Contents

1 Abstract

Functional Reactive Programming (FRP) offers an attractive alternative to traditional real-time programming paradigms on account of its inherent atomicity by function call, which can simplify the process of static analysis through stronger invariants. However, these guarantees come at the price of a more complicated timing analysis required by the Abort-And-Restart (ANR) scheduling method which Priority-Based FRP (P-FRP) employs and which provides the underpinning for the aforementioned atomicity. Earlier work by Chaitanya Belwal and Albert M. K. Cheng developed and explored the gap-enumeration algorithm, a method for characterizing the worst case response time (WCRT) of each task in a set of tasks under the ANR model that outperformed time-accurate simulation in terms of efficiency. Here the work is revisited with the aim of further optimizing the performance of the gap-enumeration algorithm

2 Introduction

The functional programming paradigm has a long history of feeding innovations in other programming paradigms, often well ahead of adoption thereof in the latter. For example, anonymous lambda closures and first-class function passing, which were not available in the C++ language standard until version 11[8], have been a staple of languages such as LISP since at least the 1970s[6, p. 2]. The use of recursion in functional programming languages makes inductive proofs of program correctness much simpler than similar iterative constructions in imperative languages, particularly in languages that allow side-effects by default. It is easier to reason about idempotent functions than procedures with the potential for side-channel input via shifting program state.

It raises the question then as to why the field of real-time systems programming has not more readily adopted the functional programming paradigm. The answer lies in the division between application-level programming and systems-level programming. While functional programming at the application level can fuel rapid application development, implementing runtime systems that enable such programming is a difficult and complicated task. The software transactional memory model described by Simon Peyton Jones in his essay “Beautiful Concurrency”[11] can still fall victim to thread starvation[9], and the stateless idempotency and atomicity guarantees of function execution ultimately have to be implemented on stateful transistors.

In an age of increasing systems complexity, it behooves practitioners of real-time systems programming to investigate whether systems can be built to employ functional programming techniques that might alleviate the complexity of the necessary systems analysis. Some research has been done in this area, but one of the primary obstacles to analysis is the required scheduling model. While much of real-time systems scheduling research has focused on preemptive scheduling since Liu and Layland’s seminal paper[10], the guarantees of functional execution atomicity in P-FRP scheduling requires abort-and-restart scheduling[3, p. 1].

Characterizing the ANR model is expensive and difficult, as indicated by Belwal and Cheng[3, p. 2]. However, if the characterization can be made less expensive, this in turn can facilitate more research into the adoption of functional programming techniques in real-time systems programming, which can provide a foundation for building safer, more reliable systems in a more systematic manner. The following work attempts to optimize the original work by Belwal and Cheng to provide an efficient worst case response time determination for the lowest priority task in a P-FRP task set. The terms and variables used follow the original work[3, pp. 3-4] except that the task period is denoted P and in cases where terms and variables are otherwise explicitly specified.

3 Methodology

3.1 Implementing the Original Algorithm

The original gap-enumeration algorithm is given below with some slight modifications for clarity.

Algorithm 1 Gap-Enumeration Algorithm[3, p. 11]

```
function GAP-ENUMERATE-DYNAMIC($\Gamma_n$, $\tau_j$, $w$)
    $L \leftarrow \lceil \frac{P_j}{w} \rceil$
    $U \leftarrow P_j + \lceil \frac{P_j}{w} \rceil$
    while $L < U$ do
        $\sigma_n(P|_0^L) \leftarrow \{[0, L)\}$
        for $i \leftarrow n-1, j$ do
            $\sigma_{i-1}(P|_0^L) \leftarrow$ GAP-TRANSFORM($L, \sigma_i(P|_0^L), \Gamma_n, j$)
            if $\sigma_{i-1}(P|_0^L) = \emptyset$ then
                return -1
            end if
        end for
        $[t_1, t_2) \leftarrow$ GAP-SEARCH($\sigma_j(P|_0^L), C_j$)
        if $t_1 \geq 0$ then
            $RT_j \leftarrow t_1 + C_j$
        end if
        if $RT_j < P_j$ then
            return $RT_j$
        end if
        $L \leftarrow L + \lceil \frac{P_j}{w} \rceil$
    end while
    return -1
end function
```

The original work set the post-search t_1 validity test as $t_1 > 0$, but this precludes scheduling work at $t = 0$, so the test $t_1 \geq 0$ was substituted.

The original gap-transformation algorithm is given below with some slight modifications for clarity as necessitated by implementation details.

Algorithm 2 Gap-Tranformation Algorithm[3, p. 12]

```
function GAP-TRANSFORM($W$, $\sigma_i(P|_0^L)$, $\Gamma_n$, $j$)
    $J_i \leftarrow \lceil \frac{W-R_j}{P_j} \rceil$
    for $q \leftarrow 1, J_i$ do
        $t \leftarrow R_j + P_j(q-1)$
        $kgap \leftarrow$ MIN-GAP($\sigma_i(P|_0^L)$)
        $t_1 \leftarrow entry[kgap]$
        $t_2 \leftarrow exit[kgap]$
        while $kgap \neq$ NIL($\sigma_i(P|_0^L)$) do
            if $t_1 > t + P_j$ then
                return $\emptyset$
            end if
            if $t < t_1$ then
                $t \leftarrow t_1$
            end if
            if $t_1 \leq t$ and $t < t_2$ then
                GAP-DELETE($\sigma_i(P|_0^L), [t_1, t_2)$)
                if $t + C_j = t_2$ then
                    GAP-INSERT($\sigma_i(P|_0^L), [t_1, t)$)
                    exit while
                end if
                if $t + C_j < t_2$ then
                    GAP-INSERT($\sigma_i(P|_0^L), [t_1, t)$)
                    GAP-INSERT($\sigma_i(P|_0^L), [t + C_j, t_2)$)
                    exit while
                end if
                if $t + C_j > t_2$ then
                    GAP-INSERT($\sigma_i(P|_0^L), [t_1, t)$)
                end if
            end if
            if $t1 = entry[kgap]$ then
                $kgap \leftarrow$ SUCCESSOR-GAP($kgap$)
            end if
        end while
    end for
    $\sigma_{i-1}(P|_0^L) \leftarrow \sigma_i(P|_0^L)$
    return $\sigma_{i-1}(P|_0^L)$
end function
```

Note that the test before performing an advance along the tree is required since the reinserted node whose entry time is $t + C_j$ in the case where the k-gap is filled with slack at the end must not be skipped in the gap traversal.

As in the original work, the GAP-SEARCH[3, pp. 12-13] algorithm is simply a variation of IN-ORDER-TREE-WALK[5, pp. 245-246] that returns either when a gap is found that fits the current task for which an execution gap is being sought or when all gaps have been exhausted and no suitable gap has been found.

3.2 Task Set Generation

Task set generation generally followed the parameters outlined in the original work[3, p. 14]. Tasks were randomly generated in 3 groups (A, B, and C) where group A was comprised of sets with 3 tasks, group B with sets of 5 tasks, and group C with sets of 7 tasks. All tasks were implicit-deadline and implicit-priority, i.e., the deadline was implicitly defined as equal to the task release period, and the priority was implicitly defined as the task period with higher priority given to tasks with smaller periods. The prioritization of tasks in this way is required in order for the gap-enumeration algorithm to work and correctly model the behavior of the P-FRP scheduler.

Task periods were randomly distributed over a uniform distribution within the interval $[40, 60)$ by means of the `std::uniform_int_distribution` facility of the C++ language[2]. The distribution was generated by of the `std::shuffle_order_engine`[1] pseudo-random number generator (PRNG) with the parameters of the default provided `knuth_b` instance seeded by the nanosecond epoch time immediately prior to the instantiation of the PRNG.

Task computation times were randomly distributed over the interval $[4, 10)$ in the same manner using an independent distribution and PRNG. The distributions were queried over a series of iterations of two nested loops, the outer loop spanning the population of task sets and the inner loop spanning the task set size. As each task set was constructed, the tasks were sorted in increasing order by priority, i.e., most urgent and frequent task first, with ties broken by computation time in decreasing order thereof.

Within a task set population, each task set was unique in terms of the task set serialization. This is to say that the shell command

```
wc -l $TASKSETFILE
```

yielded the same results as

```
sort $TASKSETFILE | uniq | wc -l
```

Since each task set was deterministically sorted first by priority and then by computation time, the uniqueness of the string serialization of the task set is equivalent to the uniqueness of the task set itself.

3.3 Optimization 1

Having implemented the original algorithm, it became clear that the constant deletion and reinsertion costs were a major component of the algorithm's computational cost, validating conclusions drawn in the original work[3, p. 17]. However, rather than devising a space-costly indexing scheme to try and speed up the tree modification operations, it was determined that a less expensive adjustment might yield even better efficiency after observing that the original algorithm reinserted a k-gap of size zero (0) back into the tree after a job had been scheduled into a gap either for a completed or aborted execution. Since these null gaps could never fit a task in following iterations of the gap-transformation function, including them in the tree incurred extraneous costs.

Consequently, modifications were made to perform a size check on k-gaps prior to reinsertion to prevent null gaps from being introduced as outlined in the following algorithm. The changes in the algorithm are highlighted in red for legibility.

Algorithm 3 Gap-Tranformation Algorithm Optimization 1: No Zero Gaps Reinserted

```
function GAP-TRANSFORM-POSITIVE($W$, $\sigma_i(P|_0^L)$, $\Gamma_n$, $j$)
    $J_i \leftarrow \lceil \frac{W-R_j}{P_j} \rceil$
    for $q \leftarrow 1, J_i$ do
        $t \leftarrow R_j + P_j(q-1)$
        $kgap \leftarrow$ MIN-GAP($\sigma_i(P|_0^L)$)
        $t_1 \leftarrow entry[kgap]$
        $t_2 \leftarrow exit[kgap]$
        while $kgap \neq$ NIL($\sigma_i(P|_0^L)$) do
            if $t_1 > t + P_j$ then
                return $\emptyset$
            end if
            if $t < t_1$ then
                $t \leftarrow t_1$
            end if
            if $t_1 \leq t$ and $t < t_2$ then
                GAP-DELETE($\sigma_i(P|_0^L), [t_1, t_2)$)
                if $t + C_j = t_2$ then
                    if $t > t_1$ then
                        GAP-INSERT($\sigma_i(P|_0^L), [t_1, t)$)
                    end if
                    exit while
                end if
                if $t + C_j < t_2$ then
                    if $t > t_1$ then
                        GAP-INSERT($\sigma_i(P|_0^L), [t_1, t)$)
                    end if
                    GAP-INSERT($\sigma_i(P|_0^L), [t + C_j, t_2)$)
                    exit while
                end if
                if $t + C_j > t_2$ then
                    if $t > t_1$ then
                        GAP-INSERT($\sigma_i(P|_0^L), [t_1, t)$)
                    end if
                end if
            end if
            if $t1 = entry[kgap]$ then
                $kgap \leftarrow$ SUCCESSOR-GAP($kgap$)
            end if
        end while
    end for
    $\sigma_{i-1}(P|_0^L) \leftarrow \sigma_i(P|_0^L)$
    return $\sigma_{i-1}(P|_0^L)$
end function
```

3.4 Optimization 2

Further examination of the original algorithm led to the conclusion that the algorithm was not making full utilization of the properties of the red-black tree data structure. Each iteration of the gap-transformation function had to find the minimum leaf at the start, an $O(h)$ operation from the root of a tree of size h[5, pp. 246-249]. Given that the height of a red-black tree of size n is a maximum of $O(log_2(n))$[5, pp. 263-265], the asymptotic cost is thus $O(log_2(n))$ just to start a transform iteration.

The sequential walk over the k-gaps in the tree meant that every iteration also incurred an $O(n)$ cost. Within each iteration was the potential for $O(log_2(n))$ insertion and deletion costs[5, pp. 268-277], as well as another possible $O(log_2(n)$ operation to find the successor k-gap.

In order to avoid these compounding costs, the next optimization preserved the prohibition on reinsertion of null gaps and changed the underlying data structure to a linked list. By doing this, the start of a transform iteration became a constant-time operation since the list maintains a pointer to its own head. To simplify operations, a nil sentinel was also maintained at the tail. The transform iteration walk remained $O(n)$, but this was unavoidable since the nature of the ANR scheduling model demanded that gaps be filled with aborted jobs until the job reached completion or the gaps were exhausted. The insertion and deletion operations became constant-time since the gaps were simply spliced in and spliced out, respectively. In some cases, no pointer adjustments were even required since the existing k-gap could simply be modified *in situ* with regard to its entry and exit.

Algorithm 4 Gap-Tranformation Algorithm Optimization 2: No Zero Gaps Reinserted, Linked List

```
function GAP-TRANSFORM-POSITIVE-LIST($W$, $\sigma_i(P|_0^L)$, $\Gamma_n$, $j$)
    $J_i \leftarrow \lceil \frac{W-R_j}{P_j} \rceil$
    for $q \leftarrow 1, J_i$ do
        $t \leftarrow R_j + P_j(q-1)$
        $kgap \leftarrow$ HEAD$(\sigma_i(P|_0^L))$
        $t_1 \leftarrow entry[kgap]$
        $t_2 \leftarrow exit[kgap]$
        while $kgap \neq$ TAIL$(\sigma_i(P|_0^L))$ do
            if $t_1 > t + P_j$ then
                return $\emptyset$
            end if
            if $t < t_1$ then
                $t \leftarrow t_1$
            end if
            if $t_1 \leq t$ and $t < t_2$ then
                if $t + C_j \geq t_2$ then
                    if $t > t_1$ then
                        $entry[kgap] \leftarrow t_1$
                        $exit[kgap] \leftarrow t$
                    else
                        SPLICE-OUT$(\sigma_i(P|_0^L), kgap)$
                    end if
                    if $t + C_j = t_2$ then
                        exit while
                    end if
                end if
                if $t + C_j < t_2$ then
                    $entry[kgap] \leftarrow t + C_j$
                    $exit[kgap] \leftarrow t_2$
                    if $t > t_1$ then
                        SPLICE-IN$(\sigma_i(P|_0^L), kgap, [t_1, t))$
                    end if
                    exit while
                end if
            end if
            if $t1 = entry[kgap]$ then
                $kgap \leftarrow$ NEXT$(kgap)$
            end if
        end while
    end for
    $\sigma_{i-1}(P|_0^L) \leftarrow \sigma_i(P|_0^L)$
    return $\sigma_{i-1}(P|_0^L)$
end function
```

3.5 Metrics

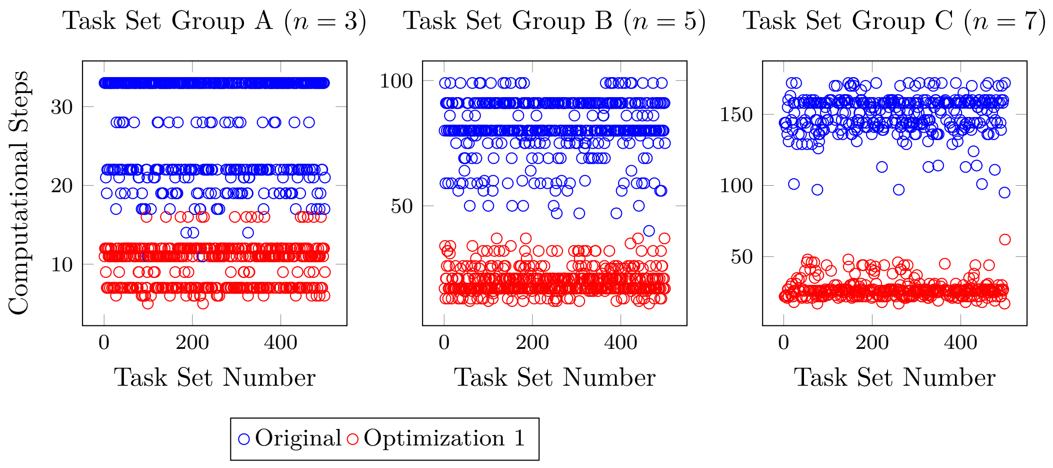
As with the original work, determining the number of computational steps required for each implementation was a desired metric. However, unlike the original work, which added a blanket $O(log_2(n))$ cost to every red-black tree insertion and deletion[3, p. 14], empirical computational steps were counted within the major loops and operations inside the various functions.

For the red-black tree versions, this included Gap-Enumerate-Dynamic, Gap-Transform or Gap-Transform-Positive, and Gap-Search, as well as RB-Insert, RB-Delete, Tree-Successor, and Tree-Minimum. For the linked-list version, this included Gap-Enumerate-Dynamic, Gap-Transform-Positive-List, and Gap-Search, as well as Splice-In and Splice-Out. The Head, Tail, and Next functions were not included since they represent simple pointer accesses.

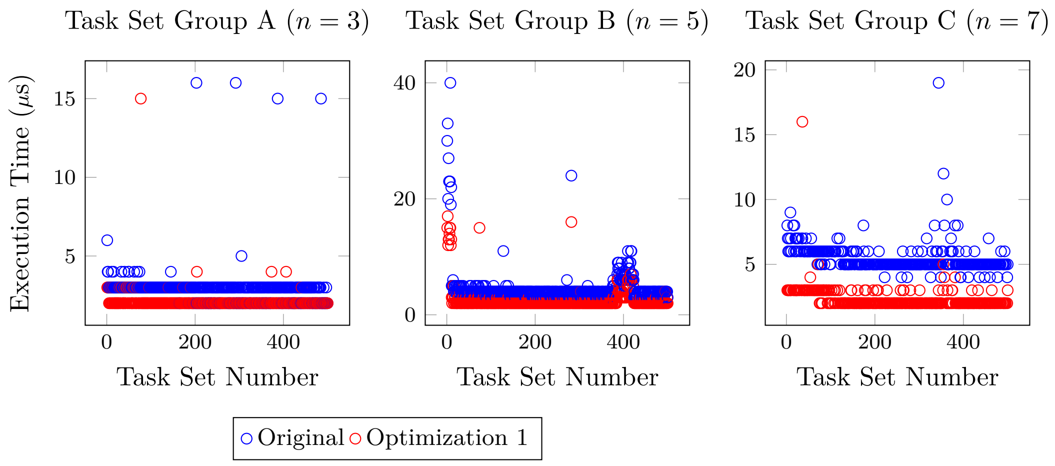
For each run of the gap-enumeration algorithm, the number of k-gaps present in the final tree or list was also counted as a measure of space complexity. Wall-clock execution time was measured with the POSIX system clock `CLOCK_MONOTONIC`.

4 Results

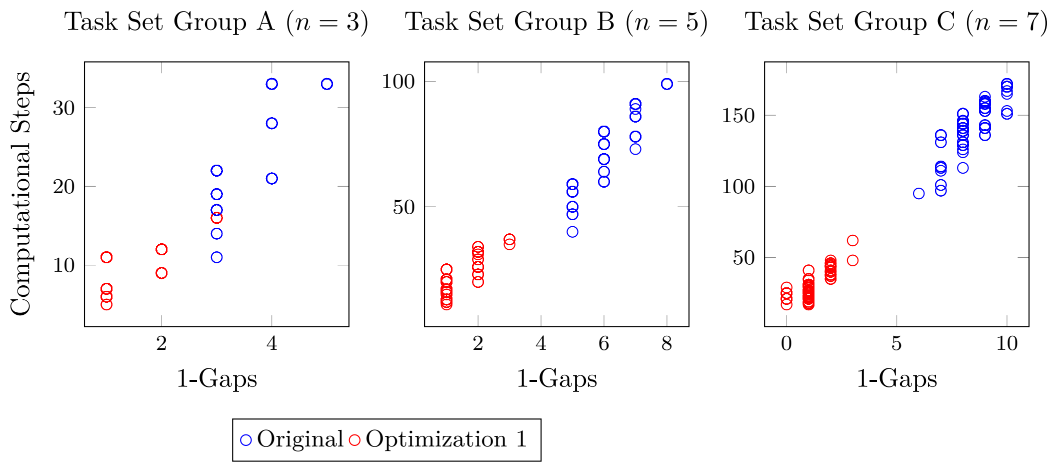
4.1 Original Method vs. Optimization 1



As expected, the prohibition on null k-gaps in the gap tree significantly reduced the number of computational steps required for gap enumeration. The range of computational steps required for Optimization 1 has a fair amount of jitter, probably owing to the nature of the exact tree structure as informed by the task set definition and the interplay between the various tasks in the ANR scheduling model. The variance is most likely a function of the red-black tree insertion and deletion operations.

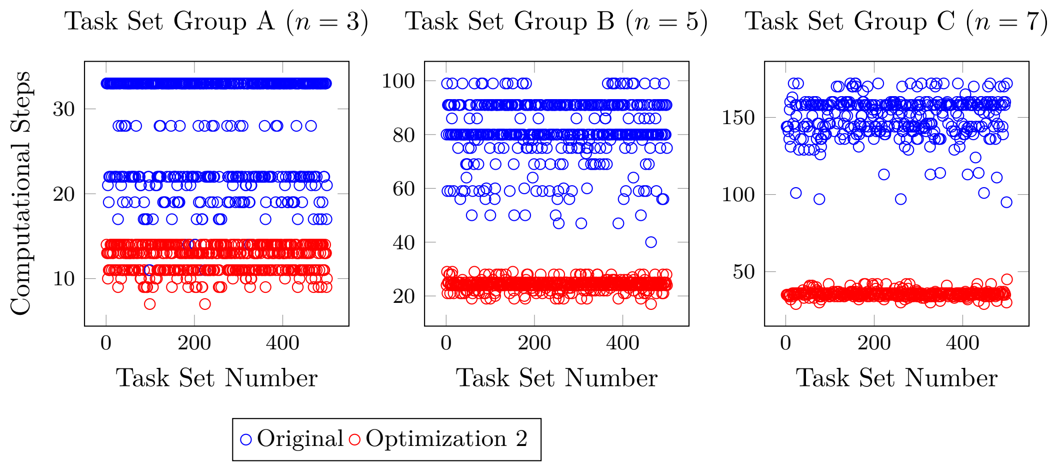


The results exhibit a decent speedup in wall-clock execution time. However, these results must be considered in the context of the testing platform as the exact execution time speedup will depend on the hardware and operating system being used. The computational step measurement is more informative from a hardware-agnostic, purely algorithmic standpoint.

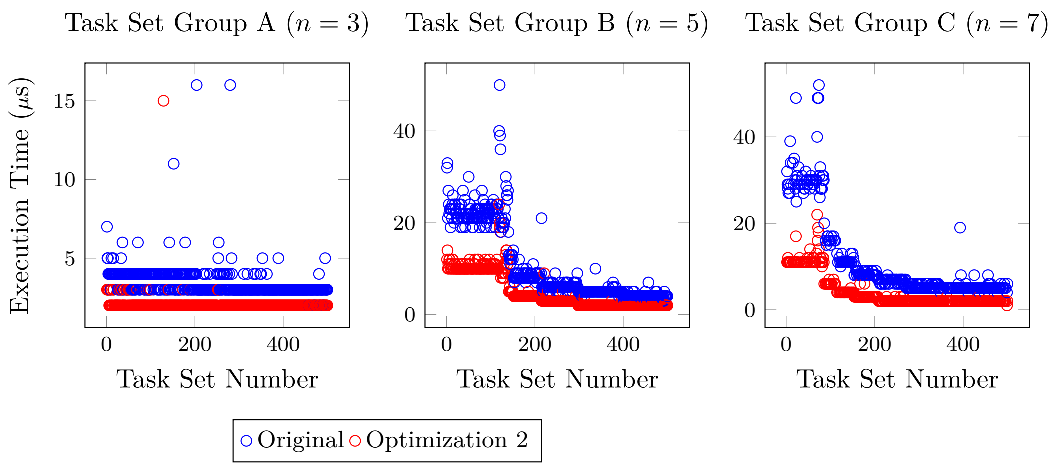


The results here clearly delineate the drop in space complexity induced by not reinserting the null k-gaps and the subsequent drop in empirical time complexity. The gap between the cost clusters increases proportional to the task set size.

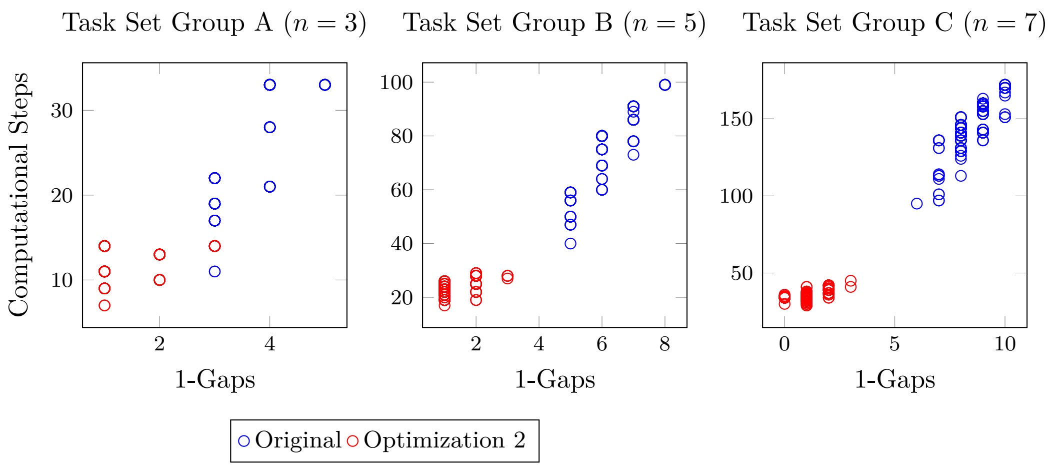
4.2 Original Method vs. Optimization 2



As expected, the linked-list implementation significantly reduced the number of computational steps required for gap enumeration. The range of computational steps required for Optimization 2 has less jitter than Optimization 2 since the insertion and deletion costs were constant. Considering the previous graphs, Optimization 2 slightly outperforms Optimization 1 in general.



The results for Optimization 2 exhibit a decent speedup in wall-clock execution time. There seems to be an anomaly with regard to the execution time of task sets listed earlier in the population. The exponential decay as population identifier increases likely stems either from whole program initialization costs or from interference due to external, transient load on the platform under test. Further experiments could validate or falsify these hypotheses.



The results here reproduce the effect seen from the first optimization with regard to space and time complexity. The improvement over the first optimization is clearly visible in the task sets from group C.

5 Conclusions

Avenues for future work include more comprehensive tests with different types of task sets. Although Belwal and Cheng illustrated the ANR scheduling model and the gap enumeration algorithm succinctly with a task set whose task periods varied greatly within the set, the experiments run both in the original work and the present work covered fairly homogeneous task sets by contrast. The homogeneity of the task sets tested does not fully explore the pathology of the worst case scenario for the algorithm and the underlying data structures, which reach their nadir in task sets where the higher priority tasks are frequently released within the search hyperperiod, thereby creating many disjoint k-gaps which must be walked by lower priority tasks in search of an execution space.

No effort was made in this work to improve the gap-search algorithm, which remains an $O(n)$ operation. Future work could focus on some form of efficient meta-indexing to reduce the search time. Ideally, this would be built during the gap-transformation function to amortize the time costs required for such indexing. Alternatively, investigations into a more flexible data structure that could encode both ordinality in time and gap-size into the structure's search metadata may prove beneficial, provided that the space costs did not become too onerous.

As demonstrated by the experiments in this work, space cost becomes time cost. Determining what information is absolutely essential for an algorithm to function and discarding what is extraneous can garner significant time savings. The structures used by the original gap-enumeration algorithm would be more amenable to whole schedule reconstruction than the techniques presented in this work, but if all one needs is a measure of worst case response time for a given task, the benefits of the optimizations presented here are undeniable. Furthermore, it would not be difficult to maintain an additional schedule structure that accumulated the positive fills of the gaps during the execution of the gap-enumeration algorithm. Continual, careful optimization of the techniques first explored by Belwal and Cheng has the potential to reap increasing benefits within the growing field of P-FRP response time analysis.

6 Acknowledgments

The author would like to thank Dr. Albert M. K. Cheng for his introduction to the original work by Chaitanya Belwal and himself. Furthermore, the author would like to acknowledge Joseph Wright et al. for their work on the beamer LaTeX package [12] which was used in making the presentation slides; Szász János for the algorithmicx LaTeX package[7] which was used to typeset the algorithms; and Domenico Camasta for comments regarding some best practices on typesetting algorithms[4].

References

[1] Anonymous. *std::shuffle_order_engine*. cppreference.com. Sept. 19, 2014. URL: `https://en.cppreference.com/w/cpp/numeric/random/shuffle_order_engine` (visited on 04/27/2018).

[2] Anonymous. *std::uniform_int_distribution*. cppreference.com. Feb. 18, 2017. URL: `https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution` (visited on 04/27/2018).

[3] Chaitanya Belwal and Albert M. K. Cheng. *Determining Actual Response Time in P-FRP*. Technical Report UH-CS-10-05. University of Houston Computer Science Department, June 28, 2010.

[4] Domenico Camasta and et al. *Using Algorithm2e package in Beamer*. May 1, 2014. URL: `https://tex.stackexchange.com/questions/174631/using-algorithm2e-package-in-beamer` (visited on 04/29/2018).

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press and McGraw-Hill Book Company, 1997. ISBN: 0-262-03141-8.

[6] Paul Hudak et al. *A History of Haskell. Being Lazy With Class*. Apr. 16, 2017. URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf` (visited on 05/06/2017).

[7] Szász János. *The algorithmicx package*. Apr. 7, 2005. URL: `http://mirrors.ctan.org/macros/latex/contrib/algorithmicx/algorithmicx.pdf` (visited on 04/04/2018).

[8] *Lambda Expressions in C++*. July 19, 2017. URL: `https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp` (visited on 05/06/2018).

[9] John Lato. *TQueue can lead to thread starvation*. Sept. 1, 2014. URL: `https://ghc.haskell.org/trac/ghc/ticket/9539` (visited on 04/27/2018).

[10] C.L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". In: JACM. Vol. 20. 1, pp. 46–61.

[11] Simon Peyton Jones. *Beautiful Concurrency.* May 1, 2007. URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/beautiful.pdf` (visited on 04/27/2018).

[12] Joseph Wright, Vedran Miletić, and Till Tantau. *The beamer class. User Guide for Version 3.50.* May 20, 2016. URL: `http://mirrors.ctan.org/macros/latex/contrib/beamer/doc/beameruserguide.pdf` (visited on 04/29/2018).

7 Supplemental Material

The C and C++ sources, taskset inputs, and experimental outputs from the experiments described in this report are included with the report as part of a git repository. Interested parties can use the facilities available via `git log` and other such commands to review the development timeline and progression of the current work in finer detail.