

Task elimination may actually increase throughput time

Citation for published version (APA):

Schunselaar, D. M. M., & Verbeek, H. M. W. (2018). *Task elimination may actually increase throughput time*. arXiv.org. <http://arxiv.org/abs/1812.11793>

Document status and date:

Published: 01/01/2018

Document Version:

Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Task Elimination may Actually Increase Throughput Time

D.M.M. Schunselaar and H.M.W. Verbeek

Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
`d.m.m.schunselaar@tue.nl`, `h.m.w.verbeek@tue.nl`

Abstract. The well-known Task Elimination redesign principle suggests to remove unnecessary tasks from a process to improve on time and cost. Although there seems to be a general consensus that removing work can only improve the throughput time of the process, this paper shows that this is not necessarily the case by providing an example that uses plain M/M/c activities. This paper also shows that the Task Automation and Parallelism redesign principles may also lead to longer throughput times. Finally, apart from these negative results, the paper also show under which assumption these redesign principles indeed can only improve the throughput time.

1 Introduction

Within the Business Process Redesign community, a well-know redesign principle is the *Task Elimination* redesign principle, e.g., see [1]. The idea behind this redesign principle is that, by removing work, the overall throughput time will decrease and that costs are reduced [2]. After all, cases which would have been processed by a particular task can now skip this task and immediately arrive at a subsequent task.

Unfortunately, the redesign principle does not need to reduce the throughput time; it might actually *increase*. Due to overtaking cases, some cases may become slower while the overtaking cases may not become faster. As a result, the average throughput time increases.

Within [3], similar ideas are presented as the Task Elimination redesign principle, i.e., removal of bottlenecks to reduce the throughput time. Furthermore, within our earlier work [4], we have used, i.a., the Task Elimination redesign principle as a base line upon which we have created patterns to deduce if the throughput time of one process model is at-least-as-good as another process model. For all the aforementioned the same holds: an expected reduction of the throughput time might actually be an increase of throughput time. Within this paper, we sketch the boundaries as well as additional assumption under which the Task Elimination redesign principle will not increase the expected throughput time.

The structure of this report is as follows: first, we touch upon some preliminaries. Second, we present an example process model on which Task Elimination

does not reduce but *increases* the throughput time. Third, we present sets of additional assumptions under which Task Elimination does reduce the throughput time. Fourth, we show in the discussion section that the reasoning for Task Elimination can also be applied to Task Automation and Parallelism. Finally, fifth, we present our conclusions. An extended version of this paper can be found in [5, Ch. 7].

2 Preliminaries

Within this Section, we briefly introduce two modelling formalisms. The first, Petri nets [6], is used to show our motivating example why Task Elimination can result in an increased throughput time. The second, BCMP networks, is used to guarantee under which assumptions Task Elimination cannot result in an increased throughput time.

Petri nets Petri nets were first introduced in [6]. A Petri net consists of places (circles), transitions (rectangles), and edges between both (Fig. 1). Transitions denote the tasks of a Petri net and come in two flavours; labelled, and silent. Labelled transitions have a name, e.g., *A*. Silent transitions do not have a label and are usually indicated with a black rectangle.

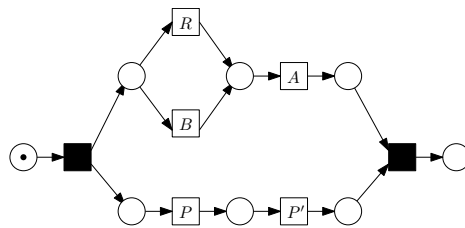


Fig. 1: Example Petri net.

The behaviour of a Petri net is encoded using *tokens*. Tokens are indicated by black dots. The state, or *marking*, of a Petri net is the distribution of tokens over the places. A transition t is allowed to *fire* if all places in its *preset* are marked, i.e., contain a token. The preset of a transition are those places that are the source of an edge to said transition. By firing a transition, a token from each of the preset places is *consumed* and on each of the *postset* places a token is *produced*. The postset of a transition are those places that are the target of an edge from said transition.

We have two special states; the *initial state* and the *final state*. The initial state of a Petri net is the state from which we started firing transitions, e.g., in Fig. 1, the left-most place is an initial state. The final state is the state in which the Petri net is considered to be terminated, e.g., the right-most place in Fig. 1.

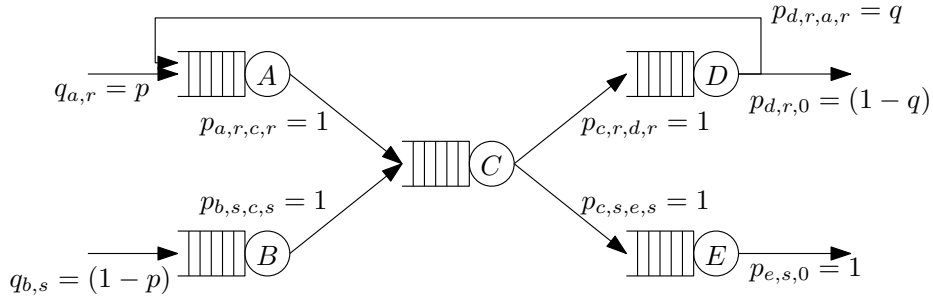


Fig. 2: Example queueing model.

BCMP Networks In order to explain *queueing networks* and *BCMP networks* [7] in particular, we start with the queueing model in Fig. 2. Where a Petri net has transitions, a queueing model has *service centres*. In queueing networks, the term *customer* is used to denote a token (although customers are not produced and consumed as in a Petri net). In its most simple form, customers arrive according to an exponential distribution with rate parameter λ . Customers receive service at a service centre by a *server* with a service rate of μ (again exponentially distributed). The average time in between two arrivals is $\frac{1}{\lambda}$ and the average time needed to handle a customer is $\frac{1}{\mu}$. The resource utilisation is denoted by ρ which in this case is equal to $\frac{\lambda}{\mu}$. If $\rho \geq 1$, then there is more work arriving per time unit than the service centre can handle. As such, in general, the requirement $\rho < 1$ is imposed.

Within queueing networks, customers can be of different *classes*. This would correspond to different types of tokens, e.g., a complaint by a gold customer or a complaint by a silver customer. Customers go from one service centre to the next service centre with a certain probability. These probabilities can depend on the customer class, e.g., in Fig. 2, after being processed at service centre A, a customer of class r moves to the queue of service centre C (again as a class r customer). This probability is encoded as $p_{a,r,c,r}$. Note that customers can change class while moving between service centres.

Next to customer classes, queueing networks can contain *feedback*, i.e., loops, and *fork-join*, i.e., parallelism. Furthermore, the queueing network can be open (customers arrive and leave the network), closed (a fixed number of customers are in the network), or mixed (open for some classes of customer, and closed for other classes of customers). Within this paper, we only consider open queueing networks without fork-joins. To denote externally arriving customers and customers leaving the queueing network, the queueing model in Fig. 2 has arrows without a source, or without a target. The incoming arrows, i.e., without a source, have a probability associated to them that a customer of a particular class will arrive at the service centre, e.g., the arrow with $q_{a,r} = p$ denotes that with probability p a customer of class r arrives at service centre A. Conversely, outgoing arrows have a probability that a customer of a particular class

will leave the network after being processed at a particular service centre, e.g., $p_{d,r,0} = (1 - q)$.

Within a BCMP network, the service centres can be any of the following types [7]:

1. The service principle is First in First out. All customers receive the same service time distribution which is an exponential distribution.
2. There is a single server at a service centre. The service principle is server sharing. This means that every time unit every customer at this service centre receives a service rate proportional to the number of customers at this service centre. The service time distribution is arbitrary.
3. The number of servers in the service centre is greater than or equal to the maximum number of customers that can be queued.
4. There is a single server at a service centre, the queueing principle is preemptive-resume last-come-first-serve. Each class of customers may have a distinct distribution for the service times.

Type 2, 3 and 4 service centres all assume service distributions with rational Laplace transforms. In [8], for various distributions, it is shown they have rational Laplace transforms, e.g., exponential, hyperexponential, and hypoexponential. In the work of [9], it is stated that an arbitrary distribution may be as closely approximated as one may wish using a convex mixture of Erlang distributions. This mixture has a rational Laplace transform.

One of the results from BCMP networks is that, for *open* networks where the *Poisson* arrival of new customers is *independent* of the state of the network, the number of customers in each service centre are *independent* random variables [7]. This means that the throughput time at a service centre is *independent* of the other service centres.

3 Motivating example

Figure 3 [5] shows our example process model at the left-hand side, and the process model after applying the Task Elimination design principle on the right-hand side. As one can see, in the right-hand model, task *R* has been eliminated. As a result, one would expect that the right-hand model would have a throughput time which is at-most that of the left-hand model. However, as we will show, this is not the case.

In the example process models, we have two types of cases: *red* cases (executing task *R*, and skipping task *B*) and *blue* cases (executing task *B*). Task *A* is executed for *all* cases. Finally, tasks *P* and *P'* are *parallel* to the aforementioned tasks. For now, assume the processing time distribution of each task is *deterministic* with the value as indicated in Fig. 3, e.g., task *A* takes 5 time units. Furthermore, assume the cases arrive in a *burst*, i.e., within a very short period of time a large amount of cases arrive. Thereafter, for a long period of time, no cases arrive. Let the time between the bursts be such that the model is empty when a new burst arrives. During a burst, a total of *N* cases arrive.

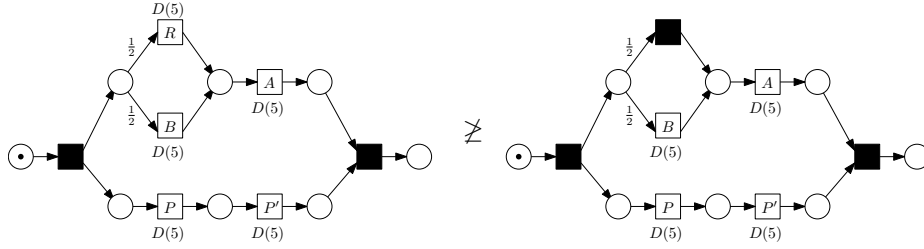


Fig. 3: Example where *Task Elimination* increases the throughput time.

Assume that $\frac{1}{2}N$ red cases and $\frac{1}{2}N$ blue cases arrive. Furthermore, assume that all tasks process the cases in a First in First out (FiFo) order.

In the original process model, in the parallel branch, tasks P and P' process the cases in the order they arrive at the process model. At the tasks R and B , the cases are also processed in the order they arrive. At task A , the ordering can be slightly different, i.e., two consecutively arriving cases can both be blue cases and then a red case can be in between at A . In general, the order of the cases at A is similar to the arrival of cases to the process model. Which in turn is similar to the order of the cases at P and P' . As a result, the time cases spend on synchronising is relatively small at the transition joining the parallel branches (in the remainder, this transition is called: AND-join).

In the redesigned process model, we have eliminated task R . This means that just after a burst has occurred, the queue at A is filled with $\frac{1}{2}N$ red cases. Behind these red cases, the blue cases are queued as these first have to be processed by task B . In the parallel branch nothing has changed, i.e., tasks P and P' still process the cases in the order they arrive at the process model. This means that at the AND-join, the top branch has been resorted on red first and blue last, while in the bottom branch, no resorting was done and the cases arrive in the same order as they arrived at task P . Only after the $\frac{1}{2}N$ red cases have been processed by task A , blue cases can be synchronised.

By eliminating task R , the sorting of cases before the AND-join has changed dramatically in the upper branch, while it did not change in the lower branch. As a result, task A started processing cases which had no priority, as in the bottom branch, no work has been done on them. As such, there was no need to work on these cases by task A . This while cases which were being processed at tasks P and P' (blue cases) were at the end of the queue of task A . The *overtaking* of cases makes that at the synchronisation the *wrong* cases are waiting to be synchronised, i.e., cases which cannot be synchronised with any of the cases at the other branch.

In our example, we have used burst arrivals and deterministic processing times. If we look into a *Poisson* arrival stream and *exponential distributions* on the tasks, i.e., not deterministic, then one can still observe the same phenomenon. Within a *Poisson* arrival stream, it is still possible that multiple cases arrive within a short period of time. The earlier sketched overtaking of cases might be

less prominent (there are fewer cases at the same time within the process model) but the end effect will be the same, i.e., Task Elimination can have a *negative* effect on the throughput time.

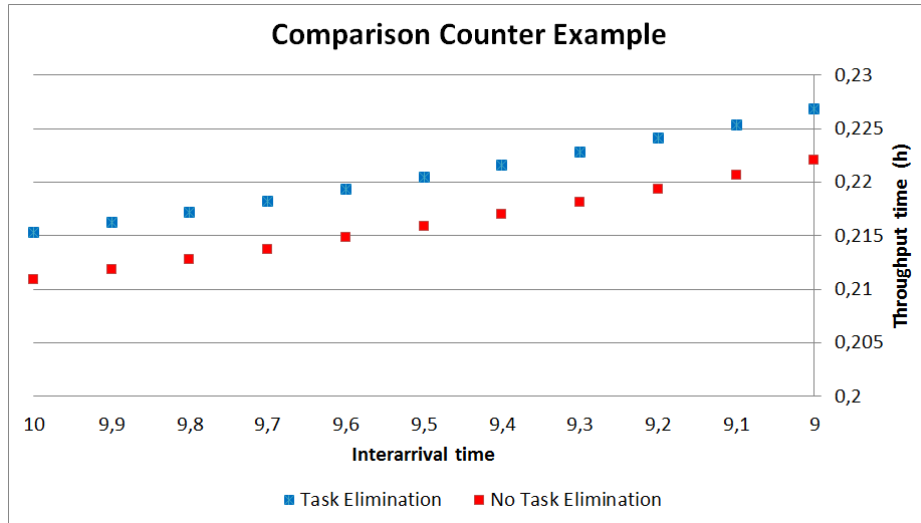


Fig. 4: Quantitative results for our example. The arrival process is Poisson and the processing times are deterministic 5 minutes with FiFo queues at every task. The blue line is with Task Elimination and the red line is no Task Elimination.

We have simulated both models from Fig. 3 with different Poisson arrival distributions (interarrival times as shown in Fig. 4) with 30 replications and a replication length of 1000 hours using L-SIM [10]. All tasks have a deterministic processing times of 5 minutes. Furthermore, every task has a FiFo queue. Finally, every task has exactly 1 dedicated resource allocated. The results are depicted in Fig. 4 (blue is Task Elimination and red is no Task Elimination). One can indeed observe that *Task Elimination* has a negative effect on the throughput time.

4 Additional assumptions

The above example shows the need for additional assumptions under which Task Elimination gives an intuitive result. Within this Section, we assume there are two models M_1 and M_2 where M_2 is obtained from M_1 by applying the Task Elimination design principle on some tasks (possibly none), i.e., these tasks have become silent transitions.

4.1 Scheduling

In this set of assumptions [5], we assume that both process models follow the same schedule in executing the cases (see Fig. 5 for an illustration of an example schedule). This means that, even though a case c' arrives earlier at a particular task than case c , the execution of c' is possibly delayed to work on c first (even though c might arrive later at the respective task than c'). This setting is particularly applicable in situations where a fixed set of cases needs to be processed during a certain time period, e.g., order picking, and if a-priori a schedule is made to optimally execute the cases. It is easy to show that the redesign principles and conversely the patterns in [4] can safely be applied. This follows from the fact that by eliminating tasks in the execution of cases, it is always possible to project the schedule for executing the cases of a model with more tasks to a model with fewer tasks.



Fig. 5: Example execution schedule for the Petri nets in Fig. 3. On the y-axis, we have the tasks with the resources (r_i). On the x-axis, we have the time. The c_i s are the cases and the small downward arrows are the arrival of cases.

4.2 BCMP networks

For BCMP networks [7], exact solutions exist to compute the mean throughput time. If a Petri net can be transformed to a BCMP network, then we can show that *Task Elimination* indeed returns the correct result. This follows from the fact that the throughput time at a task is *independent* of the other tasks. By having fewer tasks, the throughput time of the cases does not increase. As a result, the throughput time of the process model will not increase and probably will decrease. A Petri net can be transformed to a BCMP network if it is a state machine, i.e., every transition has exactly one place in its preset and exactly one place in its postset. Furthermore, the tasks within a Petri net need to adhere to the types supported by the BCMP network [5]. Finally, the arrival process of new cases needs to be *Poisson*.

The transformation of a state machine to a BCMP network is relatively straightforward (see Fig. 6 for an example). As stated, a BCMP network consists of service centres, an arrival rate, and probabilities for customers moving

that every edge in the Petri net has a probability associated to it (in Fig. 6, we have illustrated some of the probabilities). The probability on the outgoing edge of a transition is by definition 1, the probability of the incoming edge of a transition t is the probability that the transition will fire if there is a token in its preset (this probability is indicated by p_t). Next to this, a place $o \in O$ has a probability that a token will leave the Petri net from o (indicated by p_o), and a place $i \in I$ have a probability that a token is produced in i (indicated by p_i). We assume that the sum of all p_i is equal to 1, e.g., $p_{i_1} + p_{i_2} = 1$ in Fig. 6. For every set T of transitions that have the same preset, we have $\sum_{t \in T} p_t = 1$, e.g., $p_{t_1} + p_{t_3} = 1$ in Fig. 6.

Every labelled transition in the Petri net is transformed into a service centre with its respective distribution, we indicate with s_t the service centre of transition t . The arrival probability of a new customer at service station s_t (indicated by q_{s_t} in Fig. 6) is 0 if the preset of t is not in I and $p_i p_t$ if it is. Mutatis mutandis, we can deduce the probability that a customer leaves the network after having received service at a service station. The probability of a customer arriving at service station $s_{t'}$ after having received service at service centre s_t (indicated by $p_{s_t, s_{t'}}$ in Fig. 6) is $p_{t'}$ if the preset of t' equals the postset of t (note that we have a state machine so every transition only has a single place in its preset/postset). In case of silent transitions, there might be a path of silent transitions between the postset of t and the preset of t' . In this case, the probability $p_{s_t, s_{t'}}$ is equal to the sum of the probabilities of all possible paths between t and t' , e.g., from transition C to transition C in Fig. 6. The probability of a path is equal to the product of the probabilities of the edges in a path. If no such path exists, then $p_{s_t, s_{t'}} = 0$ (implicit in Fig. 6).

Using the above sketched transformation, we can transform a state machine into a BCMP network. Hence, Task Elimination will not result in an increase of throughput time within state machines that can be transformed.

5 Discussion

We have focussed primarily on Task Elimination. These results are also applicable to the *Task Automation* [1] and *Parallelism* redesign principles [2]. The Task Automation redesign principle essentially reduces the the time spent in a task to almost 0. As a result, the same reasoning as for Task Elimination can be followed. Within the Parallelism redesign principle, multiple tasks are put in parallel to decrease the throughput time. If we take our example process model in Fig. 3 and we would replace R by the sequential execution of R_1, \dots, R_n such that duration of R is the same as the sum of the durations of R_1, \dots, R_n and the duration of R_1, \dots, R_n is close to 0, then by placing R_1, \dots, R_n in parallel, we possibly would make that branch significantly faster. As a result, in front of A , we would have only red cases enqueued.

6 Conclusion

Within this paper, we have shown that various redesign principles and in particular *Task Elimination* does not need to reduce the throughput time under all circumstances; under some circumstances, the throughput time actually *increases*. As a result, our patterns presented in [4] do not always yield the correct result. At the same time, we have presented additional assumptions/settings under which Task Elimination always does give the correct result.

References

1. Limam Mansar, S., Reijers, H.A.: Best Practices in Business Process Redesign: Validation of a Redesign Framework. *Computers in Industry* **56**(5) (June 2005) 457–471
2. Reijers, H., Mansar, S.L.: Best practices in business process redesign: an overview and qualitative evaluation of successful redesign heuristics. *Omega* **33**(4) (2005) 283 – 306
3. Goldratt, E.M.: *The goal : A Process of Ongoing Improvement*. Gower, Aldershot (2004)
4. Schunselaar, D.M.M., Verbeek, H.M.W., Reijers, H.A., Aalst, W.M.P. van der: Using Monotonicity to Find Optimal Process Configurations Faster. In Accorsi, R., Ceravolo, P., Russo, B., eds.: *Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014)*, Milan, Italy, November 19-21, 2014. Volume 1293 of *CEUR Workshop Proceedings.*, CEUR-WS.org (2014) 123–137
5. Schunselaar, D.M.M.: *Configurable Process Trees: Elicitation, Analysis, and Enactment*. PhD thesis, Eindhoven University of Technology (2016)
6. Petri, C.A.: *Kommunikation mit Automaten*. PhD thesis, Institute für instrumentelle Mathematik, Bonn, Germany (1962) In German.
7. Baskett, F., Chandy, K.M., Muntz, R.R., Palacios, F.G.: Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *Journal of the ACM* **22**(2) (1975) 248–260
8. Cox, D.R.: A use of Complex Probabilities in the Theory of Stochastic Processes. *Mathematical Proceedings of the Cambridge Society* **51**(02) (1955) 313–319
9. Gelenbe, E., Muntz, R.R.: Probabilistic Models of Computer Systems - Part I (Exact Results). *Acta Informatica* **7** (1976) 35–60
10. L-SIM: <http://www.lanner.com/en/1-sim.cfm> Last accessed 2015-10-21.