On Compile Time Knuth-Morris-Pratt Precomputation

Justin Kourie[1], Bruce Watson[2,1], and Loek Cleophas[3,1]

[1] FASTAR Research Group, Department of Computer Science, University of Pretoria, 0002 Pretoria, Republic of South Africa
(justin@fastar.org)

[2] FASTAR Research Group, Centre for Knowledge Dynamics and Decision-making, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa
(bruce@fastar.org)

[3] Software Engineering & Technology Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
(loek@fastar.org)

Abstract. Many keyword pattern matching algorithms use precomputation subroutines to produce lookup tables, which in turn are used to improve performance during the search phase. If the keywords to be matched are known at *compile time*, the precomputation subroutines can be implemented to be evaluated at compile time versus at run time. This will provide a performance boost to run time operations. We have started an investigation into the use of metaprogramming techniques to implement such compile time evaluation, initially for the Knuth-Morris-Pratt (KMP) algorithm. We present an initial experimental comparison of the performance of the traditional KMP algorithm to that of an optimised version that uses compile time precomputation. During implementation and benchmarking, it was discovered that C++ is not well suited to metaprogramming when dealing with strings, while the related D language is. We therefore ported our implementation to the latter and performed the benchmarking with that version. We discuss the design of the benchmarks, the experience in implementing the benchmarks in C++ and D, and the results of the D benchmarks. The results show that under certain circumstances, the use of compile time precomputation may significantly improve performance of the KMP algorithm.

Keywords: Knuth-Morris-Pratt algorithm, compile time precomputation, metaprogramming, string processing time

1 Introduction

Keyword pattern matching is a mature field in computing science which has produced a large number of efficient keyword matching algorithms [4,10,7]. Such algorithms play a central role in a wide range of research domains such as molecular biology, information retrieval, pattern recognition, compiling, data compression, program analysis and security [8].

Taxonomies of keyword pattern matching algorithms as well as the SPARE Parts and SPARE Time toolkits implementing these taxonomies have been described in [13,2,3]. One of the benefits provided by these taxonomies is that they reveal commonalities between the algorithms and group them accordingly in the overall taxonomy. Of particular interest to this research are the common precomputation subroutines shared by various pattern matching algorithms.

Generally speaking, such precomputation subroutines take as input the keywords to be matched by the primary algorithms and produce *lookup tables* as output. The lookup tables are then used by the primary matching algorithms when a mismatch between the target keywords and text occurs to proceed more intelligently than primary algorithms not using such precomputed lookup tables. The precomputation subroutines which create the lookup tables are evaluated at *run time*. If however, the keywords to be matched are known at *compile time*, the precomputation subroutines can be implemented to be evaluated at compile time versus at run time. This will provide a performance boost to run time operations. Such compile time evaluation can be achieved using techniques such as metaprogramming or partial evaluation, depending on the implementation language being used.

We have initiated a research endeavour to investigate the application of metaprogramming in implementing such precomputation algorithms. In doing this, the magnitude of performance gains as well as the challenges and drawbacks to the metaprogramming approach will be explored. As a starting point of a broader investigation, this paper considers the classical Knuth-Morris-Pratt (KMP) pattern matching algorithm [6,13] as a case study for an experimental initial benchmark. The objective of our experiments was to investigate whether compile time evaluation of precomputation KMP subroutines could be profitable to KMP keyword pattern matching.

Experimentation is clearly a suitable approach to employ in pursuing this objective. As such, we constructed an experimental benchmark based on the KMP algorithm, to provide the data required to analyse both the advantages and disadvantages of compile time precomputation subroutines. In implementing the two variants of the algorithm to be benchmarked, we initially chose C++ as an implementation language, based on its support for metaprogramming as well as our previous experience in developing SPARE Parts [14] and SPARE Time [2] (both having been implemented using this language). However, our initial experiments showed that C++ does not have the compile time string processing capacity required to implement the type of benchmarks we had in mind for the research.

As a result, we opted to port our implementation to the related language, D. The resulting benchmark in D compares the performance of the traditional KMP algorithm with that of an optimised version which performs the precomputation of its lookup table at compile time.

It should be noted that our primary motivation at this stage is not a desire for massive performance gains. Rather, we focus on understanding the practical requirements involved in optimising a traditional pattern matching algorithm at compile time. As side effects of this focus, we find some interesting scenarios where such optimisations can be justified.

Furthermore, we contrast our initial implementation in C++ with our latter D implementation where appropriate, not as a language debate, but rather to draw attention to how important it is to use the right tool for the job in implementing the type of pattern matching optimisation our research is dealing with.

1.1 Overview

We present some basic definitions in Section 2. Thereafter, an overview of the experiment's design is given in Section 3 before briefly discussing some of the issues encountered during implementation in Section 4. Section 5 presents some hypotheses, the results of our experiments, and an analysis of both. Finally, Section 6 presents concluding remarks and ideas for future work.

2 Basic Definitions

Notational conventions used are as follows. Array subscripts are assumed to be 0-based. A subarray of array A over the range $[i, j)$ is denoted by $A_{[i..j)}$. Textual input is taken from some alphabet Σ. The text used is denoted by $x \in \Sigma^+$. A set of keywords, $K \subseteq \Sigma^+$ is also used, such that $\forall k \in K : (|k| \leq |x|)$.

2.1 Algorithmic Computations

For any algorithm a, we denote by $T(a)$ the time measured in milliseconds which it takes a to complete its execution.

In essence, the primary search algorithm of Knuth-Morris-Pratt uses a precomputed lookup table when a mismatch between the target keyword and the text occurs. This allows forward shifts of more than one position in the text to occur, hence leading to more efficient matching than in a naive algorithm. The KMP algorithm's precomputation function take as input the keywords to be matched by the primary algorithms and produce the lookup tables as output. We assume the reader to be familiar with the details of the primary and precomputation algorithms, and do not present them in detail here. Rather, we assume the following:

Precomputation KMP_{pre} denotes the precomputation function, mapping a keyword $k \in K$ to a lookup table LT_k for keyword k. KMP_{pre} defines the function at the heart of the benchmark. Not only is it timed individually for analysis, it also is used by both the run time and compile time variants of the KMP algorithm. Descriptions of KMP_{pre} and LT_k can be found in e.g. [15,6].

Main Search $KMP_{main}(LT, k, x, CB)$ is the main search procedure implementing the KMP algorithm; a procedure which searches for keyword k in text x aided by LT_k and, if a match of keyword k occurs at x_i, evaluates function $CB(i)$ to determine how to proceed. In this variant, the procedure yields control flow to some *callback function* CB whenever a match occurs—passing the index of the matched keyword to CB. CB then performs some custom operations specific to the particular CB received as input. If CB returns **true** after having completed its operations, KMP_{main} resumes its search from where it left off. If however, CB returns **false** the search is aborted.[1]

Traditional KMP Algorithm $KMP_{trad}(k, x, CB)$ defines a procedure for the traditional KMP pattern matching algorithm, which constructs LT_k at run time and then executes the main search. This defines the traditional KMP algorithm to be benchmarked against its optimised counterpart.

Optimised KMP Algorithm For each $k \in K$, procedure $KMP^k_{opt}(x, CB)$ which can search only for k in x but for which LT_k is predefined. This defines the optimised KMP pattern matching algorithm, which precomputes its lookup table for some $k \in K$ at compile time using metaprogramming.

3 Designing the Benchmark

Having defined its constituent terms, the benchmark's design can now be described. In doing so a simplified data flow diagram is described to give an overview of the

[1] Note that because matches never occur in our benchmarking context (as no $k \in K$ occurs in x), CB is never actually evaluated. As the former is not the case in typical KMP usage, we nevertheless include the function here. Many practical implementations of pattern matching algorithms, including the ones in SPARE Parts [14] and SPARE Time [2], use a callback function.

benchmark's process flow. This approach makes the description more concise whilst distancing itself from implementation specific details. The design does however, assume a programming paradigm which will allow for KMP_{pre} to be evaluated at compile time—as this is the fundamental optimisation being investigated.

Before describing the data flow diagram, several further definitions are required. They have been defined here due to their lower-level nature and direct relevance to the benchmark's data flow.

This benchmark is unorthodox in that it requires a highly generic approach to its compilation process. Specifically, in order to analyse a wide range of different output data, it must be possible to change the values of all $k \in K$ arbitrarily at compile time. The design therefore incorporates a *seed string* or *seed* s, not occurring in text x, to serve as variable input to the compilation process itself.[2] The seed acts as a catalyst in determining the generation of K, as will be discussed shortly.

Definition 1 (Program Code) *Let PC be the benchmark's program code after all metaprogramming has been evaluated. As such, KMP^{k}_{opt} for $k \in K$ as well as the timed instructions necessary to construct the output data Ω (see below) are defined in PC.*

PC can be seen as an intermediary artefact consisting of the code defined by the programmer and the code generated by the compiler after all metaprogramming code has been evaluated.

Definition 2 (Benchmark Binary) *B is defined as the fully compiled binary representation of PC.*

Whereas PC is an intermediary artefact, B on the other hand is a fully compiled program which is ready to be executed.

The set of output data generated by execution of our benchmark B, called Ω, consists of three parts:

- *Precomputation timing data*, pairing a given $k \in K$ and the time taken to compute LT_k. We denote the bag (multiset) of such pairs by P_Ω.
- *Traditional KMP search timing data*, pairing the length of a given $k \in K$ and the time taken to compute $KMP_{trad}(k,\ x,\ CB)$ (where x and CB are assumed to be fixed for the entire benchmark). We denote the multiset of such pairs by T_Ω.
- *Optimised KMP search timing data*, pairing the length of a given $k \in K$ and the time taken to compute $KMP^{k}_{opt}(x,\ CB)$ (where x and CB are again assumed to be fixed). We denote the multiset of such pairs by O_Ω.

Note 3 (Shifts by One) *It is important to note that in our benchmarks, we are interested in determining the differences between the running times of the traditional KMP algorithm and its variant for which precomputation has been performed at compile time. For both variants, the same keyword set K and text x are used, and hence the same shifts are used in both cases and the KMP* search *time will not differ among the two. We are therefore not concerned with whether the particular benchmark keyword set K and text x guarantee a certain behaviour of the Knuth-Morris-Pratt search algorithm per se, e.g. worst-case or average case behavior. To have consistent performance, we opted to always use a seed string s and text x such that the (sub-)alphabet whose characters occur in s and that whose characters occur in x are disjoint. As a*

[2] This can be achieved for example by using a compiler directive.

consequence of this choice, a mismatch will always occur on the first comparison in the KMP search algorithm, and the shift applied in the main text will always equal 1. *As noted above, the actual choice of text, keyword set and shifts applied does not matter, as long as the algorithms are compared on the same combination of text, keyword set and shifts.*

Definition 4 (Benchmark Pipeline) *Let BP denote the data flow and state transitions in the benchmark. This "benchmark pipeline" (depicted in Figure 1) operates as follows:*

- *The first compilation state, C_1, receives seed string s as input and generates keyword set K of size $n = |s|$ as output, such that:*

$$k_1 = s_{[0..1)},\ k_2 = s_{[0..2)},\ \ldots,\ k_n = s_{[0..|s|)}$$

- *The second compilation state, C_2, receives keyword set K as input and then evaluates all metaprogramming before generating PC as output.*
- *The third compilation state, C_3, receives PC as input and compiles benchmark binary B as output.*
- *The benchmark is then run in the last state, R. After loading x into memory as input and executing its timing instructions, B yields Ω for analysis.*

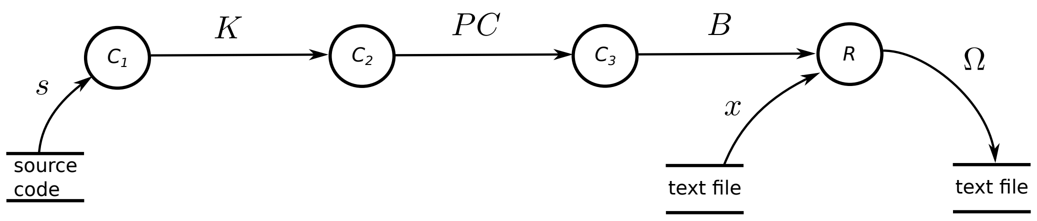


Figure 1: The Benchmark Pipeline

Observe that the text x used in the benchmark may be varied over runs, and that the benchmark can also be recompiled for different values of seed s. These two observations essentially provide the desired flexibility required to generate a wide range of data for analysis.

4 Implementing the Benchmark

The benchmark was initially implemented in C++ in order to extend the SPARE Parts toolkit [13] to include a compile time optimised KMP search. In this implementation, Boost's Meta Programming Library (MPL) [12] was used to support compile time string operations. Unfortunately, it turned out that even with the use of this library, C++ metaprogramming proved to be unsuitable. The benchmark was then ported to D and the resulting implementation was used to perform the benchmarking instead [5]. As the design and class structure of the D and the C++ implementations hardly differ, we do not present that of the C++ implementation here, but only discuss the problems with that implementation, the choice for D over C++, and the design of the D implementation.

4.1 Problems with C++ implementation

Despite being both powerful and flexible, C++ metaprogramming has never supported compile time string operations out of the box. Though excellent supporting libraries such as Boost's MPL enable this ability, its intrinsically constrained nature is the primary reason for abandoning the C++ effort.

Table 1 below summarises the problems encountered with the C++ implementation of the benchmark. As can be seen, four out of the five problems relate to compile time string operations—a feature not provided by and completely unsuited to the design of C++. Out of all the factors listed, the huge performance issues with large strings proved to be the turning point in the implementation effort. After precompiling headers to save overhead, expanding the system's virtual memory, increasing the kernel's default memory map allocation for processes and one too many heap exhaustions—it became obvious that another language should be pursued.

Problem	**Description**
Constrained string length	Length restrictions, due to performance limitations of the Boost MPL, fundamentally constrain $\lvert K\rvert$, which means that a thorough analysis of Ω cannot be done.
Intrinsic string overhead	The overhead required just to declare MPL strings is relatively high due to the complex hackery which enables the feature.
Maximum string overhead	Changing the maximum string length (defaulting to 32) to be above 128 characters results in critical compiler overhead. When setting the length to be greater than around 228 characters, heap exhaustion was repeatedly experienced.
Poor string writability	MPL string syntax makes it tedious to change s and recompile the benchmark with different input.
Array initialisation	Initialising an array with variant compile time data values is a fundamentally tricky problem which either requires potentially exponential use of the preprocessor, or language features which are not part of the current C++ standard.

Table 1: Summary of C++ Implementation Issues

4.2 Choice for D implementation

We selected the D language [5] for implementation following our experience with the C++ implementation. D was designed and originally implemented by Walter Bright and belongs to the family of C/C++/Objective-C. The main intent behind its design was to improve on C++ by being a cleaner and smaller language. D offers powerful language features which address all the issues and challenges we encountered using C++, while its similarity to C++ made our benchmark easy to port.

As D fully supports both object orientation and templates, just like C++ does, the design and object model used for the C++ implementation could be reused without modification. The port of the code itself turned out to be trivial—with any major differences between the two languages actually making the implementation easier than before. For example, D's `foreach` construct and auto type inferencing remove the need for a lot of boiler plate code in loops and type declarations.

The most striking justification for using D over C++ however, is the complete absence of the obstacles encountered with metaprogramming in C++. Firstly, D provides native support for compile time string operations. This means that no constraints on string length are placed outside of the standard system limits, and that writability is no longer an issue either. Secondly, and more importantly, D offers two approaches to metaprogramming:

Template Metaprogramming Template metaprogramming in D employs many techniques similar to those used in C++ (e.g., repetition and selection can be affected through template specialisation) but is far more powerful. Templates can serve as generic namespaces and support a much broader range of template parameters than their C++ counterparts. In addition, a compile time selection statement can be used instead of template specialisation, which avoids considerable overhead.

Compile Time Function Evaluation (CTFE) D's CTFE feature on the other hand, embeds an interpreter in the compilation environment and allows the programmer to direct it to evaluate ordinary functions at compile time. The feature does require that functions meet certain constraints in order to be evaluated, but the constraints are liberal enough to allow for KMP_{pre} to be implemented as a normal, imperative function. No template metaprogramming is required.

4.3 Implementation Overview

The architecture of the benchmark illustrates how the process states defined by *BP* (see Definition 4) are realised.

As presented here, the architecture is fairly technology neutral, and only assumes support for objects and compile time metaprogramming. The object model itself is straightforward and easy to understand as illustrated in Figure 2. The D benchmark uses a slightly simplified design: as indicated before, it uses D's Compile-Time Function Evaluation. As a consequence, the use of a specific C++ compile time precomputation implementation—`CT_KMP_pre`—is replaced by the *compile time use* of the traditional run time precomputation implementation—`RT_KMP_pre`.

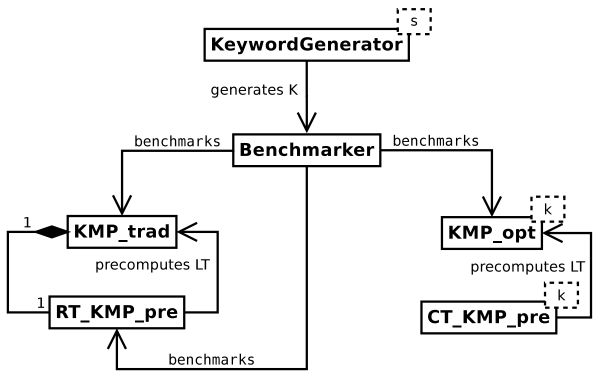


Figure 2: Benchmark Reference Model

The object model essentially consists of three parts, corresponding to control functionality, functionality for the traditional KMP search, and functionality for the optimised KMP search.

Control: This is implemented by classes `KeywordGenerator` and `Benchmarker`. Class `KeywordGenerator` is a template class which wraps the metafunctions[3] used to gen-

[3] "Metafunctions" here refer simply to the techniques employed to operate on data at compile time. For a more detailed definition, and an excellent discussion on what this means in the context of C++, see Chapters 1 and 2 of [1].

erate K from the compile time string s. `KeywordGenerator` thus implements the process defined in state C_1 of BP.

`Benchmarker` implements the control logic which ties the various aspects of the benchmark together. It also implements the timing and output code required. After obtaining K from `KeywordGenerator` at compile time, `Benchmarker` uses it to generate the code and evaluate the metafunctions required to produce PC (see Definition 1). `Benchmarker` thus implements the process defined in state C_2 of BP.

Furthermore, `Benchmarker` loads x from a file at run time and directly controls how Ω is produced (e.g., whether Ω is written to standard output or to file). `Benchmarker` therefore also implements the processes defined in state R of BP.

Optimised KMP search: `KMP_opt` is a template class used by `Benchmarker` to instantiate KMP^k_{opt} for each $k \in K$. For each given k template parameter the resulting template instantiation uses the compile time evaluation of `RT_KMP_pre` for that specific k (in the case of D) or uses `CT_KMP_pre` (in the case of C++) to generate LT_k. `Benchmarker` then times the search functions of objects instantiated for each generated class and constructs O_Ω as a result.

`CT_KMP_pre` is a template class which generates LT_k from a compile time string k. In the C++ implementation, the class is used as a delegate by `KMP_opt` in compiling LT_k into each of `KMP_opt`'s generated classes.

Traditional KMP search: `KMP_trad` is an ordinary class implementing the procedure defined by KMP_{trad}. `Benchmarker` instantiates a `KMP_trad` object for each $k \in K$. In doing so, the creation of each LT_k is delegated to `RT_KMP_pre` at run time. `Benchmarker` times both the delegated request and the ensuing search in order to create T_Ω.

`RT_KMP_pre` implements the run time version of the function defined by KMP_{pre}. `Benchmarker` instantiates the class for each $k \in K$. Each object has a function which returns LT_k for the k it was constructed with. By timing these operations separately, `Benchmarker` creates the remaining dataset P_Ω.

5 Results Analysis and Interpretation

In this section, we discuss the results obtained using the benchmarking. First, we present a number of hypotheses to guide the results analysis, as well as an overview of the benchmarking platform used. We then present the results together with our analysis and interpretation of them.

5.1 Hypotheses

The benchmark is designed to output data such that the relationship between the time, t, taken to search for a keyword k, and the keyword's length $|k|$ can be examined. In order to direct the analysis several hypotheses are proposed. Two of the hypotheses (called S_1 and S_2 below) act as sanity tests to verify the correct functioning of the benchmark. The other two (called P_1 and P_2 below) are postulated in the hope that more understanding can be gained as to when exactly compile time optimisation can prove useful to keyword pattern matching. Figure 3 helps conceptualise the hypotheses which follow in presenting a hypothetical plot of the datasets contained in Ω.

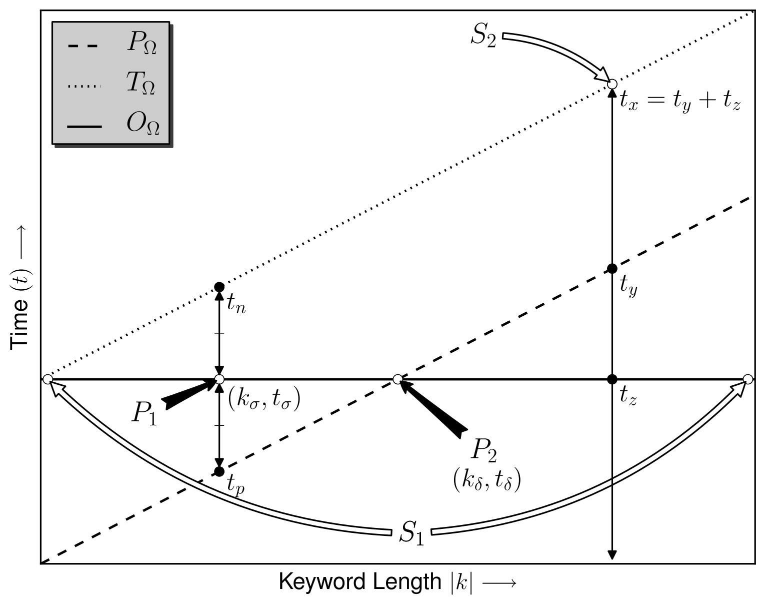


Figure 3: Hypothetical Results

Hypothesis 5 (S_1)

$$\forall (k_i, t_i) \in O_\Omega : \Big[\Big(\forall (k_j, t_j) \in O_\Omega : (t_i = t_j) \Big) \wedge \Big(\forall (k_i, t_p) \in T_\Omega : (t_i < t_p) \Big) \Big]$$

All time values in the pairs belonging to the multiset O_Ω are equal. Furthermore, all such time values are less than all time values in the pairs belonging to T_Ω.

Because each KMP^k_{opt} already has LT_k defined by definition, and because the same x and CB are used when measuring all search times; the time taken to perform the optimised KMP search must be constant. As mentioned before, the alphabet used for text x and that used for seed s and hence keywords k_i are disjoint, hence the main search time depends on the length of text x, but not on any of the keywords k_i. In other words, the same (arbitrary case) search is repeated by each KMP^k_{opt}—therefore the times of those searches must be the same.

Furthermore, since KMP_{trad} always computes some LT_k, its total running time for the same values of x and CB must take longer than any such search for which LT_k has been predefined, i.e. using KMP^k_{opt}. This explains the second part of the hypothesis.

Corollary 6 (S_2)

$$\forall (k_i, t_x) \in T_\Omega, (k_i, t_y) \in P_\Omega, (k_i, t_z) \in O_\Omega : \Big(t_x = t_y + t_z \Big)$$

Each time value in each pair belonging to T_Ω is equivalent to the sum of the time values in the corresponding pairs belonging to P_Ω and O_Ω respectively.

Following from the first hypothesis, KMP_{trad} always takes longer than KMP^k_{opt} by the time it takes to compute LT_k for any $k \in K$.[4]

The discussion of S_1 and S_2 above should make it clear that these predicates should hold at all points throughout the benchmark. Any marked deviance from these conditions is a sign that something has gone awry. Of course they cannot be expected to hold perfectly true in practice, due to systemic factors that affect time measurement (e.g., OS scheduling, CPU instruction caching etc.). As a result, *minor* deviances from these sanity tests will be ignored. However, major deviances flag potential implementation problems. In our implementation and benchmarking efforts, such deviances appeared when using the C++ implementations of the traditional and optimised KMP algorithm. This lead us to investigate the suitability of C++ for our comparison, eventually leading to the causes for its insuitability as listed in Section 4.1, and to our abandonment of C++ in favour of D as the implementation language. As the results in the next section will show, no major deviances from the above 'sanity check' predicates were observed with the D implementation.

Hypothesis 7 (P_1)

$$\exists (k_\sigma, t_\sigma) \in O_\Omega, (k_\sigma, t_n) \in T_\Omega, (k_\sigma, t_p) \in P_\Omega : \Big(t_n - t_\sigma = t_\sigma - t_p\Big)$$

A $k_\sigma \in K$ exists such that $T(KMP^{k_\sigma}_{opt}(x,\ CB))$ is faster than $T(KMP_{trad}(k_\sigma,\ x,\ CB))$ by the same amount as it is slower than $T(KMP_{pre}(k_\sigma))$.

Though an optimised search is always faster than a traditional search for the same keyword in the same text, the question that lingers is "when does the dividend gained really start to matter?". A heuristic is introduced here to try and answer that question.

The keyword length for which the traditional search is exactly equal to the precomputation time plus optimal search time represents an interesting boundary value. It is shown as P_1 in Figure 3. Note that at this point, precomputation time is exactly half of the optimised search time, so that the traditional search time is equal to 1.5 times the optimised search time. The definition below is one way of expressing the relationship between these respective search- and precomputation times.

Definition 8 (Beneficial Heuristic (Benheur) Equation)

$$t_\sigma = \frac{t_n + t_p}{2}$$

Let t_σ be a heuristic aid in determining when optimisation may be useful where:

- *t_σ is the time taken by an optimised search for a given $k \in K$.*
- *t_n is the time taken by a traditional search for $k \in K$.*
- *t_p is the time difference between t_n and t_σ.*

Hypothesis 9 (P_2)

$$\exists (k_\delta, t_\delta) \in O_\Omega, (k_\delta, t_i) \in P_\Omega : (t_\delta = t_i)$$

There exists some $k_\delta \in K$ such that $T(KMP^{k_\delta}_{opt}(x,\ CB)) = T(KMP_{pre}(k))$.

[4] Note that this assumes an implementation language that is as efficient in its compile time code execution as in its run time code execution, which may not always be the case for a language such as C++.

If it is taken as a given that a keyword being searched for is known at compile time, and it is shown that an optimised search can be completed before the traditional search even begins; there arises a strong argument *against* using the traditional search. The following definition is made for completeness:

Definition 10 (Delta Point)

$$t_\delta = t_i$$

Let t_δ be the delta point where compile time optimisation becomes preferable to traditional searching for known keywords where:

- *t_δ is the time taken by an optimised search for a given $k \in K$.*
- *t_i is the difference between the traditional search and the optimised search.*

5.2 Benchmarking Platform

Table 2 summarises the details of the benchmarking platform. All benchmarking was run in a minimal environment with only essential services running. Furthermore, one of the CPU cores was allocated to run the benchmark's system process in isolation, with the rest of the processes guaranteed to execute only on the other core. This is easily achieved using the `taskset` [11] utility. Another utility, `schedtool` [9] was used to reassign a FIFO scheduler policy to the benchmark's process. By disabling pre-emptive scheduling for the process, much more representative sample data could be obtained due to minimal extraprocess interference.

Architecture:	Intel Core2 6400 @2.13 GHz
Operating System:	Linux 2.6.34 (Gentoo sources release 6)
RAM:	2 GB
C++ Compiler:	GCC 4.5.1 (Gentoo patches 1.1)
D Compiler:	Digital Mars dmd 2.0

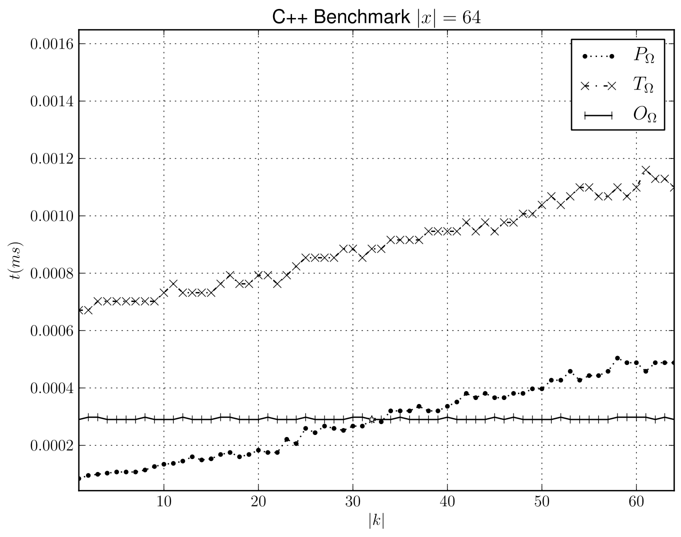
Table 2: Platform Specification

5.3 Results Interpretation

As seen in Figure 4, the C++ benchmark performed substantially faster than the D benchmark. This may be due (in part) to the decision to reduce the D compiler's optimisation level (which was interfering with the sample data's consistency). The C++ results however, are not even in the order of the bounds defined by the sanity check implied by S_2. Due to the very small inputs being benchmarked, it is suspected that language implementation factors (e.g., the time of object construction) were responsible for disproportionally tainting the output dataset. Due to the limited scope of the C++ results, further investigation was deemed unnecessary.

Interestingly, the data we analysed contained a point where the heuristic given in Definition 8 holds precisely. This is shown by the three squares in Figure 5a (page 27). The delta point mentioned in Definition 10 is seen also to occur in the order of where it was predicted. Though such relations remained approximately constant throughout our results, the results are not generalisable due to the arbitrary case analysis of the benchmark.

It was also noted during our analysis that the ratio between $|x|$ and $|k|$ has some practical implications. Take for example $|x|$ and $|k|$ in Figure 5a around the point of



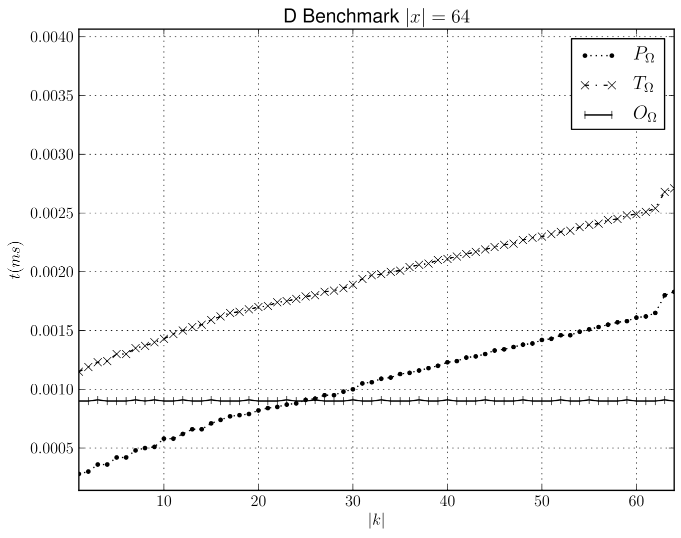
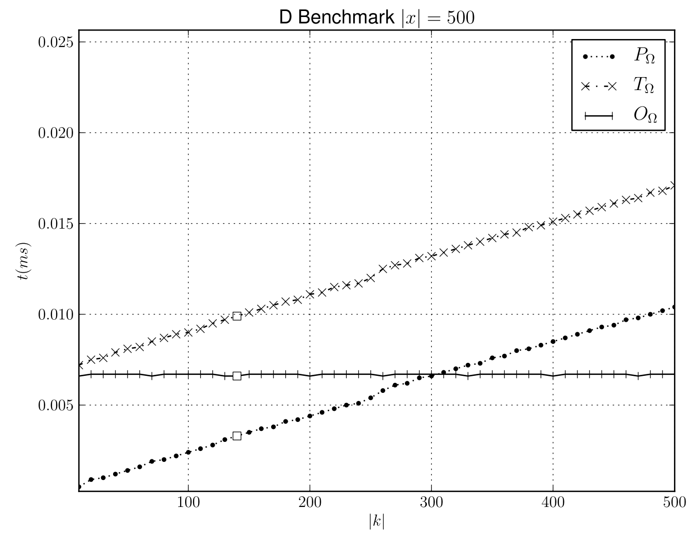


Figure 4: C++ vs D Results

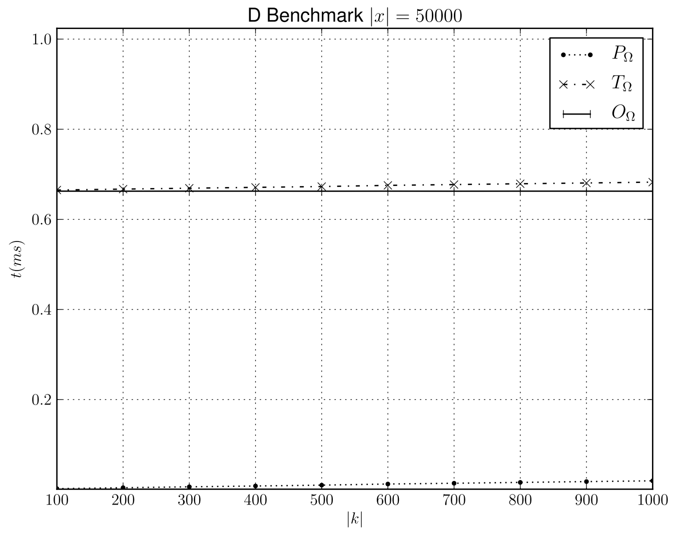
the heuristic match. Given that KMP_{opt}^{k} is arguably of practical interest at this point, we note that $|k| \approx \frac{|x|}{4}$. As no *smaller* ratios of $|k|$ to $|x|$ showed a favourable case for KMP_{opt}^{k} in our analysis, we note that the strong cases for the use of metaprogramming occurred where $|k| \geq \approx \frac{|x|}{4}$. Our analysis also noted that where $|k| \leq \approx \frac{|x|}{50}$, the difference between the KMP_{opt}^{k} and KMP_{trad} becomes so small as to be practically negligible (as seen in Figure 5b).

6 Concluding Remarks

The benchmarking of the optimised and traditional KMP algorithms, although based on a particular case analysis in which no keyword matches occur, has lead to interesting results. Firstly, it was seen that performance gains are most significant when the proportional difference in size between the search text and the keyword is small.



(a) Heuristic Match



(b) Minimal Gains

Figure 5: Cases For and Against Optimisation

This suggests that compile time optimisation may prove practical where this is the case, such as in intrusion detection systems.

Similarly, it was seen that performance gains become redundant when the proportional difference between search text and keyword size is very large. Therefore compile time optimisation appears to be less applicable in domains where such relations are typical, for example DNA pattern matching.

The clear design of the research objectives, implementation structure, and hypotheses are seen to form the basis of a good benchmark design. Such definitions also reinforce the repeatability and correctness of the experiment.

Furthermore, succinct information has been provided on the limitations of C++ regarding its suitability for optimised keyword pattern matching. C++ metaprogramming is shown to be fundamentally unsuited for even moderately demanding compile time string operations. In retrospect this is not very surprising. Template metaprogramming (let alone template *string* metaprogramming) is simply not a feature C++ was designed for. Indeed, any "feature" that is not explicitly designed from the ground up, remains—by definition—in the realm of craft.

This contrasts strongly with the solid engineering behind the D programming language. Though the D benchmarks are of limited use as far as scientific observation is concerned, they serve as a very good proof of concept. D is seen to provide highly robust metaprogramming support—exactly what is required for computationally demanding compile time optimisations. Its native support for compile time string operations as well as its Compile Time Function Execution feature, are only two of the reasons that made implementing the benchmark in D a suitable decision.

6.1 Further Research

This work presents a base for several areas of further research, including the following topics:

- The benchmark model described in Section 3 can be refined and extended to support the analysis of many of the algorithms identified in [13] and [2]. Such an effort would use the terms and definitions from those works in order to synthesise more consistently with the taxonomies they describe.
- The metaprogramming features offered by other languages such as LISP and Haskell can be investigated for possible applications in keyword pattern matching. Again, by following in the example of [13] and [2], an optimised toolkit could be constructed to supplement SPARE Parts and SPARE Time.
- Research investigating the performance gains of applying compile time optimisation to pattern matching in real world systems would prove interesting. In particular, it would be advisable to consider applications where the proportional difference between the search text size and keyword size is small.
- When the next C++ standard is published, it would be interesting to evaluate the language's overall suitability to metaprogramming. Such a review could make use of comparisons to other programming languages with an emphasis on qualitative software engineering "ilities" (e.g., maintainability, modifiability, scalability etc.).

Acknowledgements

We thank the anonymous referees for their feedback, and thank Linda Marshall and Derrick Kourie for their contributions to this research and its presentation.

References

1. D. Abrahams: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2005.
2. L. Cleophas: *Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms*, master's thesis, Eindhoven University of Technology, 2003.
3. L. Cleophas, B. W. Watson, and G. Zwaan: *A New Taxonomy of Sublinear Right-To-Left Scanning Keyword Pattern Matching Algorithms.* Sci. Comput. Program., 75 November 2010, pp. 1095–1112.
4. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, 2007.
5. *D Programming Language 2.0*: `http://www.digitalmars.com/d/2.0/index.html`.
6. D. E. Knuth, J. Morris, and V. R. Pratt: *Fast Pattern Matching in Strings.* SIAM Journal on Computing, 6(2) June 1977, pp. 323–350.
7. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
8. *Pattern Matching Pointers*: `http://www.cs.ucr.edu/~stelo/pattern.html`.
9. *schedtool(8) – Linux man page*: `http://linux.die.net/man/8/schedtool`.
10. W. Smyth: *Computing Patterns in Strings*, Addison-Wesley, 2003.
11. *taskset(1) – Linux man page*: `http://linux.die.net/man/1/taskset`.
12. *The Boost MPL Library*: version 1.43.0, `http://www.boost.org/doc/libs/1_43_0/libs/mpl/doc/index.html`.
13. B. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, 1995.
14. B. W. Watson and L. Cleophas: *SPARE Parts: A C++ toolkit for String PAttern REcognition.* Software—Practice & Experience, 34(7) June 2004, pp. 697–710.
15. B. W. Watson and G. Zwaan: *A Taxonomy of Keyword Pattern Matching Algorithms*, Computing Science Report 92/27, Technische Universiteit Eindhoven, 1992.