Independent thesis, 15 HE credits, for degree of Bachelor in Computer Science
Spring Term 2016

# A Performance Comparison of Object Oriented Programming Languages

## Comparing Java, C# and C++

Christopher Sexton

School of Health and Society

**Author**
Christopher Sexton

**Title**
A Performance Comparison of Object Oriented Languages, Comparing Java, C# and C++

**Supervisor**
Andreas Nilsson

**Examiner**
Name of examiner?

**Abstract**
This study investigates the relative performance characteristics of three market leading object-oriented languages, Java, C# and C++. A comprehensive literature review is undertaken to establish the theoretical groundwork and state of existing scholarship on the subject. Then a case study is performed in which a suite of benchmarks implemented in each of the three languages are profiled for key performance criteria including execution time and memory footprint. The initial results are analyzed and interesting patterns and anomalies investigated in a series of follow up investigations. In line with expectations, C++ implementations prove generally faster and more compact than Java and C# counterparts, but this is shown to be dependent on considerable tuning and compile time optimization. Runtime optimizations performed by the JIT compilers of the respective virtual machines are capable of giving Java and C# an advantage over C++ in some contexts, but automatic memory management is a time-space tradeoff and performance can suffer when memory is limited.

**Keywords**
Performance, object-oriented, benchmarking, time-space tradeoff, Java, C#, C++

## Table of Contents

# 1 Introduction

This section introduces the study by first providing a brief background context to its subject matter, then describing the precise aim and purpose of the work, its motivation in relation to social and ethical concerns, its limitations, and its guiding research questions.

## 1.1 Background

Since the appearance of the first major programming language, FORTRAN[1], in 1957, the number of languages has blossomed in conjunction with the rapid development and proliferation of hardware and software systems. These developments in turn fostered the growth in programming as both a hobby and as a profession integrated into the holistic field of software engineering[2]. The multitude of available programming languages, which continues to grow, presents these programmers with a question that has faced engineers and makers throughout human history: which tool to use for the job?

In some contexts the answer will be clear enough since certain problem domains have unique requirements or restrictions that specific solutions are best placed to meet. For example, C for programs close to the hardware, R for statistical analysis and Ada for US government defence systems and avionics. But when a programming language is suitable for several uses, or when several languages are viable alternatives for a particular job, the question of which language to choose moves into focus. Differing opinions on this, together with the rise of the internet as a way for programmers to share code and points of view, has helped to fuel so-called “language wars”: the ongoing discussion, at times heated, in the software development community about which language is best[3].

In this context there is a need for data and carefully weighed empirical evidence that can help to cut through the mass of opinion, conjecture and partisanship that clouds the issue. Especially so since, in an increasingly networked and digitized age, the choice of programming language is not just a matter of academic debate on hacker discussion groups. It is a real one that must be made by public and private sector managers, scientists, researchers and even militaries every day. It is thus a choice that may have real consequences for companies, for governments, for workers and thereby for many real lives.

## 1.2 Aim and Purpose

The aim of this work is to compare the performance of the three object oriented programming languages, Java, C# and C++.

### *1.2.1 Why these languages?*

There are several reasons for selecting this group of languages. One is their prevalence both in industry and academia. In 2016 the popular job-search site Indeed reported Java, C# and C++ to still be the first, second and third most frequently mentioned languages in its job adverts[4], a decade on from the perceived peak of the Object Oriented “trend”. Partly because of this, as well as their benefits as instructional languages, the Object Oriented languages continue to be used in many Computer Science-related educational programmes, including the Bachelor Programme in Software Development at Högskolan Kristianstad. This in turn helps to ensure that the popularity of the languages will be sustained.

Another reason is the existence of significant overlap between the three languages in their potential applicability, making them competitors or alternative solutions to one another in a range of possible scenarios. While Java and C# might be said to have more in common

with each other in terms of design and typical usage scenarios than either has with C++, all three languages are viable alternative solutions for a range of real-world problems where performance matters. These include scientific simulations, medical imaging and bioinformatics, high frequency trading (HFT) and data processing at scale[5]. Finally, all three languages are C-type languages with syntactical and idiomatic similarities which mean a programmer with experience in one will generally find the source code of the others intelligible.

### *1.2.2 Why Performance?*

In comparing and choosing between languages there are many factors that may be considered. Performance, which may itself be defined according to several different criteria, is just one. Others include security, stability, interoperability, cost, availability of specialist libraries, and portability (a key selling point for Java with its "write once, run anywhere" philosophy). Performance does not matter in all contexts but it continues to be very important in some. In those contexts where it matters, performance may be a factor with the power to impact human lives and larger social goals as detailed in section 1.3 (Motivation) below. Furthermore, performance is frequently a central focus in "language war" debates about the relative merits of different languages, as well as being something that lends itself to being measured quantitively.

### *1.2.3 A "fair" Comparison?*

It may be objected that an attempt to compare performance of interpreted with compiled languages is meaningless because this fundamental difference precludes the possibility of a "fair" comparison. This objection contains the assumption that only like-for-like comparisons, from an equal starting position, have the possibility to be useful or illuminating. If this were true, it would preclude any kind of performance comparison between programming languages, which all differ from one another in some fundamental ways. A level playing field between programming languages can never be attained. Much of the intellectual value in a comparative study, therefore, will come from identifying the ways in which the playing field is not level. By identifying which languages are expected to have an advantage in certain situations, and why, and then verifying these expectations against experimental data, we go a large way towards answering the research questions.

That one language is fundamentally likely to be faster than another does not eliminate interesting questions. Indeed, it tends to raise many, such as:

- Can the performance gap be closed at all and if so, how?
- Does the relationship change in different environments?
- Is it dependent on some condition?

## 1.3 Motivation

The context of Moore's Law[6], which predicts exponential growth in the number of transistors on integrated circuits over time, and the consequent gains in computing power over the last several decades, may tempt some to conclude that performance is no longer a priority. Such assertions may be valid in some contexts, such as personal computing. However, there remain many important computing contexts in which performance may be a crucial consideration attached to significant social and ethical consequences. Examples of such contexts include:

- Computing operations that are performed at scale: eg. Large, distributed web applications with many users, large-scale finance operations such as High Frequency Trading (HFT), automated swaptions, Big Data analysis and processing
- Operations that are inherently iterative or recursive, eg. Scientific analysis and modelling in physics and bioinformatics
- Operations where execution time of an individual function is critical, eg. Automated trading, accurate real-time positioning/navigation data, responsive avionics
- Computing contexts in which hardware resources are limited, such as mobile devices

These kinds of contexts and the social impacts of performance are highlighted in an interesting 2011 conference presentation[5] by Herb Sutter, a former Microsoft engineer and member of the ISO C++ standards committee. Sutter talks about languages in terms of "performance per dollar" and breaks this down further into performance per watt (power use), performance per transistor (size) and performance per cycle. He quotes an independent study by James Hamilton, a Distinguished Engineer at Amazon Web Services, who found that 88% of monthly Data Centre costs are taken up by a combination the server hardware itself and power usage: ie. two factors that are especially performance influenced. Sutter concludes that a two-fold improvement in a program's efficiency in space and time would correspond to a halving of data centre costs. Sutter also quotes the creator of C++, Bjarne Stroustrup, as stating that:

> My contribution to the fight against global warming is C++'s efficiency: Just think if google had to have twice as many server farms! Each uses as much energy as a small town.

In light of this, accurate information about programming language performance can have a positive social impact in a range of conceivable ways. Large private and public sector organizations with systems that have millions of users and perform many operations a day can save on server space and energy costs by optimizing performance. For governments this means saving tax-payer money that can be redirected to other critical resources, while for companies it means opportunities for reinvestment and improvements in their services to customers. And for both it means a potentially smaller energy footprint and a contribution to environmental sustainability. In the scientific domain, performance optimization can potentially help physicists run complex simulations with big data sets in less time and with fewer hardware resources. In bioinformatics such computationally expensive operations as DNA analysis and gene sequencing can be made more efficient to speed up diagnostic investigations and lead to faster discovery of novel cures and therapies.

Finally, it may also be reasonably argued that optimization is at the heart of the disciplines of Computer Science and Engineering and that a complacent reliance on system latency is not in keeping with the spirit of the field. An approach that prizes optimization and efficiency as intrinsic goods in themselves is likely to benefit society in many ways in the long-run by producing better programmers.

## 1.4 Limitations

The case study employs a suite of benchmarks to measure the relative performance of the three languages. Benchmarks are small programs (generally not more than a hundred lines of code) whose run-time behaviour, when monitored, can provide useful indications about the relative performance characteristics of hardware, software, the algorithm of the program

itself, or the language it is written in. Benchmarks frequently involve non-trivial mathematical operations such as sorting, searching or recursive functions, or else they may perform a simpler task like method calls or object instantiation for a very large number of iterations. They are often run together with some type of profiling software, which is capable of recording key aspects of the program's runtime behaviour and use of system resources, which can then be analysed afterwards. The data collected will depend on the aim of the benchmarking process, but may include execution time, CPU usage data, and identification of hot spots in the code where the program spends a disproportionate amount of its time. The process can thus help to identify performance bottlenecks specific to an environment, algorithm or language.

Such benchmarking is, however, necessarily limited as an evaluative tool for language performance. A benchmark program of any significant complexity can almost never be implemented in exactly the same way across languages and observed differences in performance may be partly attributable to differences in implementation rather than features of the language. Furthermore, the case study will be conducted on a specific machine with specific hardware specifications running a specific operating system and using a specific profiling tool. In this context, the results may be subject to numerous environment-specific influences including the operating system's strategy for managing resources, the accuracy of the data it supplies about the use of those resources, and the quality and maintenance state of the hardware components.

Finally, most real world deployments of programming languages are in large, complex applications integrated with larger software and hardware environments, performing input/output (I/O) operations at scale, and serving many users. In Computer Architecture: A Quantitative Approach, Hennessy and Patterson write that, "The best choice of benchmark to measure performance is real applications"[7]. Since a microbenchmark cannot reproduce these conditions, its results cannot be taken as a definitive statement about the performances of languages in many potential real-world scenarios. However, it is worth noting that benchmarking can highlight performance bottlenecks specific to a particular language or program whose effects may be felt proportionally at much larger scale.

## 1.5 Research Questions

The following are the guiding research questions for this study. the primary research question framing the work will be: *What are the relative performance characteristics of Java, C++ and C#?* The question necessarily entails a number of smaller sub-questions and issues that must be addressed, for example:

- What is meant by "performance" in this context? What criteria can be used to measure it?
- To what extent is it possible to design a fair comparison of different languages and what potential confounding factors are in play?
- What are the limits and conditions of any performance differences that are observed: Do the performance differences depend on some factor(s) and if so, which?
- What are the particular performance bottlenecks or advantages specific to each of the three languages under study, and why?

# 2 Method

Two main investigative methodologies, one theoretical and one practical, will be applied to the answering of the research questions. The details of each method and the reasons for using it are discussed here.

## 2.1 Literature Review

The first stage of the investigation consists of a comprehensive search of the existing literature related to programming language performance, optimization, and the history and characteristics of the three specific Object Oriented languages under consideration, C++, C# and Java. The aim of such a literature review is to attain a theoretical overview of what is already known about the relative performance of the three languages, and about language performance and benchmarking generally, and to use this knowledge to frame the expectations of the practical case study. The scope of the literature review encompasses three main categories of sources.

### *2.1.1 Peer-reviewed scholarship*

Peer-reviewed scientific articles discovered with the help of Högskolan Kristianstad's scholarly search tool, Summon, and the popular online research hubs SpringerLink and ACM Digital Library. Such articles form the academic bedrock of this study's information sources and are especially valuable in providing an overview of previous investigative work that has already been done in the area.

### *2.1.2 Textboks and Technical papers*

Academic textbooks, reference works and technical manuals such as whitepapers from standards authorities and programming language development teams. This type of literature can illuminate the official technical specifications of languages and the environments they operate in. The starting points for discovering such literature are the HKR library services, course literature lists for previous courses in the Bachelor Programme in Software Development, and the websites of language development houses such as Microsoft Developer Network for C# and Oracle for Java.

### *2.1.3 General literature*

A third and final category of sources are general, non-academic resources online that can supply important background information and context. These include developer blogs, conference proceedings, presentation slides and even contributions to discussion forums. While such non-academic sources might generally be considered of dubious quality and authority, the nature of programming languages as practical tools used in an applied discipline like software engineering complicates the issue. In short it means that most of the leading practitioners, developers and authorities for a given language will not be academics based at a university but rather engineers employed by major technology companies and software houses. Indeed, major companies like Microsoft and Google often have headquarters that are referred to as "campuses" and maintain their own well-funded research departments in which performance optimization projects are undertaken. With this in mind, conference presentations and even ancillary, informal writings published online by senior industry developers and architects – especially those with a role on language standards and development committees – can possess a higher degree of authority than such sources otherwise would.

### *2.1.4 Literature search results and selection*

The main research question, "What are the relative performance characteristics of C++, Java and C#?" was broken down into conceptual parts and related key words and synonyms derived from each part. For example:

**Concept:** performance **Related terms:** performance, optimization, benchmark(ing), memory, footprint, time, space, size

By applying such an analysis to every conceivable aspect of the main research question, a list of key search terms for use in this investigation was derived. Figure 1 below presents these terms as a word cloud.


**Figure 1: Word cloud of key words used in the literature search**

These key words were then entered in different combinations, and with different search words ("AND", "OR", etc.) to both Högskolan Kristianstad's scholarly search tool Summon@HKR and to Google. Further filters were then applied to limit results to peer-reviewed work only, for example, or to only articles that were accessible online. The default ordering of results, by "relevance" was tested for its effectiveness by going to the last page of results and confirming that the works listed there were generally less relevant than on the front page. This proved to be the case and so the default ordering was not changed. Table 1 displays the number of hits returned by Summon@HKR for some example combinations of key words with different search techniques and the specific references from the bibliography that were found in each case.

**Table 1: Hits from the summon database for different example searches ("PR" = peer reviewed)**

| Key Words | Search Type | Whole Text | Title Only | Whole Text PR | Title Only PR | Reference Found |
|---|---|---|---|---|---|---|
| benchmarking, java | OR | 339906 | 21529 | 180368 | 8416 | |
| | AND | 4764 | 11 | 2047 | 4 | |
| | "phrase" | 72 | 4 | 28 | 1 | Bull et al [11] |
| compare, language, performance, benchmark | OR | 14556232 | 8647532 | 433886 | 433886 | |
| | AND | 12 | 0 | 1 | 0 | Sarimbekov et al[19] |
| | "phrase" | n/a | n/a | n/a | n/a | |
| performance, memory, garbage | OR | 8930223 | 661309 | 5581536 | 458116 | |
| | AND | 16353 | 12 | 4943 | 2 | Hertz and Berger[12] |
| | "phrase" | n/a | n/a | n/a | n/a | |

A simple heuristic (see Appendix 1) was applied in assessing search results for their usefulness to the project: The title was read first to determine if it contained any of the key words or otherwise sounded potentially relevant to the subject. The abstract was then read

to see if the work was about performance, and specifically about language performance. Many articles pertaining to other aspects of performance, such as hardware or algorithm implementations, were excluded this way. Articles whose abstracts revealed them to be studies of language performance, and which included one or more of this study's three languages, were read beyond the abstract. The peer-review filter was always applied and preference given to sources that met this criterion. However, the peer review standard was not applied as an absolute authority: articles by leading researchers (ie. professors and Phds in Computer Science departments at major institutions) and practitioners (ie. senior engineers at prestigious technology firms) were not automatically excluded because their papers had not undergone the formal peer review process, for reasons already elaborated in 2.1.3 above. A general common sense test for non-peer reviewed work was to use the ACM Digital Library's "referenced by" tool to see if articles had been referenced frequently in credible scientific literature. This was also a good way to find relevant, related literature, ie. by finding one high quality, relevant article and then following the links to its references.

## 2.2 Case Study

The method employed in the case study was to measure the performance of C++, Java and C# implementations of the same core suite of benchmark programs. The results obtained from the initial run of these programs were then used as the basis for a series of follow-up investigations to dig deeper into some of the notable patterns and themes in the data. The rest of this section provides details on the benchmarks used, how and why they were selected, and the experimental process followed.

### *2.2.1 The Computer Language Benchmarks Game*

The Computer Language Benchmarks Game (hereafter CLBG) is an open source, collaborative online project with a suite of community maintained benchmark programs designed for testing the relative performance of different programming languages[8]. It is the successor to previous similar projects originating as the "Programming Language shoot-out" and the present state of the project represents over a decade of refinement and tuning, taking into account contributions and criticisms from the programming community. The CLBG benchmarks are sufficiently rigorous and refined for them to have been used by researchers in peer-reviewed work, including Li et al[9] and Sarimbekov et al[10]. The maintainers of the project are careful to note the limitations of benchmarking as a method on which to base definitive or sweeping judgments about a language, noting that a performance disparity can often be traced to the way a benchmark is implemented rather than the language it is written in. However, the CLBG project has taken as much care as feasible to mitigate this problem by setting community code submission rules that insist on the use of the same algorithm to solve each benchmark regardless of the language, and to reject submissions that use language-specific tricks to "optimize away the work". At the same time, the project forum has a healthy community of expert developers who are continuously scrutinizing existing benchmarks for signs of fundamental unfairness and contributing fixes.

In light of this, it was determined to be unrealistic for this study to attempt to produce original benchmarks across the three languages under study that could compete with the range, rigor and quality of the CLBG programs. This is especially so in the case of C++, where considerable skill is required to competently write a rigorous benchmark. Such an attempt would have consumed a disproportionate amount of time and almost certainly produced a result that was less scientifically reliable than the CLBG suite. It was therefore

decided that the experimental work for this case study would be conducted using a selection of the CLBG programs together with the monitoring tool supplied with the source code, a python script called simply "bencher".

### *2.2.2 Setup and Environment Configuration*

The open source code available from the CLBG consists of the source code files for each benchmark program and a process monitoring and profiling tool written in python, called bencher. In addition to these tools, the GNU Make[22] utility for Windows was installed to simplify the build process by making it possible to edit and save build configurations in Make files rather than having to specify compilation flags on the command line on every test run. The Cygwin utility[23], which makes GNU utilities like the gcc/g++ compiler available in Windows, was also installed.

For each of the CLBG benchmark programs there are normally several different implementations for each language, especially so for popular languages. The various implementations are given a numeric ID so that for example different Java implementations of the nbody benchmark might be called nbody.java.java, nbody.java-2.java, nbody.java-3.java, …. and so on. The specific programs and languages to be tested are specified in an INI configuration file bundled with the bencher script. By amending the properties and values in the INI file, bencher knows which program source code folders to search for source code files in, which language file extensions to look for, which input values to use, how many times to repeat the tests, and so on.

The bencher program in turn uses the system appropriate (Unix or Win32) Makefile to find the instructions for how to build the specified programs. It follows the Makefile instructions to do the requisite file copying and invoking of compilers, outputs the runnable file to a temporary folder, and then executes it as a child process.

### *2.2.3 Defining "performance" and what to measure*

The two primary criteria by which performance will be measured are execution time and memory footprint. This makes sense in terms of the priorities of many real world applications and the cost-saving considerations discussed by Sutter[5], and is in line with the focus of existing literature including Bull et all[10], Hertz and Berger[11], and Hundt[12]. However, there are other related metrics that the CLBG bencher script measures by default that add useful quantitative context. They are CPU load, CPU seconds (which would be expected to be a multiple of execution time on multiprocessor architectures) and source code file size, which is also relevant to space considerations. On Windows systems, the bencher script obtains its readings from the following Win32 API components:

- Elapsed time(s): Time taken before forking child-process and after child-process exits using QueryPerformanceCounter.
- CPU(s):QueryInformationJobObject(hJob, JobObjectBasicAccountingInformation) TotalKernelTime + TotalUserTime
- QueryInformationJobObject(hJob, JobObjectBasicAccountingInformation) PeakJobMemoryUsed
- GetSystemTimes UserTime IdleTime are taken before child-process is forked and after it exists. Percentage shown is proportion of TotalUserTime to UserTime + IdleTime, which is essentially what is shown in the Task Manager utility.

### *2.2.4 The Benchmark Programs*

The CLBG project maintains around fifteen benchmarks in total. In selecting programs to use for this study, the first consideration was the revision history of a program and the length of time it had been part of the project. The newest programs, with relatively fewer implementations contributed from the community, and with ongoing issues reported in the forums, were discounted. Instead, a core list of proven, tried-and-tested programs with many implementations for each language was chosen. This list was narrowed further by running some initial compilation and calibration tests in the case study environment. Programs with too many compilation errors, or where at least one implementation for each of the three languages could not be compiled, were discounted. Another consideration was to try to choose benchmarks that highlighted different aspects of performance. For example, some benchmarks focus on computational speed and sort or update elements in a small, fixed array, while others do a lot of dynamic memory allocation. The final eight chosen benchmarks and a brief description of what they do are given below.

*Binarytrees*

An adaptation of a benchmark originally developed to test a garbage collection utility developed for C and C++. The program focuses on memory allocation by creating and traversing a series of perfect binary trees.

*Fannkuchredux*

Adaptation of the "fannkuch" or pancake-sorting problem where unsorted numbers in a set are flipped in smaller subsets defined by the value of the term at position 1.

*Fasta*

FASTA is a text format used in bioinformatics to represent nucleotide sequences that make up DNA molecules. This program generates and writes random DNA sequences and outputs to a text file in FASTA format.

*Knucleotide*

This program uses a FASTA file as input and parses it looking for DNA sequences which it then maintains a count of in a hash table, which progressively grows over the course of the program's runtime. Therefore, like binarytrees, this is a memory intensive program.

*Mandelbrot*

A Mandelbrot set is a member of a class of fractal sets named after the mathematician Benoit Mandelbrot. This program plots the Mandelbrot set on an N-by-N bitmap and writes the output byte-by-byte in portable bitmap format.

*Nbody*

This program simulates the orbits of Jovian (Jupiter class) planets using symplectic-integration. "Body" objects representing the sun, Jupiter, Uranus, Saturn and Neptune are instantiated and then their positions repeatedly re-calculated and updated according to the mechanical model. Lots of computation but a small memory footprint.

*Reverse Complement*

The reverse complement of a DNA strand is the strand that will bind to it. This program takes a FASTA file as input and for each DNA sequence writes its ID, description and reverse complement to an output text file.

*Spectral Norm*

This program calculates the spectral norm of an infinite matrix with some pre-defined entries. Submissions are required to implement four different functions to do so.

### *2.2.5 Procedure*

All available implementations for each of the eight benchmarks in each of the three languages were run via the bencher script, using the same input (n) values and for the same number of times as the published CLBG data so that comparison would be easier. In practice this meant that, for each implementation of each benchmark program, eight total measurements were made: one at the smallest n, one at the next largest n, and six at the largest n value. Of the measurements at the largest n, the fastest implementation in each language for every benchmark was chosen and compiled into a table of fastest results, and compared against the equivalent data from the CLBG. While the real values would naturally be expected to differ across the two environments, the object of the comparison was to determine if the proportional relationship of the results for different languages and programs was roughly the same. Specifically, if there was strong correlation between both data sets in terms of the relative performance of the three languages, then this would be a strong indicator that the data was reliable. Conversely, if there were major disparities between the data sets – if programs and specific implementations that performed best in the CLBG rankings did worst in this study, or vice versa – then this could indicate a serious underlying problem.

After comparing the results of the initial data run against the CLBG data and expectations garnered from the literature review, a series of follow-up investigations were carried out to address anomalies or curious features of the data. This involved performing tunings and configuration changes in compilation or runtime settings and noting the effects on program behaviour. A follow-up investigation was done for each of the three languages under study, targeting a specific aspect of its performance that the initial results had raised questions about.

### *2.2.6 System Specs*

The case study was performed on a Dell E5430 student laptop issued as standard to computing students at Högskolan Kristianstad, with the following specs:

2.6Ghz Intel® Core™ i5-3230M CPU with 4GB of RAM and 108GB Samsung SSD PM830 disk drive; using Windows 7 Professional 64-bit, service pack 1.

The machine used by CLBG to acquire their published data was the following:

Quad-core 2.4Ghz Intel® Q6600® with 4GB of RAM and 250GB SATA II disk drive; using Ubuntu™ 17.04 Linux x64 4.10.0-19-generic.

# 3 Results

This section presents the summarized findings of the literature review followed by the experimental results obtained in the benchmarking case study.

## 3.1 Literature Review

The existing scientific and technical literature offers much valuable information regarding the relative performance characteristics of Java, C# and C++. A summary of this information is given here, beginning with general performance factors relevant to all three languages followed by an examination of the issues specific to each language under study. Previous scientific work related to the subject is then discussed and predictions for the expected results of the case study are made in light of the insights gained.

### *3.1.1 Hardware and Operating Systems*

The available hardware resources and how they are managed by an operating system are environmental factors that will affect the performance of benchmark programs, and these effects may be felt more for one programming language than another. Different operating systems use different algorithms and procedures for memory management and scheduling CPU access for waiting processes. Some of these algorithms are more efficient than others and their performance can in turn be affected by disk fragmentation[14]. In this context the memory allocation conventions specific to certain languages or programming styles can be a factor: for example, a memory allocation scheme that requests small amounts of memory from the OS frequently can incur greater overhead than one that requests a large contiguous block up front and holds onto it. An IBM article focusing on the case of C++ memory management states that,

> `malloc` and `new` make calls to the operating system kernel requesting memory, while `free` and `delete` make requests to release memory. This means that the operating system has to *switch between user-space code and kernel code every time a request for memory is made*. Programs making repeated calls to `malloc` or `new` eventually run slowly because of the repeated context switching.[15]

Another factor is system architecture. Different processors have different address spaces (eg. 32-bit vs 64-bit) which influences the number and size of the registers and how much RAM can be addressed. Architectures also differ in their support of different instruction sets and instruction set extensions, such as SSE, which determine the range of arithmetical operations that the hardware can be instructed to perform. Another factor is the system's support for multiprocessing, where multiple processors can allow for the concurrent execution of multiple threads. From the programming language performance perspective, the ability of a particular language to recognize and make maximal use of these system features when they are available can be critical.

### *3.1.2 Compiled vs Interpreted*

An important consideration relating to performance is whether a programming language is compiled or interpreted. Compiled languages are those whose source code is converted by a compiler directly to the native language of the machine that is to execute it. Interpreted or managed languages are instead compiled to an intermediary language that is then translated into machine code at run time by the language's virtual machine or run-time environment. This difference gives compiled languages a strong theoretical advantage in

terms of run-time performance since the machine can execute code it understands directly, without the additional computational overhead of an interpreter, which occupies both time and space at execution. However, as the virtual machines of interpreted languages continue to be improved, they are increasingly capable of performing intelligent optimizations at run-time. These include identifying "hot spots" or sections of code that are executed frequently and converting them to machine code to speed these sections up, and "loop unrolling" to convert computationally expensive loop iterations into longer but more computationally simple operations.

These kinds of smart optimizations performed at run-time are not available to compiled languages and therefore the optimization work must be done by the programmer and compiler beforehand. If such optimization is done properly, the resultant native code would almost always be expected to show better performance than code which requires an additional interpretation layer. However, if the native code has not been properly optimized for the hardware environment by the programmer and the compiler, it may be expected to be beaten, at least in execution time, by the run-time adaptations made by a smart virtual machine.

### *3.1.3 Memory Management and Garbage Collection*

Dynamic memory allocation is the process by which a program allocates a portion of memory to the data or objects (in the object-oriented paradigm) that are created and used during the course of its runtime. The other side of this problem is memory deallocation, or how objects are removed from memory once they are no longer needed by the program. Different programming languages use different conventions for memory management, placing different degrees of control directly in the programmer's hands. Some schemes for so-called manual memory management rely on the programmer creating and destroying objects individually with the use of appropriate methods and keywords. This creates the potential for "memory leaks" when objects are not deleted correctly and remain occupying space in memory while serving no purpose. Many modern languages try to overcome these difficulties and simplify life for the programmer by implementing automatic memory management with garbage collection (GC). Garbage collection utilities run periodically to remove objects that are no longer needed by the program and free up the space they were occupying for reuse.

Modern GC languages often use "generational" collectors, which are so called because they divide the program's allocated memory into sections representing the longevity of objects: new objects are classified as young or first generation and after they have survived a garbage collection sweep they may be promoted to a second or survivor generation. There are usually several layers of abstraction for memory management schemes even if they are manual. To avoid the overhead of directly calling the OS to release small blocks of memory every time an object is instantiated, memory management libraries and runtime environments will tend to claim larger blocks of memory which they often structure as a heap. There are several performance considerations here. A larger heap will likely cut down on system calls and the attendant overhead but may needlessly inflate a program's memory footprint. Meanwhile GC will require additional computational overhead versus manual memory allocation and deallocation, but in some contexts it may help performance by preventing fragmentation and reclaiming memory rather than having to grow the heap.

### *3.1.4 Java*

Java is an interpreted, statically typed, object-oriented language first released by Sun Microsystems in 1995. It has grown to become the world's most popular programming language thanks to its type-safety, stability and especially portability. A major selling point of Java from the beginning was its cross-platform portability, allowing programmers to "write once, run anywhere" and today Java code runs on many platforms from tiny embedded systems to huge data centers.

Java source code (.java file extension) is compiled to Java bytecode (.class file extension) and the byte code is interpreted and converted into machine-readable instructions at run-time by the Java Virtual Machine (JVM). There are several JVMs developed by different parties but the reference implementation supplied by Oracle with the Java Standard Edition is the Java Hotspot Performance Engine (Hotspot). Hotspot offers a range of features for performance tuning and optimization. A heap is used for dynamic memory allocation together with generational garbage collection: short-lived objects are assigned to the "young generation" and are quickly garbage collected, while longer persisting objects are placed in the "old" or "tenured" generation.

The size limits of this heap can be controlled by supplying arguments, or "flags", to the JVM when initializing it. The flags **-Xms** and **-Xmx** respectively set the minimum and maximum amounts of memory that the heap should occupy. It should be noted that these settings will only determine the minimum and maximum amount of system memory used for the heap and that the total memory footprint of a Java program will also include meta data and the space required by the JVM itself. Oracle recommend that the initial and maximum heap size should be set as equal since re-sizing the heap dynamically triggers a full garbage collection, which carries a performance overhead[15].

The Hotspot JVM also performs several kinds of optimization on the bytecode it interprets. These include:

- Hot spot detection
- Method inlining
- Loop unrolling
- Feedback-directed optimizations [16]

### *3.1.5 C#*

C# was developed by a team led by Anders Hejlsberg at Microsoft and was first released in 2000. As an object-oriented, interpreted language with a C-type syntax, automatic garbage collection, type-safety, and a focus on portability, it is very comparable to Java and the two languages have a very similar problem domain. Whereas Java boasts a large open source development community and large number of libraries, C# is integrated into Microsoft's .NET Framework which was initially proprietary, but has made gradual steps toward community development, culminating in the free and open source .NET Core.

Many features of the C# compilation and interpretation process are analogous to Java. In the case of C#, source code is compiled by the C# compiler, CSC, into Common Intermediate Language (IL) files. The IL is then translated into instructions for the machine at run-time by the .NET equivalent of the JVM, the Common Language Runtime (CLR). As with the JVM, the CLR allocates memory for the processes it manages in a heap and categorizes objects in the heap based on their longevity. The generations are designated 0, 1 and 2, and just as with the JVM, "The runtime's garbage collector stores new objects in

generation 0. Objects created early in the application's lifetime that survive collections are promoted and stored in generations 1 and 2" [17].

### *3.1.6 C++*

C++ is a statically typed, free-form, general purpose, compiled, object-oriented language developed by Bjarne Stroustrup that first appeared in 1983. C++ source code is compiled directly to native instructions for the machine in a two step build process referred to as compiling and linking. Several C++ compilers are available but perhaps the most prominent is g++, the C++ interface of the GNU Compiler Collection (gcc). Various compile time optimizations can be attained through supplying compiler flags or arguments to g++ that indicate which features to activate for the compilation process. Some of these flags pertain to system architecture, for example specifying the processor type and supported instruction sets of the host environment will cause the compiler to produce machine code tailored to the strengths of that system. The O- series of flags specify the level of general optimizations that the compiler should use, ranging from O (no optimizations) up to O-3 for all. Many of the optimizations activated at O-3 resemble things done by the JIT compilers for interpreted languages, such as loop optimization and method inlining.

### *3.1.7 Related Work*

Hundt and his colleagues[13] at Google attempted to compare the performance of four languages, Java, C++, Scala, and Go, with a novel benchmark program which they developed and subsequently performed tuning on for each of the language implementations. The study highlighted the difference language-specific expertise and experience can make in the relative fairness of cross-language benchmarking. Optimal C++ code is unlikely to be written by a programmer whose expertise and experience is all in Java, and vice versa. A colleague of Hundt's, for instance, gained a 30% improvement in the C++ benchmark by leveraging language-specific expertise and using hash_maps instead of maps. Then another colleague, Jeremy Manson, a Java programmer, achieved improvements in the Java version by replacing expensive for-loops, which create a temporary object on loop entry, to while loops. The study seemed to provoke animated debate among the Google engineers themselves who all variously felt that their own specialty language was at times being treated unfairly. In the end Hundt concludes that, "in regards to performance, C++ wins out by a large margin. However, it also required the most extensive tuning efforts, many of which […] would not be available to the average programmer."

Li et al compared the performance of Java and other non-Java languages that run on the JVM, Scala, Clojure, Jruby and Jython[9]. Their work is of interest to this study because they utilize a suite of benchmarks from the CLBG and look in detail at tunings to the JVM and its Garbage Collection settings. They also note an interesting detail about the .NET CLR in comparison to the JVM, stating that the CLR "was originally designed to be an appropriate target architecture for multiple high-level programming languages" and therefore "has a slightly richer bytecode instruction set" than the JVM. Sarimbekov et al also studied JVM languages and also used a selection of ten CLBG benchmarks for their experimental work, noting the limitations of microbenchmarks as measures of real-world performance but defending their value for their study[19]. The authors make several interesting observations about the JVM environment and the CLBG programs. In particular their data on Garbage Collector workload shows the binarytrees and knucleotide programs as the biggest memory allocators and consequently where the GC seems to be busiest.

Sestoft[20] compared C, C# and Java's performance on numeric operations in three small case studies that cover matrix multiplication, a division-intensive loop, and polynomial evaluation. He notes that the Java and C# JIT compilers do not tend to optimize inner loops as aggressively as C and C++ compilers and that this can hurt the interpreted languages on numeric computations. He concludes that C# can be sped up with the use of small amounts of unsafe code and that Java performs surprisingly well given its safety. Bull et al benchmarked Java against C and Fortran with a specific focus on Java's potential suitability for scientific use[11]. The authors run their benchmarks in a range of hardware and software environments, using several machines, Linux and Windows, and a range of JDK versions. They state that on Linux "the mean ratio of fastest Java to fastest C execution times is only 1.07" and conclude that in that context "there is no case for preferring C to Java on grounds of performance".

Hertz and Berger[12] conducted a very interesting study which compared the effects on performance of Java Garbage Collection versus C/C++ style explicit memory management. They use a novel experimental methodology to simulate explicit memory management in Java through an oracular memory manager. After running tests on a benchmark suite with a range of different garbage collectors and different oracular memory manager configurations, they conclude:

> when garbage collection has five times as much memory as required, its runtime performance matches or slightly exceeds that of explicit memory management. However, garbage collection's performance degrades substantially when it must use smaller heaps. With three times as much memory, it runs 17% slower on average, and with twice as much memory, it runs 70% slower. Garbage collection also is more susceptible to paging when physical memory is scarce.

Bacon et al also treat the time-space tradeoff in the course of a study that looks at potential amendments to the Java object model[21]. They note a phenomenon also mentioned by Sarimbekov et al, namely the performance costs associated with maintaining an identifying hash code in the header of all Java objects. The authors note this and other features of the object model can put Java at a disadvantage in programs that rely on the creation of a lot of small objects, which is a consideration to be borne in mind for the case study.

### *3.1.8 Expected Case Study Results*

From the information garnered in the literature review a number of sensible predictions can be derived. The compiled language C++ would be expected to outperform the managed languages Java and C# both in terms of execution times and memory footprint, given the overhead required for virtual machines and garbage collection. This follows from the foundational theory and the findings of Hundt. However, the work of Sestoft and Bull et al suggest that Java and C# will be far from sluggish and may be capable of outperforming insufficiently optimized C++ on some metrics and in some contexts. As the relevant technical literature and Hertz and Berger show, Garbage Collection involves a time-space tradeoff and signs of performance degradation may be detectible in Java if memory available is limited. The work of Li et al and Sarimbekov et al, who used a similar selection of programs from the CLBG as this study, suggests that the binarytrees and knucleotide benchmarks can be expected to allocate the most memory and spend most time in Garbage Collection.

## 3.2 Case Study

Of the eight programs that were selected from the CLBG suite of benchmarks for use in this study, six produced a sufficiently large and balanced result set to be of use in comparing the three languages. Each benchmark program has multiple implementations in each of the three languages, so that the total number of source code files run was 108 (8 programs multiplied by 3 languages multiplied by an average of 4.5 implementations per language). Not all the programs would compile and run successfully within the designated timeout limits and so data was obtained for 86 out of the 108 programs (see Appendix 2, run logs). The most frequently failing language was C++ due to the programs using libraries and environment variables appropriate to Unix systems. A particular source of trouble was the "boost" C++ library and the cpu_ call. Attempts to install and configure the boost library on Windows with Cygwin were not successful and were abandoned due to their consumption of too much time.

In light of this a decision was taken to only compare the three languages' performance on benchmarks where *either* a) more than one implementation had successfully compiled and run for each language *or* b) in cases where only one implementation would run, it ought to be one of the fastest versions according to the CLBG rankings. In the case of two benchmarks, binarytrees and spectralnorm, these criteria were not fulfilled. Only one C++ implementation would compile for each of these programs, and in both cases it was one of the worst performing ones according to the official CLBG data. For this reason, the results from these two benchmarks were excluded from cross-language comparison, to avoid skewing the data by comparing two languages' strongest implementations with one language's weakest. However, since several Java and C# results were obtained for both binarytrees and spectralnorm, the data was retained for later use in measuring those languages against themselves.

For the six remaining benchmarks after the exclusion of binarytrees and spectralnorm, the fastest result obtained from six repeated runs of each program at the largest input value (n) are shown in Table 2 and Table 3. Table 2 groups together the benchmarks that have a high computational workload but relatively little memory allocation and Table 2 shows the more memory intensive programs.

**Table 2: Fastest results for relatively low memory benchmarks**

| Program | Language | Id | Size(B) | CPU(s) | Mem(KB) | Load | Elapsed(s) |
|---|---|---|---|---|---|---|---|
| fannkuchredux | c++ | 7 | 1905 | 13.525 | 6528 | 25% | 13.538 |
| fannkuchredux | java | 1 | 1401 | 58.5 | 135972 | 98% | 14.852 |
| fannkuchredux | csharp | 4 | 1384 | 98.733 | 9924 | 99% | 24.885 |
| fasta | c++ | 5 | 2677 | 4.384 | 7860 | 95% | 1.142 |
| fasta | java | 5 | 2575 | 4.555 | 139312 | 83% | 1.38 |
| fasta | csharp | 4 | 1609 | 8.284 | 43792 | 87% | 2.367 |
| nbody | c++ | 3 | 1937 | 4.54 | 6696 | 25% | 4.541 |
| nbody | java | 4 | 1634 | 6.583 | 135272 | 25% | 6.575 |
| nbody | csharp | 3 | 1471 | 7.285 | 9564 | 25% | 7.279 |

**Table 3: Fastest results for higher memory allocation benchmarks**

| Program | Language | Id | Size(B) | CPU(s) | Mem(KB) | Load | Elapsed(s) |
|---|---|---|---|---|---|---|---|
| knucleotide | c++ | 3 | 1405 | 35.287 | 204280 | 79% | 11.043 |
| knucleotide | csharp | 6 | 1685 | 38.797 | 163688 | 74% | 13.043 |
| knucleotide | java | 6 | 1738 | 49.359 | 749944 | 91% | 13.536 |
| mandelbrot | java | 6 | 996 | 26.848 | 180636 | 97% | 6.893 |
| mandelbrot | csharp | 1 | 975 | 32.838 | 41480 | 99% | 8.244 |
| mandelbrot | c++ | 9 | 1222 | 12.948 | 38128 | 24% | 12.95 |
| revcomp | csharp | 2 | 1942 | 1.217 | 701248 | 31% | 0.843 |
| revcomp | java | 3 | 1849 | 1.841 | 390308 | 51% | 0.861 |
| revcomp | c++ | 3 | 915 | 1.295 | 130784 | 19% | 1.303 |

When compared against the data published on the CLBG website these results seem to largely conform to the expected patterns and relationships. The following key characteristics are consistent across both data sets:

- The relationship of the benchmarks themselves is the same, ie. The order of the programs from shortest to longest execution time and smallest to largest memory footprint is the same.
- The relationship of the different implementations for the same language is generally consistent, ie. the fastest implementation for a given language in the CLBG data is the fastest here, and likewise the slowest. Where this is not the case, it is often because the fastest CLBG implementation would not compile for this case study.
- The relative performance of the languages is generally consistent across both data sets and in line with expected results. C++ programs have consistently lower memory footprint and execution time than Java and C#, whose performance in execution time is often very similar, but with C# using noticeably less memory than Java in most cases.

There are some notable points of discrepancy between the data in Tables 2 and 3 and the CLBG data. In terms of execution time, these differences are best illustrated by comparing the elapsed time data from the case study, shown in Figure 2, with the equivalent data from the CLBG website in Figure 3. In general, C++, despite being fastest in the majority of benchmarks, seems to have underperformed somewhat relative to the CLBG data. In the CLBG data, C++ wins by a clear margin across all six benchmarks, being on average 31% faster than Java. But in the case study data C++ comes first in only four out of six benchmarks and in those cases averages just 19% faster than Java. In the other two benchmarks, mandelbrot and reverse-complement, C++ actually placed last. Mandelbrot is the most noteworthy of these cases, since the specific C++ implementation, id number 9, is the same in both CLBG and case study data sets. In the case of reverse-complement, the fastest C++ program according to CLBG did not compile successfully.

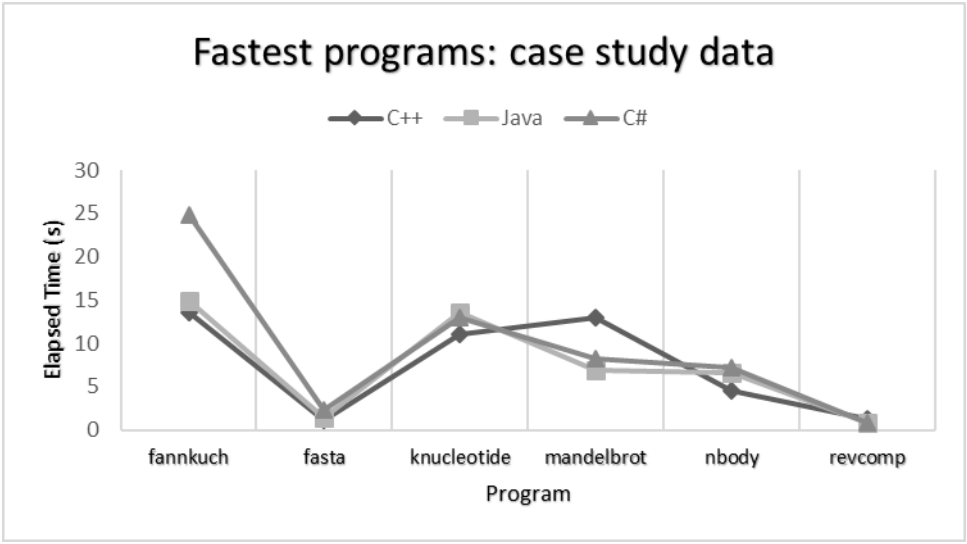



**Figure 2: execution times of the fastest versions of each program in the case study**

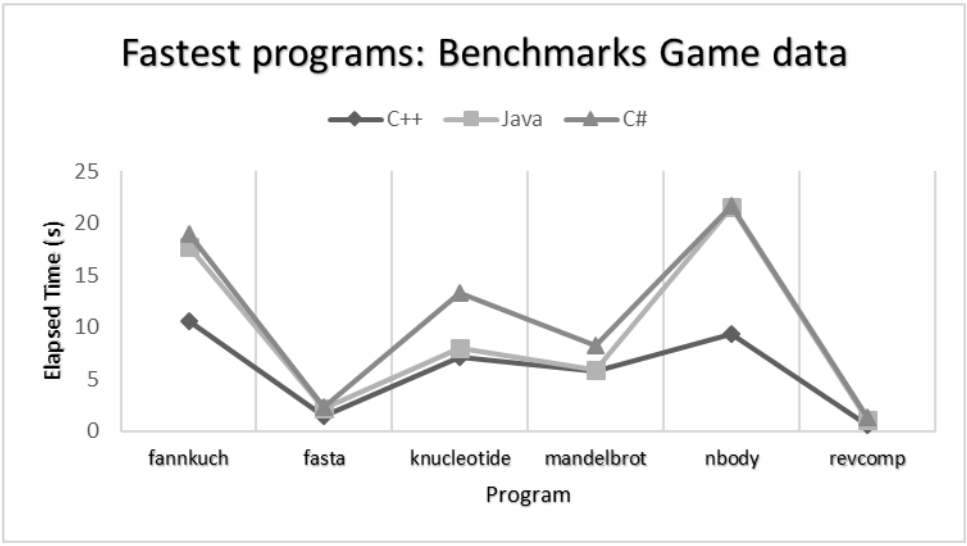



**Figure 3: execution times of the fastest version of each program (CLBG data)**

In terms of memory footprint, there is a consistent pattern to the data from the case study and the CLBG website, illustrated in Figures 4 and 5 respectively. In line with the expected results, C++ consistently shows a significantly smaller memory footprint than the interpreted languages who must allocate additional space to their virtual machines and garbage collectors. Noteworthy is that Java, while consistently the heaviest memory user in both data sets, averages an even higher memory footprint in the case study results than in the CLBG measurements, often by more than 100MB. Also of interest is the memory usage of C#, which for the most part tracks more closely to C++ than Java.

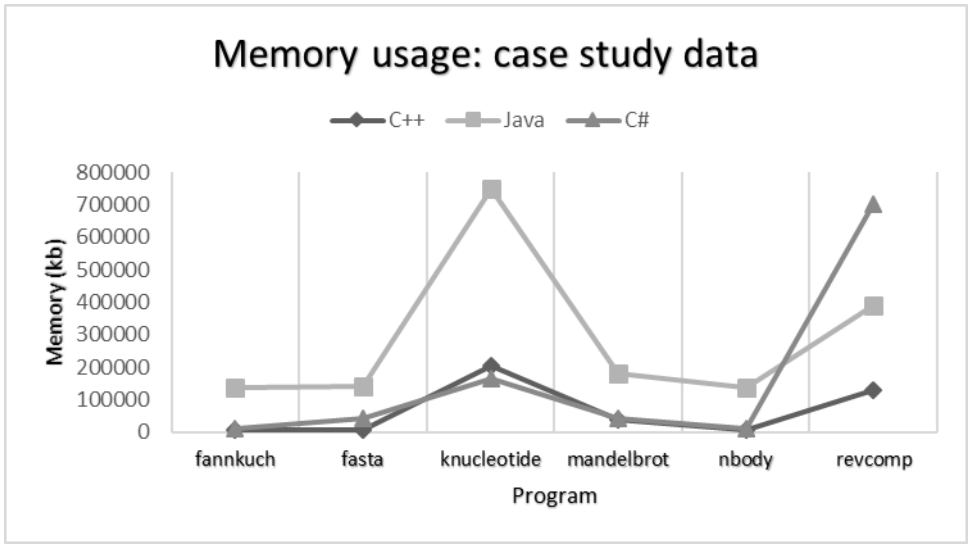



**Figure 4: Memory usage of the fastest versions of each program, case study data**

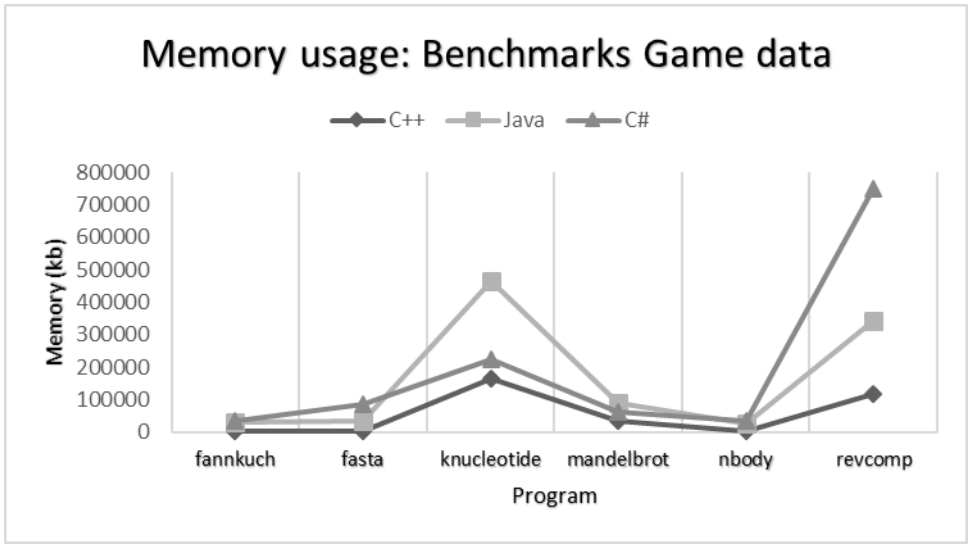



**Figure 5: Memory usage of the fastest versions of each program (CLBG data)**

### *3.2.1 Follow Up Investigation #1: C++ and Mandelbrot*

As noted above, one of the intriguing aspects of the results when compared to the CLBG data is the relative underperformance of C++ in terms of execution time. This underperformance is particularly marked and significant in the mandelbrot benchmark, where the exact same implementation (id number 9) which is a clear winner in the CLBG rankings is a clear loser to both Java and C# in our results. When the data in Table 3 is examined in search of an explanation for this disparity, an apparently significant data point seems to be the figure for CPU load. In the case study results, the Java and C# programs are making efficient use of the CPU, at 97% and 99% respectively. In contrast, the C++ program shows a CPU load of only 24%. This suggests that underutilization of the CPU is at the heart of the problem and, indeed, a check of the CLBG data (see Appendix 3) reveals that CPU utilization by the same Mandelbrot C++ program is up between 95% and 100%. Since the program source code is exactly the same in both cases, this implied an inconsistency in the compilation and/or environments.

After consulting the CLBG website page for the specific mandelbrot program in question, where the source code and build logs are given, it was noticed that the compiler flags used differed slightly from the case study build logs. Specifically an extra instance of the flag *-fopenmp* was used in the CLBG version. The C++ entry in the makefile was therefore amended to add this extra *-fopenmp* flag and the benchmark was re-run. The results of the second run, alongside those of the first, are shown in Table 4.

**Table 4 – first and second run of Mandelbrot C++ no. 9, at all n values**

| Program | Language | Id | n | Size(B) | CPU(s) | Mem(KB) | Run | Load | Elapsed(s) |
|---|---|---|---|---|---|---|---|---|---|
| mandelbrot | c++ | 9 | 1000 | 1222 | 0.062 | 6916 | 1 | 24% | 0.07 |
| mandelbrot | c++ | 9 | 1000 | 1222 | 0.109 | 7196 | 2 | 58% | 0.047 |
| mandelbrot | c++ | 9 | 4000 | 1222 | 0.827 | 8756 | 1 | 24% | 0.843 |
| mandelbrot | c++ | 9 | 4000 | 1222 | 1.888 | 9036 | 2 | 94% | 0.49 |
| mandelbrot | c++ | 9 | 16000 | 1222 | 12.948 | 38128 | 1 | 24% | 12.95 |
| mandelbrot | c++ | 9 | 16000 | 1222 | 29.656 | 38412 | 2 | 98% | 7.558 |

As the table shows, the addition of the extra compiler flag made a significant difference, cutting total execution time by over 40% and boosting CPU utilization to just below maximum for the two largest input values. The nearly five and a half seconds shaved off execution time at n=16000 would now place C++ in second place for this benchmark, ahead of C# and still trailing the fastest Java implementation, but now only by around two thirds of a second.

### *3.2.2 Follow Up Investigation # 2: Java and Memory footprint*

Another notable observation from the initial data run was the relatively high memory use of the Java benchmarks. Although the pattern of memory use by Java was the same in the case study as in the CLBG data (compare Figure 4 and Figure 5), the line graph for the case study data is consistently higher by at least 100MB, and sometimes considerably more. This provided an opportunity to investigate Java and the JVM more closely and raised a number of interesting questions. For example, in cases where Java seems to be faster in our case study than in the CLBG rankings (such as is the case with nbody and fannkuchredux) is it because the JVM is using this extra memory and benefiting from a space-time trade-off? Conversely, if limits are imposed on Java's memory use to bring it more in line with the CLBG data, will this then affect execution times?

To investigate this, the bencher configuration file was amended to include the JVM arguments that specify minimum and maximum heap size, -Xms and -Xmx respectively. All Java implementations for each of the original eight benchmarks (including the two excluded from cross-language comparisons) were then re-run, with the heap initialized at 8MB with -Xms8m and allowed to grow up to a maximum of 512MB with -Xmx512m. These limits were chosen as reasonable lower and upper bounds based on the average size of the programs across all benchmarks and languages. By initializing the heap small and forcing the JVM to resize it dynamically as required, it was hoped that some of the potential performance overhead caused by extra garbage collection, which was referred to in the literature[12], could be captured in the data.

The results for running the Java programs with heap size limits proved highly interesting. The key points are that setting the heap size limits for the JVM led to a considerable decrease in the memory footprint for Java programs across all eight benchmarks. Additionally, in six out of eight cases, this memory footprint reduction was achieved without a significant increase in execution time. For each benchmark, the average memory usage was calculated as a mean of the memory footprint for every individual Java implementation of that benchmark at the highest input value. The resulting average, for the initial data with no memory limits specified and for the second run with heap limits specified, is shown in Figure 6.

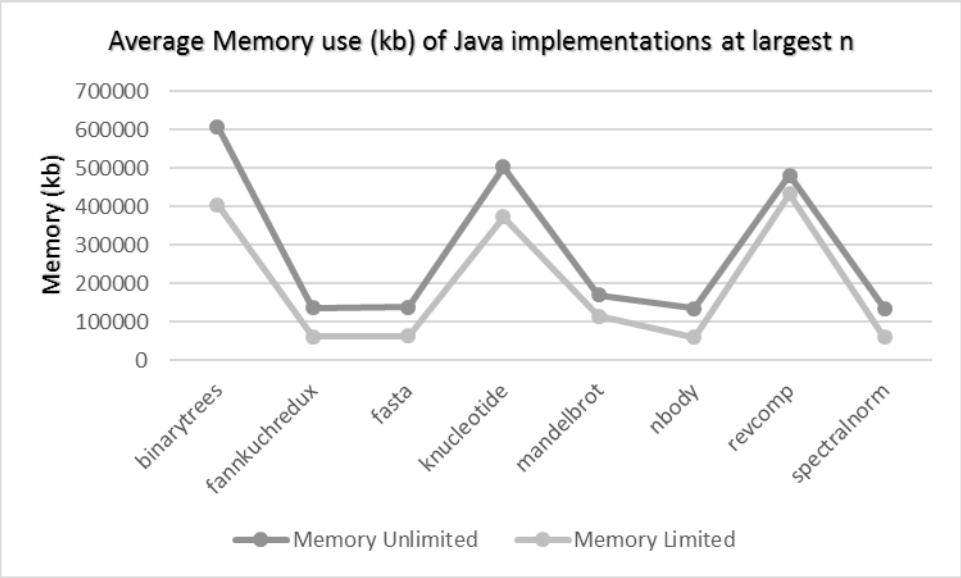



**Figure 6: average memory use of Java implementations of each benchmark, at largest input**

Figure 7 shows the average (mean) of the execution times for every Java implementation of each benchmark, before and after heap memory limits were applied. As the two figures taken together show, the reduction in average memory footprint was bought at no extra performance cost for six out of eight benchmarks. The clear exceptions are binarytrees and knucleotide, where considerable reductions in average memory usage seem to have come at the price of significant increases in average execution time.

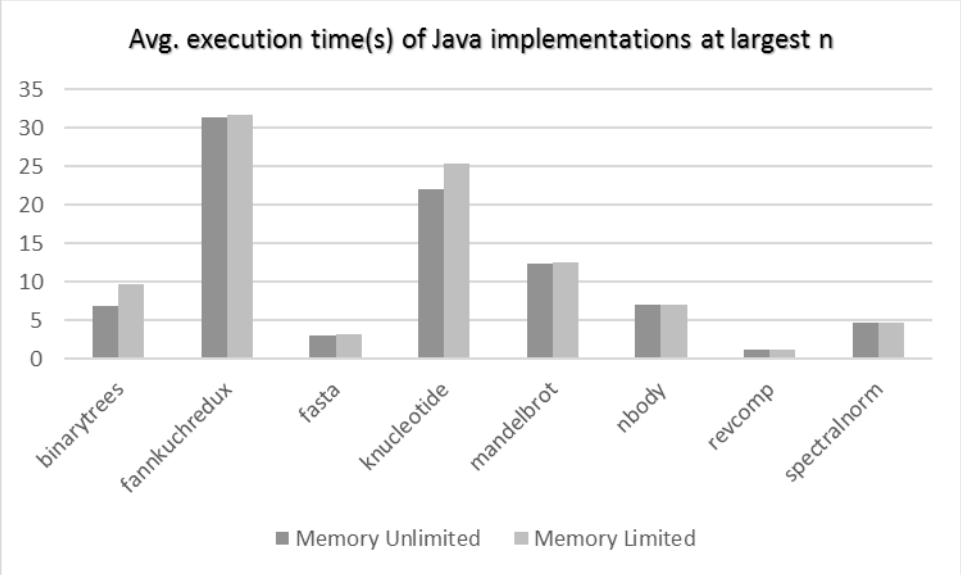


**Figure 7: average execution times of Java implementations for each program, at largest input value**

It is worth taking a closer look at the data for binarytrees and knucleotide to see if one particular implementation that has been affected by the heap size limits and is responsible for pushing up the average execution times. Included as Appendix 4 are two figures showing average execution times for each individual Java implementation of binarytrees and knucleotide, before and after the imposition of heap size limits. The figures show that in the case of binarytrees, the increase in execution time is generalized across all five Java implementations of the benchmark (with program Id number 7 being the most affected) whereas, for knucleotide, program Ids 4 and 5 are responsible for the increase, while programs 3 and 6 are not adversely affected.

### *3.2.3 Follow Up Investigation # 3: Optimizing C#*

Wherever possible this study has attempted to use the same compilation flags and arguments as the CLBG when compiling and running programs. In the case of C++ this was not always straightforward, because several different combinations of flags were given in the build logs at CLBG, varying from benchmark to benchmark and even from implementation to implementation. In the case of C#, the CLBG build logs showed no special compiler flags or arguments supplied when building and running programs. However, since the MSDN documentation lists an "-optimize" flag that can be supplied to the C# compiler, CSC, it was considered interesting to re-run the full suite of benchmarks for all C# implementations with this flag specified, to see if a noticeable difference was made in performance.

For the most part, the results for C# with the compiler -optimize flag set did not vary greatly from the original run. Average memory use did not seem to be notably affected in any of the benchmarks. For average execution times, there were small increases and small decreases in some programs, but a particularly notable decrease was observed in fannkuchredux, where the average execution time fell by 6.4 seconds, as shown in Figure 8.

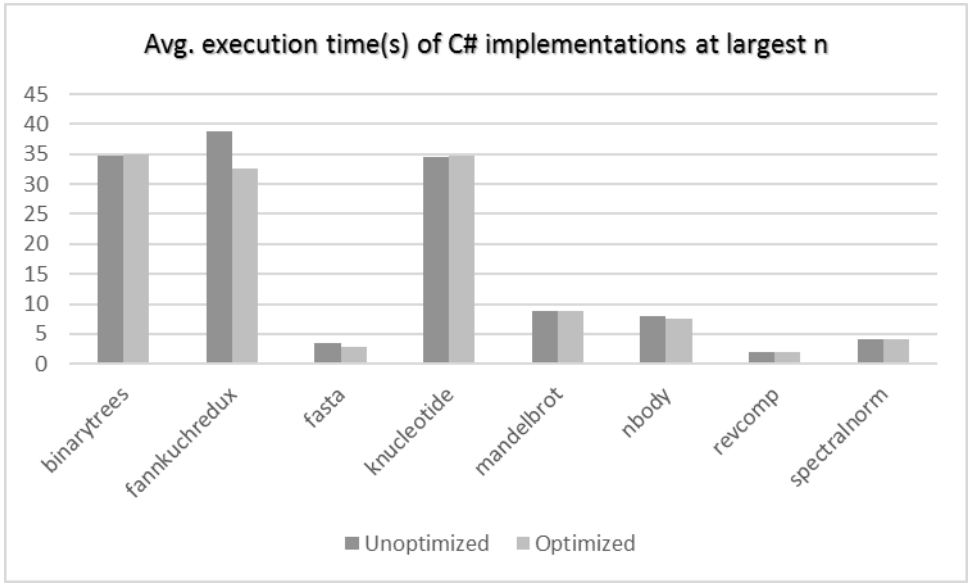



**Figure 8: average execution times of C# implementations for each benchmark, at largest input**

Just as with the Java memory tunings, the question which arises is whether the average time saving is being affected by just one particular implementation or whether it is a trend

observable across all implementations for this benchmark. Figure 9 is a close-up look at average time taken for each of the four C# versions of this fannkuch program, and shows that the effect was detectable on every one.

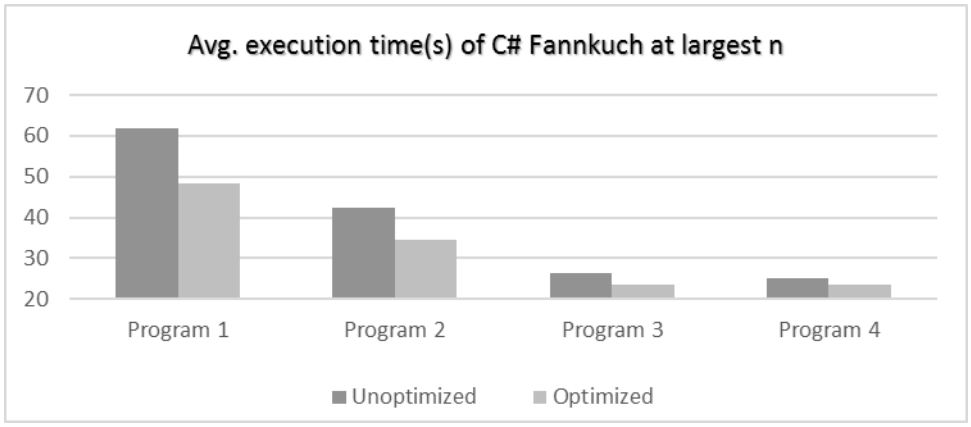



**Figure 9: average execution times for C# fannkuch implementations before and after optimization**

# 4 Discussion

This section discusses the study's findings in the context of the research questions. Some central objectives are to identify and suggest explanations for key features of the case study results, to evaluate the extent to which the data aligns with prior expectations, and to determine what conclusions might reasonably be drawn from it about the relative performance characteristics of C++, Java and C#.

## 4.1 Results as expected in general

The fundamental pattern of the results obtained is in line with expectations and information garnered from the literature review. C++ programs almost always have a considerably smaller memory footprint than their Java and C# counterparts thanks to their being compiled to machine code and not requiring the extra overhead of a runtime environment. C++ is also most frequently the winner in execution time, being fastest in four out of the six benchmarks where more than one implementation for each language would compile. Again, this matches expectations for a language compiled to native code which does not perform garbage collection but rather allows for fine-grained memory management by the programmer. The results also generally line up well with the data from the Computer Language Benchmarks Game, which suggests that there were no major anomalies or configuration problems in the case study environment that might fatally compromise the reliability of the findings. The notable points where the results did stray somewhat from prior expectations and the CLBG data, and which prompted the secondary investigations, will now be examined in more detail.

## 4.2 A general underperformance of C++?

As noted in the presentation of the case study, the average margin of victory for C++ in execution time across the benchmarks suite is somewhat lower in this study's results set than in the CLBG rankings. The difference is illustrated in Table 5 below which shows Java and C# execution times as a factor of C++ runtime, for the fastest performing programs in each of the six benchmarks used for comparison, with this study's data alongside that from the CLBG.

**Table 5: Java and C# execution times as factor of C++ execution times, fastest programs at largest n**

| Program | Benchmarks Game | | Case Study | |
|---|---|---|---|---|
| | Java | C# | Java | C# |
| fannkuch | 1.68 | 1.79 | 1.10 | 1.84 |
| fasta | 1.45 | 1.58 | 1.21 | 2.07 |
| knucleotide | 1.12 | 1.86 | 1.23 | 1.18 |
| mandelbrot | 1.01 | 1.42 | 0.53 | 0.64 |
| nbody | 2.31 | 2.33 | 1.45 | 1.60 |
| revcomp | 1.87 | 2.31 | 0.66 | 0.65 |

With the exception of Mandelbrot, which was the subject of a follow up investigation, these data should probably not be seen as indicating a very significant problem. This is due to a

number of factors. The relatively high failure rate at compilation for C++ programs in this case study must be taken into account. A comparison of the case study results in Tables 2 and 3 with the equivalent CLBG data in Appendix 3 reveals that the two data sets only have the exact same C++ implementation among their fastest results in three out of the six benchmarks. In the case of fannkuchredux, fasta and reverse-complement, the top C++ implementation according to the CLBG either would not compile or had some performance enhancing libraries missing for this study. Of the remaining three benchmarks, knucleotide actually shows Java trailing C++ by a slightly larger factor in the case study, Mandelbrot was a somewhat anomalous result as discovered in the follow-up investigation, and nbody remains as an interesting but now isolated data point.

When we also consider the difference in hardware architecture and environment (Ubuntu for CLBG vs Windows 7 for this study) and all the attendant unknown factors relating to background processes and underlying OS procedures, a great deal cannot be reliably inferred from a marginal difference in C++ performance. It may be that the g++ compiler running directly in the Linux environment on the CLBG test machine was able to pick up more details about the native architecture through flags like "-march=native" than the g++ compiler running via the Cygwin utility in Windows 7. Conversely, it may even be that Java and/or C# were able to relatively overperform in the Windows environment.

## 4.3 Explaining the case of Mandelbrot and C++

As detailed in the results section, a follow-up investigation revealed that a missing "-fopenmp" flag was at least partly responsible for the unusually slow initial reading for C++. When the program was re-run with the flag added, performance improved dramatically (Table 4). An important data point in deducing the probable reason for this is the CPU load. As noted in the results, CPU load was boosted by a factor of 3 from just under 25% to just under 100% with the addition of the flag. According to the documentation for the GNU compiler collection[26], the "-fopenmp" flag enables use of the OpenMP API[27]. OpenMP stands for "Open Multi-Processing" and the API supports multiprocessing by enabling the forking of additional threads to carry out work in parallel, where the environment architecture supports this. The low CPU load on the initial run, around 25%, was a sign that this feature was not enabled and the CPU was only processing a single thread. A look at the source code for the program in question, C++ Mandelbrot ID no. 9, excerpted in Figure 10, reveals several #pragma directives that indicate sections of code where OpenMP threading can be used:

```
#pragma omp parallel for
    for (int x = 0; x < max_x; ++x)
    {
        for (int k = 0; k < 8; ++k)
        {
            const int xk = 8 * x + k;
            cr0[xk] = (2.0 * xk) / width - 1.5;
        }
    }

#pragma omp parallel for schedule(guided)
    for (int y = 0; y < height; ++y)
    {
        Byte* line = &buffer[y * max_x];

        const double ci0 = 2.0 * y / height - 1.0;

        for (int x = 0; x < max_x; ++x)
        {
```

**Figure 10: #pragma omp parallel directives in C++ Mandelbrot #9 source code**

The addition of the extra "-openmp" flag enabled the threading feature, leading to a much busier CPU processing multiple threads in parallel, and shaving almost five and a half seconds off total elapsed execution time. It is perhaps worth noting that one "-openmp" flag was used in the initial run and was used as a standard flag in the compilation of all the benchmarks, but it was a second one that was required in this instance. This is because the build process for C++ comprises two stages, compiling and linking. The flag was included as standard at link time in the CLBG build logs (a practice emulated by this study) to link in the supporting libraries. However, in order to process the #pragma omp directives, the flag must also be added at compile time.

## 4.4 Java and the space-time tradeoff for GC

As detailed in the results section, the initial run of the benchmarks suite showed the Java programs as heavy memory users relative to the other languages. An experiment was conducted in follow-up investigation #2 to see if this memory footprint could be reduced by specifying heap size through arguments to the JVM, and whether this would impact performance. Interestingly, performance was not adversely affected for six out of eight benchmarks, where significant reductions in memory footprint could be gained at no cost in execution time. However, two notable exceptions were binarytrees and knucleotide, where the heap size limitations resulted in a significant increase in average execution times.

These results seem reasonable when considering the characteristics of the benchmarks in question. Specifically, the benchmarks that saw no adverse impact on execution time are almost all programs that do not do much memory allocation. Nbody and fannkuch, for example, are initializing a limited number of objects and then updating or sorting those objects in place, performing many calculations or operations but not requiring more space in which to do it. Binarytrees and knucleotide, on the other hand, are two of the programs that allocate the highest numbers of objects to memory. Binarytrees does so because it creates a succession of large binary trees to traverse, while knucleotide keeps many count values in an ever-growing hash table. These are the sorts of programs for which we might expect to see evidence of space-time trade-off of the kind identified by Hertz and Berger[12].

Partly in hopes of capturing such a trade-off in the data, the heap size was deliberately initialized small at 8MB. Based on the Oracle documentation on JVM memory management, Garbage Collection will be performed aggressively each time the heap is approaching its current limit[17]. If enough memory is not freed up by GC, the heap size will be increased, up to an eventual maximum ceiling determined either by the -Xmx flag or the system resources. The most convenient option for the JVM from a performance perspective is usually to perform GC as infrequently as possible: ie. to grab a large chunk of memory from the operating system initially, and then fill it all. By only allocating 8MB for the heap from the start, the JVM must perform full Garbage Collection on the existing heap before it takes more from the operating system. The added performance overhead for this extra Garbage Collection, as described in the Oracle documentation, is believed to be what caused the slowdown in the binarytrees and knucleotide programs.

In order to try to confirm this hypothesis, the binarytrees implementation, Java # 7, which saw the biggest increase in time with limited heap size, was loaded into the Neatbeans IDE[24] and profiled with that software's built-in profiler. The reason for choosing binarytrees for this test was that knucleotide reads its input data from a file in a process that is managed by the bencher python script. Since binarytrees simply takes a command line argument, it was simpler to get running in Netbeans without requiring any modifications to the source code. The Netbeans profiler was configured with the Telemetry option selected, which monitors CPU and Garbage Collection activity. The program was the run through the profiler, once without heap size arguments and once with the -Xms8m (8MB initial and minimum) and -Xmx512m (512MB maximum) arguments used in the case study.

Figures 11 and 12 show the Netbeans profiler outputs for CPU and Garbage Collector (GC) activity for the memory unlimited and limited runs respectively. The graphs would seem to further support the hypothesis that heap size limits, and in particular heap size initialization at a small size, increase total time in GC and are responsible for longer execution times. The full set of visualizations from the profiling session, including GC detail and Memory, are included as Appendix 5.

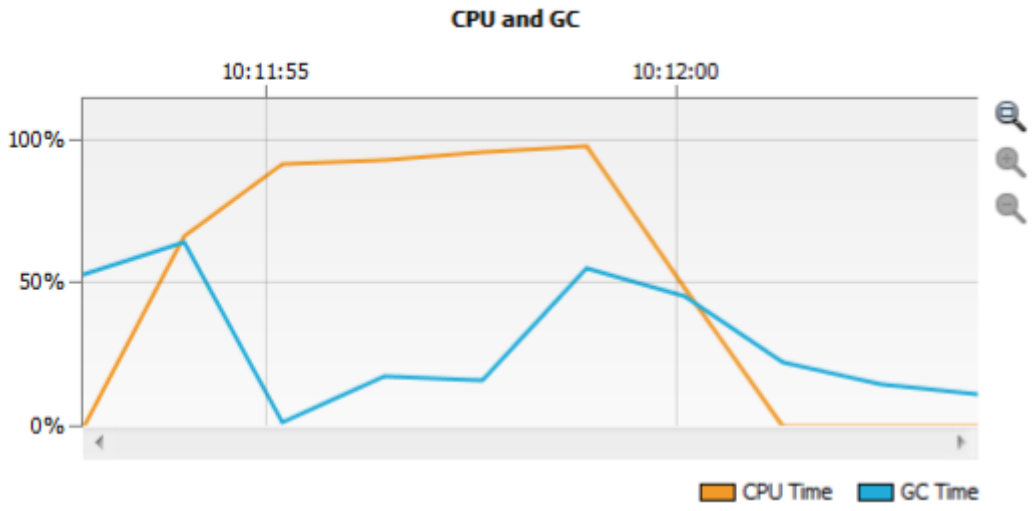



**Figure 11: CPU and GC activity for Java binarytrees #7, memory unlimited**

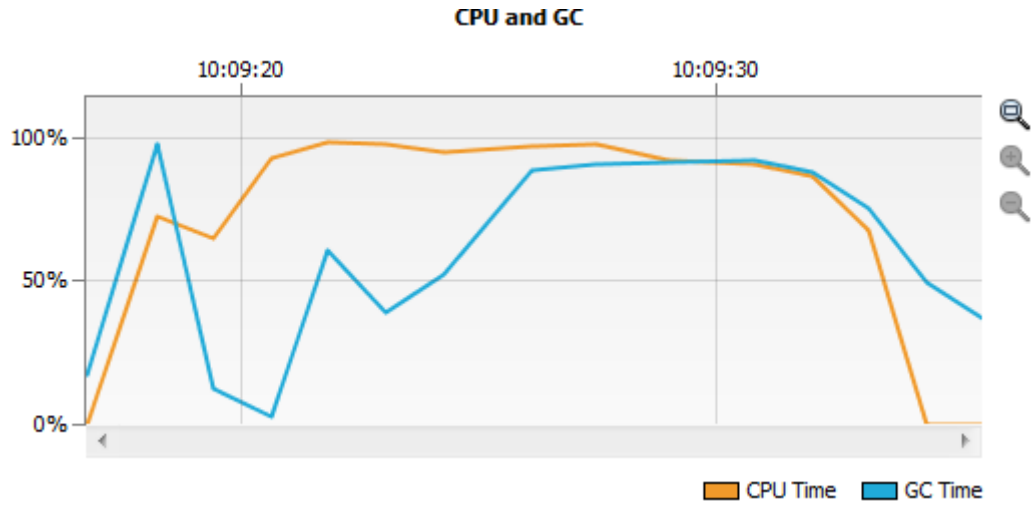



**Figure 12: CPU and GC activity for Java binarytrees #7, memory limited**

## 4.5 C# Optimization and remarks

Follow-up investigation #3 focused on C# and supplying the -optimize flag to CSC at compile time to see if there was a noticeable difference in performance. The result was a significant improvement in average execution time for just one out of eight benchmarks, fannkuchredux. To examine why this might be the case, the source code for the C# fannkuch implementation that was most benefited by switching -optimize on was studied in combination with a 2009 article on the Microsoft Developer's Network (MSDN) site from a then Principal Developer on the C# Compiler Team, Eric Lippert[22]. Lippert explains what the -optimize switch does and gives several examples which match up to features in the fannkuch source code. These include:

- Omitting code generation for assignments like "int foo=0; because we know that the memory allocator will initialize fields to default values". An example of a relevant line from the C# fannkuch source code (see Appendix 6):
  int f = 0, i = 0, k = 0, r = 0, flips = 0, nperm = 0, checksum = 0;
- Attempting to reduce the number of local variable and temporary slots allocated by, for example, reusing storage allocated for the local variable "int i" in a for-loop if another for loop uses "int i" later on. Since most of the work in the fannkuch program involves going into loops and manipulating temporary and permanent int variables, it is reasonable to assume that much of it can be caught by these kinds of optimizations.

Lippert notes that the C# Compiler Team still "rely upon the jitter [ie. Just-in-Time compilation] to do the heavy lifting of optimizations" but the compile-time optimizations he lists seem to fit with speeding up fannkuch. In general, it was more difficult to find detailed documentation on the CLR and its options than for the JVM. This is perhaps partly down to the .NET Framework having been proprietary software for most of its lifetime and therefore more of its features being considered commercial secrets. Much of the information that is available appears to be scattered across the MSDN pages, often in the form of informal developer blog posts and with disclaimers noting that the information is now out of date. This has made it difficult to perform the kind of tweaking and tuning with C# that was possible with the JVM and heap memory.

In general, however, it is worth noting that even without much tuning the performance of C# in this study's data seems more than respectable. While it frequently occupies third place behind Java in terms of execution time, the gap from second to third is usually marginal. The fastest C# implementation is faster than the fastest Java implementation in three of the original eight benchmarks: spectralnorm, knucleotide and reverse-complement. What's more, the C# memory footprint seems impressively compact. The C# knucleotide implementation, ID # 3, which beat Java for speed did so while also coming in 40MB under the C++ program. In three other benchmarks – fannkuch, Mandelbrot and nbody – the fastest C# program has a memory footprint that only exceeds that of C++ by a factor of 1.5 or less.

Some of this may be aided by operating in its native Windows environment for this study. It is noticeable that the C# memory figures are consistently higher in the CLBG data tested on Ubuntu.

## 4.6 Implications for the Research Questions

The findings of both the literature review and the case study have numerous implications for helping us to define the relative performance characteristics of C++, Java and C# and the factors these characteristics depend upon. The technical specifications of the three languages and the foundational theory tell us to expect the compiled language, C++, to execute faster and with a smaller memory footprint than the interpreted, garbage collected languages in the majority of cases. It has been shown, however, that this relationship is dependent on a range of factors. Since an interpreted language's optimizations must all be built-in prior to run-time, this places great importance on the contributions of first the programmer and then the compiler.

Hundt[13] and his colleagues at google demonstrated some of the ways in which the programmer is crucial. Their study showed that elements of language-specific coding style and choice of appropriate objects and idioms can make an important difference. Optimal C++ code is unlikely to be written by a programmer whose expertise and experience is all in Java, and vice versa. A colleague of Hundt's, for instance, gained a 30% improvement in a C++ benchmark by leveraging language-specific expertise and using hash_maps instead of maps. Furthermore, the fine-grained memory management and creation and destruction of objects requires a C++ programmer of considerable skill and experience. Each language has its own performance bottlenecks and optimizations that it takes an experienced developer to exploit.

The C++ section of the case study and the follow-up of the Mandelbrot anomaly helped to underscore the importance of the compilation stage. The two stage build process for C++ programs, compiling and linking, provides a valuable opportunity to build critical optimizations into the executable file. If enough is known about the environment and architecture in which the program will run, as well as the various performance-boosting directives and libraries the programmer has added to the source code, then the addition of appropriate compiler and linker flags can maximize the program's potential and make the resultant native code practically unbeatable on most performance metrics. However, if the proper flags are not set, the program may still compile but run with only a fraction of its potential in the given environment. If the under-optimized C++ cannot make full use of its environment, it is then quite capable of being beaten in execution time, as the case study data has shown, by smart virtual machines and their just-in-time optimizations.

Some important aspects of the limits on interpreted languages' performance have also been demonstrated. Again, the importance of language-specific expertise and appropriate coding style is demonstrated in Hundt[13], where his colleague Jeremy Manson, a Java expert, achieved improvements in the Java implementation of their benchmark by replacing expensive for loops, which create a temporary object on loop entry, to while loops. Meanwhile, in this case study's second follow-up investigation it was demonstrated that garbage collection for Java, and presumably for garbage collected languages in general, is a space-time tradeoff as Hertz and Berger showed[12]. This study has been able to offer empirical confirmation of the literature's claim that Java performance is likely to deteriorate if more time is spent in garbage collection, and that more time in garbage collection can be induced by the imposition of memory limits. For Java to perform optimally, the JVM should preferably be allocated all the heap memory it will require – or ideally more – from the start. If the heap size is capped too low and sufficient memory cannot be freed up by garbage collection, the JVM will throw an out-of-memory exception. If the heap is initialized too low, the subsequent overhead of heap resizing will tend to hurt overall execution time.

In general, it can be said that Java and other interpreted languages can give tremendous speed but at the expense of using much more space, compared with compiled languages. However, some caveats are required. As this study and Hundt[13] have shown, in Java's case the JVM offers extensive options for different kinds of garbage collection tuning, so with patience and expertise the perfect space-time tradeoff balance can be struck. Secondly, the added memory overhead of the virtual machine and garbage collection itself is likely to seem large when dealing with microbenchmarks. But as programs scale up towards the size of large enterprise applications, the percentage of the total application memory footprint taken up by the runtime environment will be progressively less. Therefore, unless multiple servers with multiple virtual machine instances are required, the JVM, CLR or other VM footprint may not be practically important.

# 5 Conclusion

The results of the both the literature review and the experimental case study take us some considerable way in answering the research questions. This section summarizes the insights gained and then discusses opportunities for further work.

## 5.1 Summary of Findings

It is now possible to make several points about the relative performance characteristics of C++, Java and C# and support them with empirical evidence in the form of primary and secondary sources. The chief findings supported both by the literature review and case study are as follows:

- The managed languages Java and C# will almost always be more expensive in terms of space and time than fully optimized C++ because of the extra overhead required for interpretation, garbage collection, JIT profiling and meta data.
- This superiority is however very dependent upon the condition of C++ being fully optimized by the programmer and compiler, incorporating as much information about the native architecture as possible. Suboptimal C++ code can demonstrably be beaten, at least on the axis of time, by run-time optimized languages.
- Manual memory allocation (C++) vs. Garbage Collection (Java/C#) is a major component of observable performance differences between the languages. GC is a time-space tradeoff and the less surplus memory is available for it, the more it will have to run and the more total execution time will be affected.
- Maximizing the optimization potential for a given language requires considerable language-specific skill and experience. The best C++ code should not be written in Java style, and vice versa. However, Java and C# prioritize developer productivity and take much of the fine-grained optimization burden off the programmer, placing it on the virtual machine instead. C++ will generally demand more programmer hours to tune and get right in order to match or better the performance of the managed languages.

## 5.2 Recommendations

Given the findings, the general recommendations would be to choose C++ over Java or C# in contexts where the absolute fastest and most efficient possible result is required, regardless of other considerations. However, most real-world scenarios will require taking a holistic view of several competing factors. Unlocking the full performance potential of C++ does not come cheap in terms of developer expertise and man hours, especially when considering the need to work around the language's flaws like memory leaks and type-unsafety. In many contexts, such as general business applications, the performance of managed languages will be considered more than good enough and will come at a lower cost in terms of developer time, bugs and complexity. However, as Herb Sutter notes[5] at the level of the Data Centre and the technology companies running services with millions of users at massive scale, the cost of employing specialist C++ developers will be small compared to the potentially enormous savings that its efficiency can bring, in terms of hardware and power usage.

## 5.3 Further Work

Future studies could expand upon this work by attempting to more accurately reflect real-world uses for the languages by profiling full size applications rather than benchmarks. While the CLBG benchmarks suite in this study helped to identify some key performance characteristics and pain points native to each language, the question of how these issues scale up and interact with other factors cannot be known unless full applications are studied. It would also be advantageous to repeat this and other, more advanced studies across a wider range of system architectures to see if the observed patterns are in any way environment dependent. The relatively high degree of correlation between the data sets for 64-bit Windows in this study with the 64-bit Ubuntu data from CLBG speaks in favour of the results' validity. However, more architectures and operating systems should be used, and a range of different profiling programs, to ensure the repeatability of the findings.

# 6 Bibliography


[1] Knuth, D E.; Pardo, L T. *Early development of programminglanguages*. Encyclopedia of Computer Science and Technology. Marcel Dekker. 7: 419–493

[2] Ghezzi, C; Jazayeri, M; Mandrioli, D. *Fundamentals of Software Engineering*, Prentice Hall PTR, Upper Saddle River, NJ, 2002

[3] Stefik, A; Hanenberg, S. *The Programming Language Wars: Questions and Responsibilities for the Programming Language Community*, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, October 20-24, 2014, Portland, Oregon, USA

[4] Widman, J. *The Most Popular Programming Languages of 2016*, web, available at: https://blog.newrelic.com/2016/08/18/popular-programming-languages-2016-go/

[5] Sutter, H: *Why C++?*, *C++ and Beyond 2011*, presentation and slides available from MSDN at: https://channel9.msdn.com/posts/C-and-Beyond-2011-Herb-Sutter-Why-C

[6] Moore, G E. *Cramming more components onto integrated circuits*, Readings in computer architecture, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2000

[7] Hennessy, J. L; Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 5th Ed, Elsevier, 2012.

[8] Gouy, I. *The Computer Language Benchmarks Game*. Web .<http://benchmarksgame.alioth.debian.org/>

[9] Li, W H; White, D R; Singer, J. *JVM-hosted languages: they talk the talk, but do they walk the walk?*, Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, September 11-13, 2013, Stuttgart, Germany

[10] Sarimbekov, A., Stadler, L., Bulej, L., Sewe, A., Podzimek, A., Zheng, Y., Binder, W.: *Workload Characterization of JVM Languages*. Software: Practice and Experience (2015), http: //dx.doi.org/10.1002/spe.2337

[11] Bull, J. M; Smith, L. A; Pottage, L; Freeman, R. *Benchmarking Java against C and Fortran for scientific applications*, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, p.97-105, June 2001, Palo Alto, California, USA

[12] Hertz, M; Berger, E. D. *Quantifying the performance of garbage collection vs. explicit memory management*, Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 16-20, 2005, San Diego, CA, USA

[13] Hundt, R.: *Loop recognition in C++/Java/Go/Scala*. In: Proceedings of Scala Days (2011), available online at: http://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[14] Silberschatz, A., Galvin, P., Gagne, G. *Operating System Concepts*, 8th Ed. Ch. 8

[15] Sen, A., Kardam, R., *Building your own memory manager for C/C++ projects*, IBM Developer Works, Web. Available at: https://www.ibm.com/developerworks/aix/tutorials/au-memorymanager/

[16] Khubay, A., Wang, P., Shreenidhi, R. *Oracle AIA 11g Performance Tuning on Oracle's SPARC T4 Servers A Guide for Tuning Pre-Built Integrations on Oracle AIA 11g*, Oracle Whitepaper, available at: http://www.oracle.com/us/products/applications/aia-11g-performance-tuning-1915233.pdf

[17] Oracle. *The Java Hotspot Performance Engine Architecture*, Oracle Whitepaper, available at: http://www.oracle.com/technetwork/java/whitepaper-135217.html#garbage

[18] MSDN. *Automatic Memory Management*, Web. Available at: https://msdn.microsoft.com/en-us/library/f144e03t(v=vs.110).aspx

[19] Sarimbekov, A; Podzimek, A; Bulej, L; Zheng, Y; Ricci, N; Binder, W. *Characteristics of dynamic JVM languages*, Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, October 28-28, 2013, Indianapolis, Indiana, USA

[20] Sestoft, P. *Numeric performance in C, C# and Java*, IT University of Copenhagen, 2010, available at: http://www.itu.dk/~sestoft/papers/numericperformance.pdf

[21] Bacon, D. F; Fink, S. J; Grove, D. *Space- and Time-Efficient Implementation of the Java Object Model*, Proceedings of the 16th European Conference on Object-Oriented Programming, p.111-132, June 10-14, 2002

[22] Lippert, E. *What does the optimize switch do?, Fabulous Adventures in Coding, MSDN*, Web, available at: https://blogs.msdn.microsoft.com/ericlippert/2009/06/11/what-does-the-optimize-switch-do/

[23] GNU Make for Win32, Web: available at: http://gnuwin32.sourceforge.net/packages/make.htm

[24] Cygwin Project homepage, available at: https://www.cygwin.com/

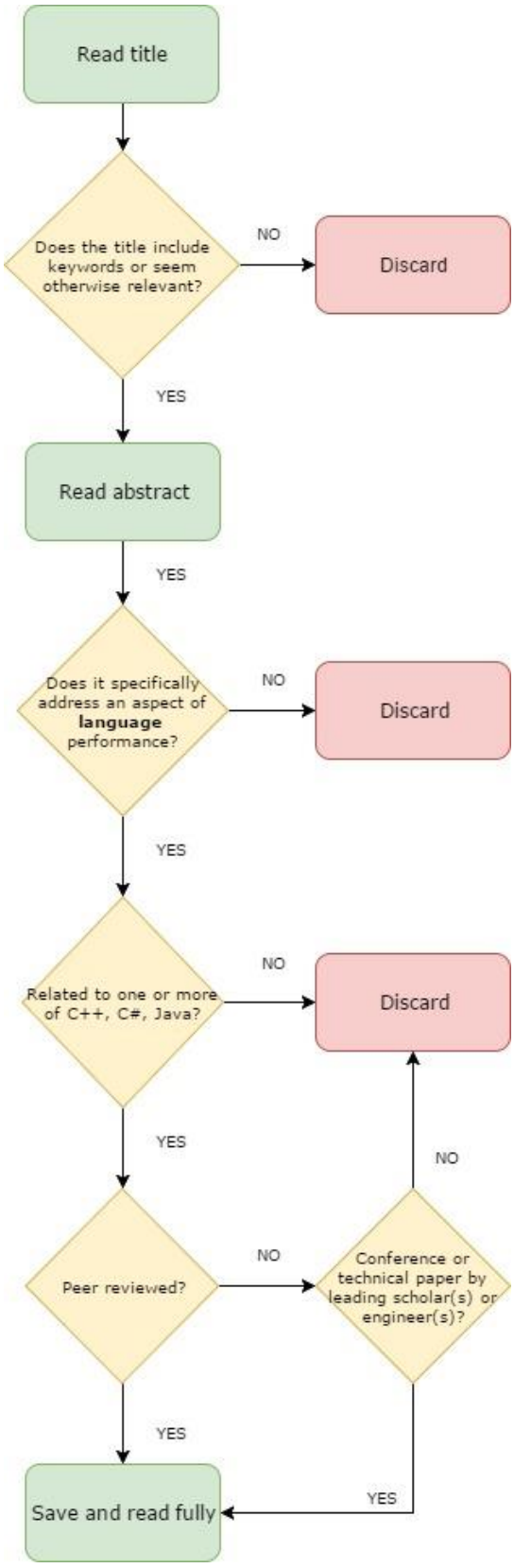
[25] Netbeans Integrated Development Environment, available at: https://netbeans.org/

[26] GNU Compiler Docs. *Enabling OpenMP*. Available at: https://gcc.gnu.org/onlinedocs/libgomp/Enabling-OpenMP.html

[27] OpenMP API, available at: http://www.openmp.org/

# 7 Appendices and Enclosures

## Appendix 1: Heuristic for literature selection



# Appendix 2: Run logs

| Program | Language | Implementation ID | Run Status | Comment |
|---|---|---|---|---|
| binarytrees | csharp | 1 | Succeeded | |
| binarytrees | csharp | 2 | Succeeded | |
| binarytrees | csharp | 3 | Succeeded | |
| binarytrees | csharp | 4 | Succeeded | |
| binarytrees | c++ | 1 | FAILED | Missing dependencies |
| binarytrees | c++ | 2 | Succeeded | |
| binarytrees | c++ | 3 | FAILED | Missing dependencies |
| binarytrees | c++ | 6 | FAILED | Missing dependencies |
| binarytrees | c++ | 8 | FAILED | Missing dependencies |
| binarytrees | c++ | 9 | FAILED | Missing dependencies |
| binarytrees | java | 2 | Succeeded | |
| binarytrees | java | 3 | Succeeded | |
| binarytrees | java | 4 | Succeeded | |
| binarytrees | java | 6 | Succeeded | |
| binarytrees | java | 7 | Succeeded | |
| fannkuchredux | csharp | 1 | Succeeded | |
| fannkuchredux | csharp | 2 | Succeeded | |
| fannkuchredux | csharp | 3 | Succeeded | |
| fannkuchredux | csharp | 4 | Succeeded | |
| fannkuchredux | c++ | 1 | Succeeded | |
| fannkuchredux | c++ | 2 | FAILED | Missing dependencies |
| fannkuchredux | c++ | 3 | Succeeded | |
| fannkuchredux | c++ | 4 | FAILED | Missing dependencies |
| fannkuchredux | c++ | 5 | FAILED | Missing dependencies |
| fannkuchredux | c++ | 7 | Succeeded | |
| fannkuchredux | java | 1 | Succeeded | |
| fannkuchredux | java | 2 | Succeeded | |
| fannkuchredux | java | 3 | Succeeded | |
| Fasta | csharp | 2 | Succeeded | |
| Fasta | csharp | 3 | Succeeded | |
| Fasta | csharp | 4 | Succeeded | |
| Fasta | c++ | 1 | FAILED | Compilation Error |
| Fasta | c++ | 2 | FAILED | Compilation Error |
| Fasta | c++ | 3 | Succeeded | |
| Fasta | c++ | 4 | FAILED | Compilation Error |
| Fasta | c++ | 5 | Succeeded | |
| Fasta | c++ | 6 | Succeeded | |

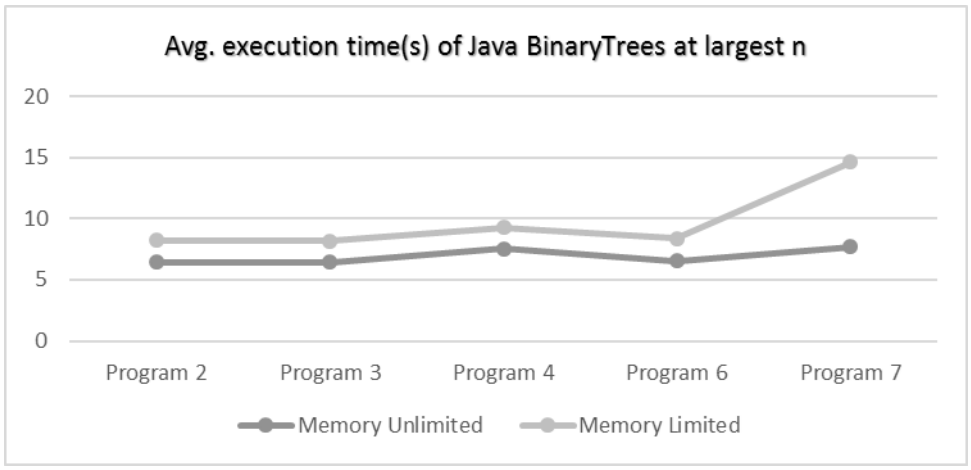
| | | | | |
|---|---|---|---|---|
| Fasta | java | 2 | Succeeded | |
| Fasta | java | 4 | Succeeded | |
| Fasta | java | 5 | Succeeded | |
| knucleotide | csharp | 1 | Succeeded | |
| knucleotide | csharp | 2 | Succeeded | |
| knucleotide | csharp | 3 | Succeeded | |
| knucleotide | csharp | 4 | Succeeded | |
| knucleotide | csharp | 5 | FAILED | Compilation Error |
| knucleotide | csharp | 6 | Succeeded | |
| knucleotide | csharp | 7 | Succeeded | |
| knucleotide | c++ | 1 | FAILED | Compilation Error |
| knucleotide | c++ | 3 | Succeeded | |
| knucleotide | java | 1 | FAILED | Compilation Error |
| knucleotide | java | 3 | Succeeded | |
| knucleotide | java | 4 | Succeeded | |
| knucleotide | java | 5 | Succeeded | |
| knucleotide | java | 6 | Succeeded | |
| mandelbrot | csharp | 1 | Succeeded | |
| mandelbrot | csharp | 2 | FAILED | Compilation Error |
| mandelbrot | csharp | 3 | Succeeded | |
| mandelbrot | csharp | 5 | Succeeded | |
| mandelbrot | c++ | 2 | Succeeded | |
| mandelbrot | c++ | 3 | Succeeded | |
| mandelbrot | c++ | 5 | Succeeded | |
| mandelbrot | c++ | 6 | FAILED | Missing dependencies |
| mandelbrot | c++ | 7 | FAILED | Missing dependencies |
| mandelbrot | c++ | 8 | Succeeded | |
| mandelbrot | c++ | 9 | Succeeded | |
| mandelbrot | java | 1 | Succeeded | |
| mandelbrot | java | 2 | Succeeded | |
| mandelbrot | java | 3 | Succeeded | |
| mandelbrot | java | 6 | Succeeded | |
| Nbody | csharp | 1 | Succeeded | |
| Nbody | csharp | 2 | Succeeded | |
| Nbody | csharp | 3 | Succeeded | |
| Nbody | csharp | 6 | Succeeded | |
| Nbody | csharp | 8 | Succeeded | |
| Nbody | c++ | 1 | Succeeded | |
| Nbody | c++ | 3 | Succeeded | |
| Nbody | c++ | 4 | Succeeded | |

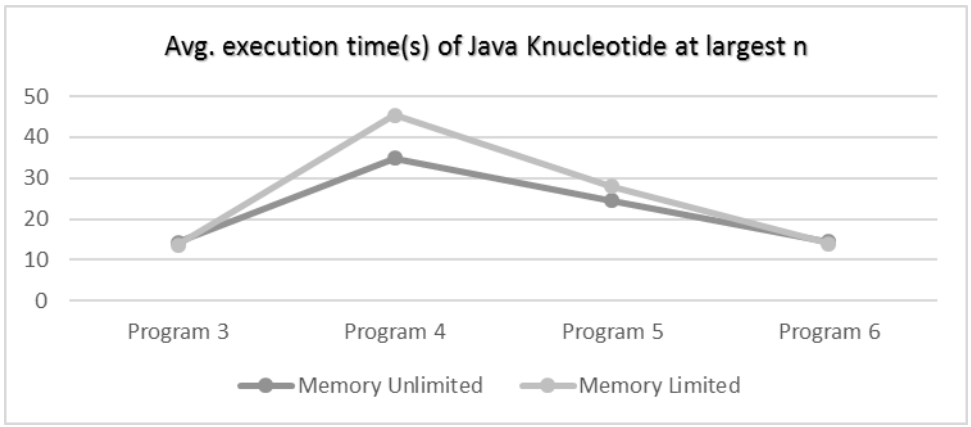
| | | | | |
|---|---|---|---|---|
| Nbody | c++ | 5 | Succeeded | |
| Nbody | c++ | 6 | Succeeded | |
| Nbody | c++ | 7 | Succeeded | |
| Nbody | c++ | 8 | Succeeded | |
| Nbody | java | 1 | Succeeded | |
| Nbody | java | 2 | Succeeded | |
| Nbody | java | 3 | Succeeded | |
| Nbody | java | 4 | Succeeded | |
| Revcomp | csharp | 1 | Succeeded | |
| Revcomp | csharp | 2 | Succeeded | |
| Revcomp | csharp | 3 | Succeeded | |
| Revcomp | csharp | 4 | Succeeded | |
| Revcomp | c++ | 1 | Succeeded | |
| Revcomp | c++ | 2 | FAILED | Source code bug |
| Revcomp | c++ | 3 | Succeeded | |
| Revcomp | c++ | 4 | FAILED | Source code bug |
| Revcomp | c++ | 5 | Succeeded | |
| Revcomp | c++ | 6 | Succeeded | |
| Revcomp | java | 3 | Succeeded | |
| Revcomp | java | 4 | Succeeded | |
| Revcomp | java | 5 | Succeeded | |
| Revcomp | java | 6 | Succeeded | |
| Revcomp | java | 7 | Partial Success | Timed out at largest n |
| spectralnorm | csharp | 1 | Succeeded | |
| spectralnorm | csharp | 3 | Succeeded | |
| spectralnorm | gpp | 1 | Succeeded | |
| spectralnorm | gpp | 5 | FAILED | Compilation Error |
| spectralnorm | gpp | 6 | FAILED | Compilation Error |
| spectralnorm | gpp | 8 | FAILED | Compilation Error |
| spectralnorm | java | 1 | Succeeded | |
| spectralnorm | java | 2 | Succeeded | |

# Appendix 3: Fastest results (CLBG Data)

| Program | Language | Id | Size(B) | CPU(s) | Mem(KB) | Load | Elapsed(s) |
|---|---|---|---|---|---|---|---|
| binarytrees | c++ | 9 | 846 | 10.592 | 158148 | 81% 82% 78% 75% | 3.363 |
| binarytrees | java | 7 | 889 | 40.536 | 542340 | 89% 92% 85% 89% | 11.517 |
| binarytrees | csharp | 4 | 955 | 52 | 754680 | 91% 90% 89% 89% | 14.634 |
| fannkuchredux | c++ | 5 | 950 | 41.876 | 2520 | 100% 100% 99% 98% | 10.579 |
| fannkuchredux | java | 1 | 1282 | 69.904 | 28776 | 98% 98% 100% 99% | 17.739 |
| fannkuchredux | csharp | 4 | 1172 | 74.752 | 34896 | 99% 99% 98% 99% | 18.911 |
| fasta | c++ | 6 | 2291 | 5.132 | 4068 | 88% 88% 87% 88% | 1.473 |
| fasta | java | 5 | 2457 | 5.68 | 34404 | 71% 58% 62% 77% | 2.139 |
| fasta | csharp | 3 | 1904 | 8.172 | 84972 | 85% 98% 85% 85% | 2.33 |
| knucleotide | c++ | 3 | 1252 | 24.212 | 164080 | 82% 83% 80% 96% | 7.15 |
| knucleotide | java | 1 | 1802 | 25.572 | 465936 | 76% 98% 73% 74% | 8.022 |
| knucleotide | csharp | 6 | 1585 | 36.896 | 223584 | 57% 97% 53% 73% | 13.278 |
| mandelbrot | c++ | 9 | 726 | 22.4 | 33952 | 96% 95% 95% 100% | 5.818 |
| mandelbrot | java | 2 | 796 | 23.084 | 87924 | 98% 98% 98% 99% | 5.889 |
| mandelbrot | csharp | 5 | 839 | 32.664 | 62480 | 99% 100% 99% 99% | 8.265 |
| nbody | c++ | 3 | 1763 | 9.288 | 1692 | 100% 1% 1% 0% | 9.295 |
| nbody | java | 4 | 1489 | 21.516 | 27236 | 1% 1% 100% 0% | 21.501 |
| nbody | csharp | 3 | 1305 | 21.684 | 34920 | 0% 100% 1% 2% | 21.693 |
| revcomp | c++ | 4 | 2275 | 0.836 | 118840 | 26% 78% 12% 34% | 0.593 |
| revcomp | java | 3 | 1661 | 2.44 | 343752 | 33% 58% 54% 80% | 1.109 |
| revcomp | csharp | 2 | 1670 | 2.208 | 750280 | 71% 49% 22% 24% | 1.368 |
| spectralnorm | c++ | 6 | 1044 | 7.996 | 1788 | 100% 100% 100% 100% | 2.012 |
| spectralnorm | csharp | 3 | 878 | 15.8 | 40152 | 97% 97% 98% 99% | 4.052 |
| spectralnorm | java | 2 | 950 | 16.56 | 28644 | 97% 96% 98% 97% | 4.287 |

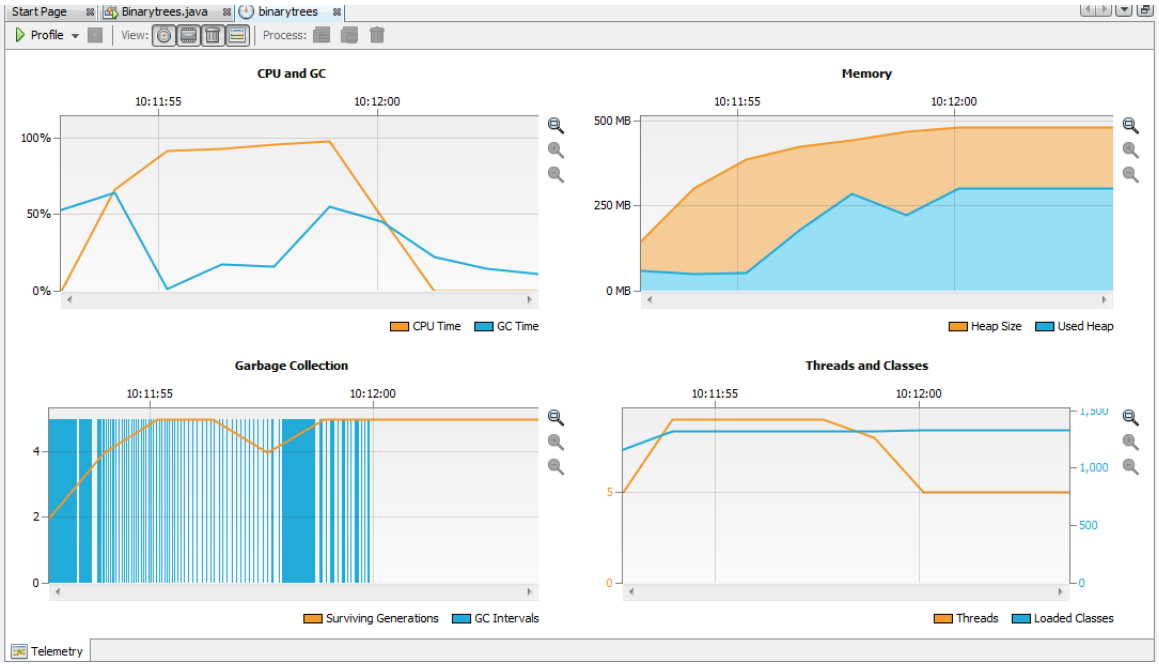
# Appendix 4: Java BinaryTrees and Knucleotide execution times, with and without memory limtis



# Appendix 5: Netbeans Profiling session screenshots, Java binarytrees

## Memory Unlimited:




## Memory Limited:

# Appendix 6: source code for C# Fannkuch implementation # 1

```
/* The Computer Language Benchmarks Game
   http://benchmarksgame.alioth.debian.org/

   contributed by Isaac Gouy, transliterated from Rex Kerr's Scala program
*/

using System;

class FannkuchRedux
{
   public static int fannkuch(int n) {
      int[] perm = new int[n], perm1 = new int[n], count = new int[n];
      for(int j=0; j<n; j++) perm1[j] = j;
      int f = 0, i = 0, k = 0, r = 0, flips = 0, nperm = 0, checksum = 0;

      r = n;
      while (r>0) {
         i = 0;
         while (r != 1) { count[r-1] = r; r -= 1; }
         while (i < n) { perm[i] = perm1[i]; i += 1; }

         // Count flips and update max  and checksum
         f = 0;
         k = perm[0];
         while (k != 0) {
           i = 0;
           while (2*i < k) {
             var t = perm[i]; perm[i] = perm[k-i]; perm[k-i] = t;
             i += 1;
           }
           k = perm[0];
           f += 1;
         }
         if (f>flips) flips = f;
         if ((nperm&0x1)==0) checksum += f;
         else checksum -= f;

         // Use incremental change to generate another permutation
```

```
            var go = true;
            while (go) {
               if (r == n) {
                  Console.WriteLine(checksum);
                  return flips;
               }
               var p0 = perm1[0];
               i = 0;
               while (i < r) {
                  var j = i+1;
                  perm1[i] = perm1[j];
                  i = j;
               }
               perm1[r] = p0;

               count[r] -= 1;
               if (count[r] > 0) go = false;
               else r += 1;
            }
            nperm += 1;
         }
         return flips;
      }

      static void Main(string[] args){
         int n = 7;
         if (args.Length > 0) n = Int32.Parse(args[0]);
         Console.WriteLine("Pfannkuchen({0}) = {1}", n, fannkuch(n));
      }
   }
```