministically chosen) tuple of Booleans, such that there is at least one k such that b_k is `True` *and P_k not empty.*
   For all $i = 1,\ldots,n$: set $P_i = tail(P_i)$ if b_i and p_i is not empty; otherwise do not change P_i.
   Continue with item 2.

Definition 3.2. *Let $P_1,\ldots,P_n$ be processes. The* space usage *$sps(S)$ of an interleaving S of $P_1,\ldots,P_n$ is the maximum of the sums of the elements in the tuples in S, i.e. $sps(S) = \max\{\sum_{i=1}^{n} a_i \mid (a_1,\ldots,a_n) \in S\}$. The* required space *$spmin(P_1,\ldots,P_n)$ for n processes $P_1,\ldots,P_n$ is the minimum of the space usages of all interleavings of $P_1,\ldots,P_n$, i.e. $\min\{sps(S) \mid S$ is an interleaving of $P_1,\ldots,P_n\}$.*
A peak *of P_i is a maximal element of P_i, and a* valley *is a smallest element in P_i. A* local peak *of P_i is an maximal element in P_i which is not smaller than its neighbors. A* local valley *of P_i is a minimal element in P_i which is not greater than its neighbors.*

Example 3.3. *For two processes $[1,7,3],[2,10,4]$ the spmin-value is 11, by first running the second one and then running the first. I.e. such a (space-optimal) interleaving is $[(1,2),(1,10),(1,4),(7,4),(3,4)]$. The interleaving that results from an "eager" scheduling is $[(1,2),(7,10),(3,4)]$, with sps-value 17, and hence is not space-optimal.*

4 Standard Form of Processes

We will argue that an iterated reduction of single processes by the following 5 patterns permits to compute *spmin* from smaller processes. This is a first step like a standardization of processes for the purpose of *spmin*-computation, and is a preparing step for the optimization algorithm SPOPTN in Definition 5.4.

Definition 4.1. *The trivial pattern M_0 is $a_i = a_{i+1}$. There are two further, nontrivial patterns: The first pattern M_1 is $a_i \le a_{i+1} \le a_{i+2}$ and the second pattern M_2 is $a_i \ge a_{i+1} \ge a_{i+2}$.*
A pattern matches a process *$[a_1,\ldots,a_k]$ at index i, if for index i the conditions are satisfied.*

A single pattern application is as follows: If the patterns M_0, M_1 or M_2 matches a process for some index i, then a_{i+1} is removed.

Proposition 4.2. *Let $P_1,\ldots,P_n$ be n processes and let $P_1',\ldots,P_n'$ be the processes after removal of subsequent equal entries, i.e. using M_0. Then $spmin(P_1,\ldots,P_n) = spmin(P_1',\ldots,P_n')$.*

Proof. This is obvious by rearranging the schedules, leading to different interleavings, which have the same *spmin*-value. □

Proposition 4.3. *Let $P_1,\ldots,P_n$ be n processes. Let $P_1',\ldots,P_n'$ be the processes after several application of the pattern-reduction process using M_1 and M_2. Then $spmin(P_1,\ldots,P_n) = spmin(P_1',\ldots,P_n')$.*

Proof. It is sufficient to assume that exactly one change due to a pattern match is performed. It is also sufficient to assume that the pattern is M_1 and that it applies in P_1. We can also look only at a subpart of an interleaving to have easier to grasp indices. For argumentation purposes, we choose the correspondence between the interleavings $(P_1,P_2,\ldots,P_n)$ and $(P_1',P_2,\ldots,P_n)$ as follows.
Let $[p_{1,1},p_{1,2},p_{1,3}]$ with $p_{1,1}\le p_{1,2}\le p_{1,3}$ be the subprocess of P_1 that is replaced by $[p_{1,1},p_{1,3}]$. Consider the part $(p_{1,1},\ldots,p_{n,1}) : [(p_{1,2},p_{2,2},\ldots,p_{n,2}) \mid (p_{2,2},\ldots,p_{n,2}) \in B]$++$[(p_{1,3},\ldots,p_{n,3})]$ of the interleaving, where B is a sequence of $n-1$-tuples. Then the modified interleaving for $(P_1',P_2,\ldots,P_n)$ can be defined as: $(p_{1,1},\ldots,p_{n,1}) : [(p_{1,1},p_{2,2},\ldots,p_{n,2}) \mid (p_{2,2},\ldots,p_{n,2}) \in B]$++$[(p_{1,3},\ldots,p_{n,3})]$
and since for every interleaving of $(P_1,P_2,\ldots,P_n)$ we obtain an interleaving of $(P_1',P_2,\ldots,P_n)$ with a *sps* that is smaller or equal, and since *spmin* is defined as a minimum, we obtain $spmin(P_1,P_2,\ldots,P_n) \ge spmin(P_1',P_2,\ldots,P_n)$.
For the other direction, consider the part $[(p_{1,1},\ldots,p_{n,1}),(p_{1,3},p_{2,2},\ldots,p_{n,2})]$ of an interleaving of the processes $P_1',P_2,\ldots,P_n$. Then *spmin* of the part $[(p_{1,1},\ldots,p_{n,1}),(p_{1,2},p_{2,2},\ldots,p_{n,2}),(p_{1,3},p_{2,2},\ldots,p_{n,2})]$ of the interleaving of $P_1,\ldots,P_n$ is the same as before, thus $spmin(P_1,\ldots,P_n) \le spmin(P_1',P_2,\ldots,P_n)$.
The two inequations imply $spmin(P_1,\ldots,P_n) = spmin(P_1',P_2,\ldots,P_n)$. □

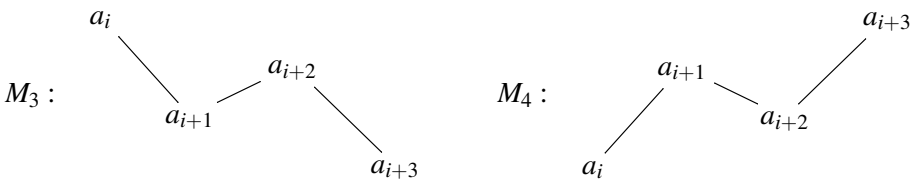
Definition 4.4. *If in a process P every strict increase is followed by a strict decrease and every strict decrease is followed by a strict increase, then the process P is called a* zig-zag process.

By exhaustive application we can assume that the pattern M_0, M_1 and M_2 above are not applicable to processes which means that the processes can be assumed to be zig-zag.

Now we show that there are more complex patterns that can also be used to reduce the processes before computing *spmin*. The following patterns M_3, M_4 are like stepping downstairs and upstairs, respectively.

Definition 4.5. *The patterns M_3, M_4 are defined as follows:*

- *M_3 consists of $a_i, a_{i+1}, a_{i+2}, a_{i+3}$, with $a_i > a_{i+1}$, $a_{i+1} < a_{i+2}$, $a_{i+2} > a_{i+3}$ and $a_i \ge a_{i+2}, a_{i+1} \ge a_{i+3}$.*
- *M_4 consists of $a_i, a_{i+1}, a_{i+2}, a_{i+3}$, with $a_i < a_{i+1}$, $a_{i+1} > a_{i+2}$, $a_{i+2} < a_{i+3}$ and $a_i \le a_{i+2}, a_{i+1} \le a_{i+3}$.*



If for some i: M_3 or M_4 matches, then eliminate a_{i+1}, a_{i+2}.

We show that the complex patterns can be used to restrict the search for an optimum to special processes:

Lemma 4.6. *Let $P_1,\ldots,P_n$ be processes. If one of the patterns M_3, M_4 matches one of the processes, then it is sufficient to check the shortened $P'_1,\ldots,P'_n$ for the space-minimum.*

Proof. It is sufficient to assume that exactly one change due to a pattern match is performed. It is sufficient to assume that the pattern is M_3 and that it applies in P_1. We can also look only at a subpart of an interleaving to have easier to grasp indices. For argumentation purposes, we choose the correspondence between the interleavings $(P_1,P_2,\ldots,P_n)$ and $(P'_1,P_2,\ldots,P_n)$ as follows.
Let $[p_{1,1},p_{1,2},p_{1,3},p_{1,4}]$ with $p_{1,1} > p_{1,2}$, $p_{1,2} < p_{1,3}$, $p_{1,3} > p_{1,4}$, $p_{1,1} \geq p_{1,3}$ and $p_{1,2} \geq p_{1,4}$ be the subprocess of P_1 that is replaced by $[p_{1,1},p_{1,4}]$. Consider the following part of the interleaving, where B_2, B_3 are sequences of $n-1$-tuples:

$(p_{1,1},\ldots,p_{n,1}) : [(p_{1,2},p_{2,2},\ldots,p_{n,2}) \mid (p_{2,2}\ldots,p_{n,2}) \in B_2]$++$[(p_{1,3},p_{2,3},\ldots,p_{n,3}) \mid (p_{2,3}\ldots,p_{n,3}) \in B_3]$ ++$[(p_{1,4},\ldots,p_{n,4})]$

The modified interleaving for $(P'_1,P_2,\ldots,P_n)$ can be defined as $[(p_{1,1},\ldots,p_{n,1})]$++$[(p_{1,4},q_{2,4},\ldots,q_{n,4}) \mid (q_{2,4},\ldots,q_{n,4}) \in B_2$++$B_3$++$[p_{2,4},\ldots,p_{n,4}]]$ and since for every interleaving of $(P_1,P_2,\ldots,P_n)$ we obtain an interleaving of $(P'_1,P_2,\ldots,P_n)$ with a *sps* that is smaller or equal, and since *spmin* is defined as a minimum, we obtain $spmin(P_1,P_2,\ldots,P_n) \geq spmin(P'_1,P_2,\ldots,P_n)$.
Now consider the part $[(p_{1,1},\ldots,p_{n,1}),(p_{1,4},\ldots,p_{n,4})]$ of an interleaving of $P'_1,P_2,\ldots,P_n$. Then $spmin(.)$ of the part $[(p_{1,1},p_{2,1},\ldots,p_{n,1}),(p_{1,2},p_{2,1}\ldots,p_{n,1}),(p_{1,3},p_{2,1},\ldots,p_{n,1}),(p_{1,4},\ldots,p_{n,4})]$ of the interleaving of $P_1,\ldots,P_n$ is the same as before, thus $spmin(P_1,\ldots,P_n) \leq spmin(P'_1,P_2,\ldots,P_n)$.

The two inequations imply $spmin(P_1,\ldots,P_n) = spmin(P'_1,P_2,\ldots,P_n)$. □

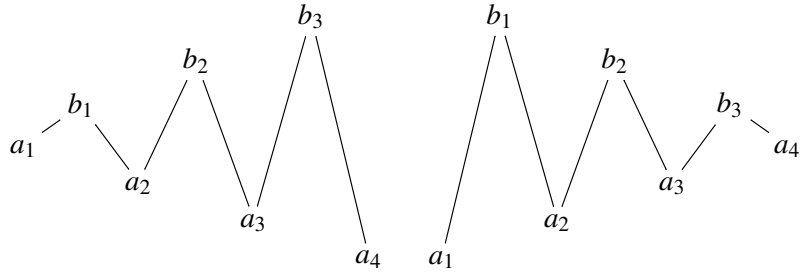
Definition 4.7. *A process $[a_1,b_1,a_2,b_2,\ldots,a_n]$ (or $[b_0,a_1,b_1,a_2,\ldots,a_n]$, or $[a_1,b_1,a_2,b_2,\ldots,a_n,b_n]$, or $[b_0,a_1,b_1,a_2,b_2,\ldots,a_n,b_n]$, resp.) is a monotonic increasing zig-zag (mizz), iff $a_i < b_j$ for all i,j, and $a_1,a_2,\ldots,a_n$ is strictly monotonic decreasing, and $b_1,b_2,\ldots,b_{n-1}$ (and $b_0,b_1,b_2,\ldots,b_{n-1}$ and $b_0,b_1,b_2,\ldots,b_{n-1},b_n$, resp.) is strictly monotonic increasing.*

A process $[a_1,b_1,\ldots,a_n]$ is a monotonic-decreasing zig-zag (mdzz), iff $a_i < b_j$ holds for all i,j, and $a_1,a_2,\ldots a_n$ is strictly monotonic increasing, and $b_1,b_2,\ldots,b_{n-1}$ (or $b_0,b_1,b_2,\ldots,b_{n-1}$, resp.) is strictly monotonic decreasing.

A process is midzz*, if it is a mizz followed by a mdzz. More rigorously, there are essentially two cases, where we omit the cases with end-peaks and/or start-peaks.*

1. *the mizz $[a_1,b_1,a_2,b_2,\ldots,a_n]$ and the mdzz $[a'_1,b'_1,\ldots,a'_n]$, where $a_n = a'_1$ are combined to $[a_1,b_1,a_2,b_2,\ldots,a_n,b'_1,\ldots,a'_n]$,*
2. *the mizz $[a_1,b_1,a_2,b_2,\ldots,a_n,b_n]$ and the mdzz $[b'_0,a'_1,b'_1,\ldots,a'_n]$, where $b_n = b'_0$ are combined to $[a_1,b_1,a_2,b_2,\ldots,a_n,b_n,a'_1,b'_1,\ldots,a'_n]$.*

Typical graphical representations of mizz- and mdzz-sequences are:



If the goal is to compute the optimal space, then there are several reduction operations on processes that ease the computation and help us to concentrate on the hard case. First we show that one-element

processes can be excluded, and second that processes with start- or end-peaks can be reduced by omitting elements. Then we show that through the use of the 5 patterns $M_0,\ldots,M_4$ for reductions we can concentrate on special forms of zig-zag-processes, so-called midzz.

Proposition 4.8. *If $P_1=[a_1]$ and $P_2,\ldots,P_n$ are processes then $spmin(P_1,\ldots,P_n)=a_1+spmin(P_2,\ldots,P_n)$.*

Proof. a_1 is the first element of every tuple in any interleaving of $P_1,\ldots,P_n$, hence the claim is valid. □

Proposition 4.9. *Let $P_i=[p_{i,1},\ldots,p_{i,n_i}]$ for $i=1,\ldots,n$ be processes. If $p_{1,1}$ is a start-peak of P_1, then let $P_1'=[p_{1,2},\ldots,p_{1,n_1}]$. Then $spmin(P_1,\ldots,P_n)=\max(\sum_i p_{i,1}, spmin(P_1',P_2,\ldots,P_n))$. The same holds symmetrically if P_1 ends with a local peak.*

Proof. Let $q=[(p_{1,1},q_{1,2},\ldots,q_{1,n}),\ldots,(p_{1,1},q_{h,2},\ldots,q_{h,n})]$++ $[(p_{1,2},q_{h+1,2},\ldots,q_{h+1,n})]$++$R$ be an interleaving for $P_1,\ldots,P_n$ and some h. If $h\neq 1$, this can be changed to $[(p_{1,1},q_{1,2},\ldots,q_{1,n}),(p_{1,2},q_{2,2},\ldots,q_{2,n}),\ldots,(p_{1,2},q_{h,2},\ldots,q_{h,n})]$ ++ $[(p_{1,2},q_{h+1,2},\ldots,q_{h+1,n})]$++$R$ without increasing the necessary space. Hence $spmin(P_1,\ldots,P_n)\geq\max(\sum_i p_{i,1},\ spmin(P_1',P_2,\ldots,P_n))$.

On the other hand, if we have a space-optimal schedule of $P_1',P_2,\ldots,P_n$, then we can extend this by starting with $(p_{1,1},\ldots,p_{n,1})$ and obtain $spmin(P_1,\ldots,P_n)\leq\max(\sum_i p_{i,1},spmin(P_1',P_2,\ldots,P_n))$.

Hence $spmin(P_1,\ldots,P_n)=\max(\sum_i p_{i,1},spmin(P_1',P_2,\ldots,P_n))$. □

Lemma 4.10. *We can assume that processes $P_1,\ldots,P_n$ are all of length at least 3 for computing the optimal space.*

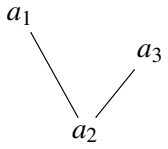
Proof. Proposition 4.8 permits to assume that the length is at least 2. Proposition 4.9 allows to assume that there is no start- nor an end-peak. Hence we can assume that processes are of length at least 3. □

Lemma 4.11. *Let P be a process that starts and ends with local valleys. Then the application of the patterns $M_0,\ldots,M_4$ with subsequent reduction always produces a process that also starts and ends with local valleys.*

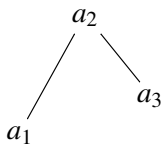
Proof. The reduction either removes according to pattern M_0 or it removes inner entries of the lists. □

Proposition 4.12. *A process such that none of the patterns M_0, M_1, M_2, M_3, M_4 matches and which does not start or end with a local peak is a midzz.*

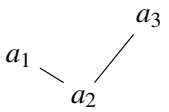
Proof. We consider all four different cases how small sequences may proceed, if no pattern applies.

1. 

Case $a_1>a_2$, $a_2<a_3$ and $a_3<a_1$. Then $a_4<a_3$. The relation $a_4\leq a_2$ is not possible, since then pattern M_3 matches. Hence $a_3>a_4>a_2$. Then a_1,a_2,a_3,a_4 is a tail of a mdzz.
The case $a_1>a_2$, $a_2<a_3$ and $a_3=a_1$ leads to the same relations $a_3>a_4>a_2$. Then a_2,a_3,a_4 is a tail of a mdzz.

2. 

Case $a_1<a_2$, $a_2>a_3$ and $a_3>a_1$. Then $a_4>a_3$. The relation $a_4\geq a_2$ is not possible, since then pattern M_4 matches. Hence $a_3<a_4<a_2$. Then a_1,a_2,a_3,a_4 is a mdzz.
The case $a_1<a_2$, $a_2>a_3$ and $a_3=a_1$ leads to the same relations $a_3<a_4<a_2$. Then using case 1 for the the next element a_5, the sequence a_3,a_4,a_5 is a tail of a mdzz.

3. 

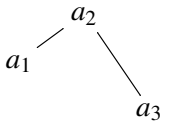
a_4?

Case $a_1 > a_2, a_2 < a_3$ and $a_3 > a_1$. Then $a_3 > a_4$ and there are three cases:

(i) If $a_4 = a_2$ then the sequence starting from a_3 is a mdzz.

(ii) If $a_4 > a_2$ then case 2 is applicable and the sequence starting from a_2 is a mdzz.

(iii) If $a_4 < a_2$ then the sequence starting with a_1 proceeds as mizz. It may later turn into a mdzz.

4. 

a_4?

Case $a_1 < a_2, a_2 > a_3$ and $a_3 < a_1$. Then $a_3 < a_4$ and there are three cases:

(i) If $a_4 = a_2$ then the sequence starting from a_3 is a mdzz.

(ii) If $a_4 < a_2$ then case 1 is applicable and the sequence starting from a_2 is a mdzz.

(iii) If $a_4 > a_2$ then the sequence starting with a_1 proceeds as mizz. It may later turn into a mdzz.

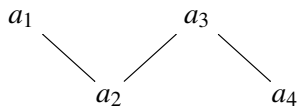
Now we put the parts together and conclude that the sequence must be a midzz. □

Note that the definition of midzz permits the simplified case that the process is a mizz or mdzz.

Definition 4.13. *A process is called* standardized *if it is a midzz of length at least 3, and does not start nor end with a local peak.*

Lemma 4.14. *Let P be a midzz-process, where no pattern M_0, M_1, M_2, M_3,M_4 applies, and which is of length at least* 3*, and does not start nor end with a local peak: Then a midzz-process has one or two global peaks, it has one or two global valleys, but not two global peaks and two global valleys at the same time.*

Proof. The considerations and cases in the proof of Proposition 4.12 already exhibit the possible cases. Since the patterns M_3,M_4 do not apply, there cannot be three global peaks nor three global valleys. If there are two global peaks and two global valleys, then the picture is



and we can apply pattern M_3, which is forbidden by the assumptions. Similarly for the case where a_1 is a global valley. □

Hence, a standardized process in midzz-form has three different possibilities for the global peaks and valleys: (i) there is a unique global peak and a unique global valley; (ii) there is a unique global peak and two global valleys; (iii) there are two global peaks and a unique global valley.

5 Optimizing Many Independent Processes

Let us assume in this section that there are N processes $P_1, \ldots, P_N$ of total size n.

Algorithm 5.1 (Standardization)**.** *For an input of N processes $P_1, \ldots, P_N$:*

1. *For every process P_i in turn: Scan P_i by iterating j from* 0 *as follows:*
   If the patterns $M_0, \ldots, M_4$ apply at index j then reduce accordingly and restart the scan at position $j-3$, otherwise go on with index $j+1$.

2. *Let K_0 be the sum of all first elements of $P_1,\ldots,P_N$. Let $P'_1,\ldots,P'_N$ be obtained from $P_1,\ldots,P_N$ by removing all start-peaks only from processes of length at least 2.*
3. *Let K_ω be the sum of all last elements of $P'_1,\ldots,P'_N$. Let $P''_1,\ldots,P''_N$ be obtained from $P'_1,\ldots,P'_N$ by removing all end-peaks only from processes of length at least 2.*
4. *Let A be the sum of all elements of one-element processes, and let $P'''_1,\ldots,P'''_{N'}$ be $P''_1,\ldots,P''_N$ after removing all one-element processes.*
5. *If M''' is $spmin(P'''_1,\ldots,P'''_{N'})$, then $spmin(P_1,\ldots,P_N)$ is computed as* $\max(M'''+A,K_0,K_\omega)$.

Theorem 5.2. *Algorithm 5.1 for standardization reduces the computation of spmin for N processes $P_1,\ldots,P_N$ of size n to the computation of spmin for standardized processes in time $O(n)$.*

Proof. Algorithm 5.1 is correct by Propositions 4.8 and 4.9.
The required number of steps for pattern application is $O(n)$: Every successful application of a pattern strictly reduces the number of elements. The maximum number of steps back is 3, hence at most $4n$ total steps are necessary. Stepping back for 3 is correct, since a change at index k cannot affect pattern application for indices less than $k-3$. The overall complexity is $O(n)$ since scans are sufficient to perform all the required steps and computations in Algorithm 5.1. □

Algorithm 5.3. Algorithm for Left-Scan of N processes. *We describe an algorithm for standardized processes which performs a left-scan until a global valley is reached and returns the required space for the left part.*
The following index $I_{i,ends}$ in process P_i for $i=1,\ldots,N$ is fixed: It is the index in P_i of the global valley, if it is unique, and of the rightmost global valley if there are two global valleys.

1. *Build up a search tree T that contains pairs $((p_{i,2}-p_{i,1}),i)$ for each process $P_i=[p_{i,1},\ldots,p_{i,n_i}]$, where the first component is the search key.*
2. *Set $S=M=\sum_i p_{i,1}$. Also for each process P_i there are indices I_i indicating the current valley positions of the process, initially set $I_i=1$ for each process.*
3. *If T is empty then return M and terminate.*
4. *Remove the minimal element $V=(d,i)$ from T.*
   If $I_i+2\le I_{i,ends}$, then set $M=\max(M,S+d)$, $S=S+(p_{i,3}-p_{i,1})$, insert $(p_{i,4}-p_{i,3},i)$ into T (only if P_i contains at least 4 elements), set $I_i=I_i+2$ and remove the first two elements from P_i. Note that P_i is not considered anymore in the future if $I_i+2>I_{i,ends}$ or if there is no further peak in P_i after I_i.
   Goto (3).

The right-to-left algorithm is the symmetric version and yields also the required space for the right part.

Algorithm 5.4. SPOPTN **Computation of** *spmin* **for N processes**

1. *Let M_{start} be the sum of all start elements, and M_{end} be the sum of all end elements of the given processes $P_1,\ldots,P_N$.*
   Also let M_{one} be the sum of all elements of one-element-processes.
2. *Transform the set of processes into standard form.*
3. *Compute M_{left} using the left-to-right scan and M_{right} using the right-to-left scan.*
4. *Return the maximum of $(M_{left}+M_{one})$, $(M_{right}+M_{one})$, M_{start} and M_{end}.*

Theorem 5.5. *Algorithm 5.4,* SPOPTN*, computes spmin of N processes.*

Proof. Let $P_1, \ldots, P_N$ be N processes. To achieve the standard forms Algorithm 5.1 is applied. First we argue that for those processes the required space is at least the computed space by the left-to-right scan.

Consider a state $(i_1, \ldots, i_n)$ during the construction of a space-optimal interleaving using space M, where every i_j is not after the index of the smallest valley, which means $i_j \leq I_{j,ends}$. An invariant of the state is that $p_{i_1} + \ldots + p_{i_n} \leq M$. We also assume as an invariant that the current state belongs to an optimal interleaving. If some i_j is the position of a local peak, then the optimal interleaving can be changed to $i_j + 1$ such that the next tuple is $(i_1, \ldots, i_j + 1, \ldots, i_n)$. Repeating this argument, we can assume that $(i_1, \ldots, i_n)$ contains only indices of local valleys. Now consider the set S of positions j in the tuple, such that $i_j < I_{j,ends}$. For at least one such index the optimal interleaving must proceed. For the indices in S, the next index will be a local peak, so the best way is to look for the smallest peak p_{i_j+1} for $j \in S$. If the sum of the spaces exceeds M then we have a contradiction, since the interleaving must proceed somewhere. Hence M is at least $\min\{p_{i_j+1} + \sum_{h \neq j} p_{i_h} \mid j = 1, \ldots, n\}$. This argument also holds, if the indices i_j for $j \notin S$ are beyond $I_{j,ends}$, since the valley at $I_{j,ends}$ is smaller. For a better efficiency the algorithm calculates these sums implicitly by keeping track of the sum of the current valleys, i.e. $\sum_h p_{i_h}$. Then it uses a search tree containing the space differences between the corresponding local valley and the next peak to step forward, i.e. to calculate p_{i_j+1}. For the right-to-left scan the same arguments hold, symmetrically where by slight asymmetry, we only scan to the rightmost minimal valley for every process.

Thus we have two lower bounds M_{left} and M_{right} for the optimal interleaving.

The only missing argument is that we can combine those two values. For processes that have a unique global minimal valley, the combination is trivial. For the case of processes that have global minimal valleys, we glue together the left interleaving with the reversed right interleaving. This is an interleaving and it can be performed in space at most the maximum of M_{left} and M_{right}. Concluding, the algorithm computes *spmin* for the input processes. □

Theorem 5.6. *If there are N processes $P_1, \ldots, P_N$ of total size n, then the optimal space and an optimal schedule can be computed in time $O(N \log N + n \log N)$.*

Proof. The algorithm SPOPTN computes the optimal space and an optimal schedule (see Theorem 5.5). We estimate the required time: The time to produce a standardized problem is linear, which follows from Theorem 5.2. The left-to-right and the right-to-left scan can be performed in time $O(N \log N + n \log N)$: The search tree can be initially constructed in $O(N \log N)$. Since the search tree contains at most N elements during the whole calculation, we need $O(n \log N)$ steps for all lookups and insertions.

Note that the bit-size of the integers of the space-sizes is not relevant, since we only use addition, subtraction, and maximum-operations on these numbers. □

Example 5.7. *This example illustrates the computation (without the optimization using search trees) as follows:*
Let $P_1 = [10,1,12,5,7,1]$, $P_2 = [3,11,2,10,3]$ and $P_3 = [1,2,3,4,3,2,1]$.
Then we first can reduce the processes as follows: P_3 can be reduced by pattern M_1, M_2 to $P_3' = [1,4,1]$. P_2 is already a zig-zig process, therefore no pattern applies. P_1 starts with a local peak, hence we keep in mind 14 *as the sum of the first elements and replace P_1 by $P_1' = [1,12,5,7,1]$. The next step is to apply the pattern M_3, which reduce it to $P_1'' = [1,12,1]$. Thus the new problem is $P_1'' = [1,12,1]$, $P_2 = [3,11,2,10,3]$, $P_3' = [1,4,1]$.*
A short try shows that 15 *is the optimum. However, we want to demonstrate the algorithm:*

The left scan starts with Max $= 5$. *The peak in* P_3' *then enforces Max* $= 8$ *and* P_3' *is not considered anymore, since the left scan reached the final position in* P_3*, i.e. the rightmost global valley. The peak in* P_2' *then enforces Max* $= 13$ *and also* P_2 *is not considered anymore, since the final position is reached. Finally the peak in* P_1'' *enforces Max* $= 15$ *and the left scan terminates.*
The right scan starts with Max $= 5$. *Then the peak in* P_3' *enforces Max* $= 8$*, after this the peak in* P_2' *enforces Max* $= 12$ *and finally the peak in* P_1'' *enforces Max* $= 15$.
Hence in summary, also taking the local peak at the beginning of P_1 *into account, the result is* 15.

6 Processes with Synchronizations

We indicate how to extend our model to timing and synchronization restrictions. For example, in CHF writing into a filled MVar requires the process to wait until the MVar is empty. There are also race-conditions, for example if several processes try to write into an empty MVar, or several processes try to read the same MVar. These constraints are captured by the constraints below, where the race conditions can be modeled by disjunctions.

Definition 6.1. *There may be various forms of synchronization restrictions. We will only use the following forms of fundamental restrictions:*

1. *simul*(P_1,P_2,i_1,i_2)*: for processes* P_1,P_2 *the respective actions at indices* i_1,i_2 *must happen simultaneously.*
2. *starts*(P_1,P_2,i)*: process* P_1 *starts at index i of process* P_2
3. *ends*(P_1,P_2,i)*: process* P_1 *ends at index i of process* P_2.
4. *before*(P_1,P_2,i_1,i_2)*: for processes* P_1,P_2 *the action at index* i_1 *of* P_1 *happens simultaneously or before the action at* i_2 *of* P_2.

For a set R of restrictions only schedules are permitted that obey all restrictions. This set R is also called a set of basic *restrictions.*

We also permit Boolean formulas over such basic restrictions. In this case the permitted schedules must obey the complete formula.

Note that in CHF these restrictions correspond to synchronization conditions of: start of a future, waiting for an MVar to be in the right state. The simultaneous condition is not necessary for single reduction steps in CHF, but can be used for blocks of monadic commands.

We show that there is an algorithm for computing the optimal space and an optimal schedule that has an exponential complexity, where the exponent is $b \cdot N$ where b is the size of the Boolean formula and N is the number of processes.

Theorem 6.2. *Let there be N processes and a set B of Boolean restrictions where b is the size of B and the size of the input is n. Then there is an algorithm to compute the optimal space and an optimal schedule of worst case asymptotic complexity of* $O(poly(n) \cdot n^{O(b \cdot N)})$*, where poly is a polynomial.*

Proof. The algorithm is simply a brute force method of trying all possibilities: For every condition try all tuples of indices. The number of different tuples is at most n^N and for trying this for every basic restriction we get an upper bound of $n^{N \cdot b}$. Now we have to check whether the time constraints are valid, i.e. there are no cycles, which can be done in polynomial time. Now we can split the problem into at most $b+1$ intervals with interception of an index of a condition and apply for every interval the algorithm SPOPTN (see 5.4), which requires time sub-quadratic in n by Theorem 5.6. Thus we get an asymptotic time complexity as claimed. □

Corollary 6.3. *Let there be N processes and a set B of Boolean restrictions where b is size of B and the size of the input is n. Assume that the number N of processes and the size of B is fixed. Then there is a polynomial algorithm to compute the optimal space and an optimal schedule.*

In general, the optimization problem with synchronization restrictions is NP-complete:

Theorem 6.4. *In the general case of synchronization restrictions, the problem of finding the minimal space is NP-hard and hence NP-complete.*

Proof. We use the (perfect) partition problem, which is known to be NP-hard. An instance is a multi-set A of positive integers and the question is whether there is a partition of A into two sub-multi-sets A_1, A_2, such that $\sum A_1 = \sum A_2$.

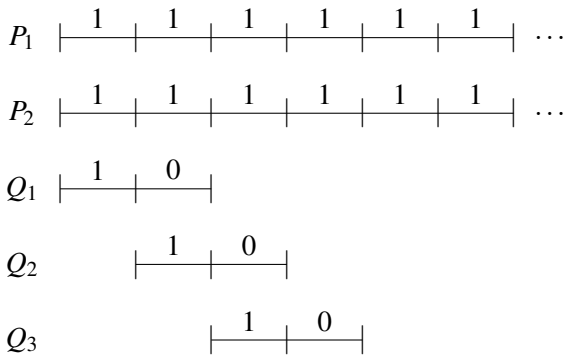
This can be encoded as the question for the minimal space for a scheduling: Let $P_i = [0, a_i, 0, 0]$ for $A = \{a_1, \ldots, a_n\}$ and $P_0 = [0,0,0,0]$, where the indices are $1,2,3,4$. The condition is a conjunction of the following disjunctions: $(P_0, P_i, 2, 3) \vee (P_0, P_i, 3, 2)$. The optimal space is reached for a schedule, where indices $1,4$ are zero and where at index 2 and 3, there is a perfect partition of A. □

Example 6.5. *We illustrate how an abstract version of the producer-consumer problem can be modeled using interleavings and synchronization restrictions. The idea is that the consumer process P_1 produces a list/stream that is consumed by the process P_2. The single elements are also modeled as processes. Our modelling will be such that the optimal space modelling coincides with the intuition that the space usage of the intermediate list is minimal if there is an eager consumption of the produced list elements.*

We represent the problem as follows. There are two processes P_1, P_2, the producer and the consumer, which consist of n times the symbol 1. *There are also n processes $Q_1, \ldots, Q_n$ that only consist of two elements: a* 1 *followed by a* 0*, where the processes represent the unconsumed parts of the exchanged list. We represent the possible executions by synchronization restrictions:*

- *Q_i is started by P_1 at time point i: $starts(Q_i, P_1, i)$*
- *Q_i is consumed by P_2 at a time point i or later:*
  This can be represented by $before(P_2, Q_i, i, 2)$ for all i.
- *Q_{i+1} ends later than Q_i for all i: $before(Q_i, Q_{i+1}, 2, 2)$ for all i.*

The start of the space-optimal schedule is as follows and requires 3 units of space:



7 Applications

7.1 A Variant of Job Shop Scheduling

A variant of job shop scheduling is the following: Let there be n jobs (processes) that have to be performed on a number of identical machines. If the focus is on the question how many machines are sufficient for processing, then we can ignore the time and thus only specify the number of machines that are necessary for every single sub-job of any job (process). The necessary information is then the list of numbers (of machines) for every job. Note that also the number 0 is permitted. The trivial solution would be that all jobs run sequentially, in case the machine lists of every job are of the form $[0,k_2,\ldots,k_n,0]$.

If there are in addition (special) time constraints, for example every job starts immediately with a nonzero number of machines, and also all jobs end with a nonzero number of machines and they terminate all at the same time, then our algorithm SPOPTN can be applied in a nontrivial way and will compute the minimal total number of necessary machines.

In the case of further time constraints, Corollary 6.3 shows that in certain cases there are efficient algorithms and Theorem 6.4 shows that the problem, if there are general time constraints, is NP-complete.

Our approach and algorithm is related to resource constrained project scheduling [1] insofar as we are looking and optimizing the space resource of several given processes (projects). The difference is that in job shop and project scheduling the primary objective is to minimize the overall required time, whereas our algorithm computes a minimal bound of a resource (here space) not taking the time into account.

7.2 An Implementation for Checking Space Improvements

The interpreter CHFi calculates all possible interleavings for CHF-Programs (the program can be downloaded here: `www.ki.cs.uni-frankfurt.de/research/chfi`). The interpreter also provides a contrary mode that parallelizes as much as possible. We implemented Algorithm SPOPTN, see Definition 5.4. It can be used with the eager parallelization mode to calculate the required space for independent processes. The interpreter can be used to affirm the space improvement property of program transformations for examples and also to falsify conjectures of space improvements by comparing the required space returned by the interpreter for the same program before and after the transformation was applied. The development of an efficient method to compute the optimal space consumption and runtime of processes with synchronizations is left for future work.

8 Conclusion and Future Research

We developed an offline-algorithm SPOPTN that optimizes a given set of parallel and independent processes w.r.t. space and computes a space-optimal schedule with runtime $O((N+n)\log N)$ where n is the size of the input and N the number of processes. The algorithm is applicable to independent processes in concurrent (lazy-evaluating) languages. An application is to find the minimum resources that permit a global schedule in the resource-restricted scheduling projects problem.

Acknowledgements

We thank David Sabel for discussions and valuable remarks on the subject of this paper.

References

[1] Christian Artigues, Sophie Demassey & Emmanuel Néro (2008): *Resource-Constrained Project Scheduling*. ISTE and Wiley, London, UK, and Hoboken, USA, doi:10.1002/9780470611227.

[2] Haskell Community (2016): *Haskell, an advanced, purely functional programming language*. Available at `www.haskell.org`.

[3] M. R. Garey (1976): *The Complexity of Flowshop and Jobshop Scheduling*. *Mathematics of Operations Research* 1(2), pp. 117–129, doi:10.1287/moor.1.2.117.

[4] M. R. Garey & D. S. Johnson (1977): *Two-Processor Scheduling with Start-Times and Deadlines*. *Siam J. Comput.* 6(3), pp. 316–426, doi:10.1137/0206029.

[5] Jörgen Gustavsson & David Sands (1999): *A Foundation for Space-Safe Transformations of Call-by-Need Programs*. *Electr. Notes Theor. Comput. Sci.* 26, pp. 69–86, doi:10.1016/S1571-0661(05)80284-1.

[6] Jörgen Gustavsson & David Sands (2001): *Possibilities and Limitations of Call-by-Need Space Improvement*. pp. 265–276, doi:10.1145/507635.507667.

[7] S. Peyton Jones, A. Gordon & S. Finne (1996): *Concurrent Haskell*. In Guy L. Steele Jr. H.-J. Boehm, editor: *Proc. 23th ACM Principles of Programming Languages*, ACM, pp. 295–308, doi:10.1145/237721.237794.

[8] David Sabel & Manfred Schmidt-Schauß (2011): *A contextual semantics for concurrent Haskell with futures*. In Peter Schneider-Kamp & Michael Hanus, editors: *Proc. 13th ACM PPDP 2011*, ACM, pp. 101–112, doi:10.1145/2003476.2003492.

[9] David Sabel & Manfred Schmidt-Schauß (2011): *A Contextual Semantics for Concurrent Haskell with Futures*. Frank report 44, Institut für Informatik. Goethe-Universität Frankfurt am Main. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

[10] Manfred Schmidt-Schauß & Nils Dallmeyer (2018): *Space Improvements and Equivalences in a Functional Core Language*. In Horatiu Cirstea & David Sabel, editors: Proceedings Fourth International Workshop on *Rewriting Techniques for Program Transformations and Evaluation*, Oxford, UK, 8th September 2017, *Electronic Proceedings in Theoretical Computer Science* 265, Open Publishing Association, pp. 98–112, doi:10.4204/EPTCS.265.8.

[11] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2017): *Improvements for Concurrent Haskell with Futures*. Frank report 58, Institut für Informatik. Goethe-Universität Frankfurt am Main. `http://www.ki.informatik.uni-frankfurt.de/papers/frank/`.

[12] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2018): *Sequential and Parallel Improvements in a Concurrent Functional Programming Language*. In Peter Thiemann & David Sabel, editors: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, ACM, New York, USA, pp. 20:1–20:13, doi:10.1145/3236950.3236952.