Principled software micro-engineering

Adrian Johnstone and Elizabeth Scott

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey, United Kingdom

Abstract

Object oriented and pattern based metaphors for software present a solid engineering base for software understanding and construction, but sometimes impose a high performance overhead. We quantify this overhead for one form of generalised parsing and propose support for *implementation facets* in high level programming languages.

Keywords: Programming, performance, programming languages

1. Productivity and performance

In any form of engineering, success is primarily measured in terms of correctness, safety, price and performance, and requires an ability to predict price and performance in advance so that trade-off decisions may be made before and during resource commitment. This last is particularly difficult for software. Unlike, say, mechanical systems, software displays a very wide spread of price-performance possibilities. For instance even Formula One cars are only small integer factors faster than road cars, and perhaps only one order of magnitude faster than a bicycle. However, in digital electronics (and by extension software) the playing out of Moore's 'Law' has until recently delivered a doubling in performance every 18 months. As a result, multiple generations of systems are in the marketplace at any one time at different price points.

Performance improvements have allowed software engineers to attack complexity by abstracting away from practices which are natural at machine level, though with a concomitant loss of relative colck speed performance which is masked by the exponential improvement in cycle time. However, there are application areas such as gaming, scientific programming and embedded systems where the work done per machine cycle must be maximised, so as to either extract maximum throughput or minimum power consumption. In these areas detailed optimised designs which we call *software microengineering* predominate.

Email address: `a.johnstone@rhul.ac.uk, e.scott@rhul.ac.uk` (Adrian Johnstone and Elizabeth Scott)

There are also subsystems in desktop computing that merit a microengineering approach. In this paper we present some relative performance data for one such application — general context free parsing — and by borrowing ideas from the field of Hardware Description Languages (HDLs) propose the use of *faceted* programming languages to facilitate principled software microengineering.

2. Software engineering in computing pre-history

The inverse of Moore's Law is that early machines were extraordinarily slow and small, as well as rare and expensive. As a result, in the earliest years, the emphasis was mostly on how to maximise the productivity of the machine. In April 1948, James Wilkinson reported on the work of Turing and his collaborators on the Pilot ACE, and the planned subsequent full scale ACE model [1]. This was two years before Pilot ACE ran its first real program (in May 1950), and indeed two months before the Manchester Baby ran the world's first stored-program code. Even though no stored-program computers existed at the time of Wilkinson's report, the emphasis was already on performance. Instructions and data were stored in mercury delay lines and read out sequentially-by-bit. Partial computations were held in short tanks (analogous to modern registers) and programs in long tanks, or later in a drum. Turing proposed that instructions be arranged in consecutive lock-stepped delay lines in such a way that the next instruction would be available as soon as the last had finished processing.

> The most obvious way of arranging the coded instructions in the memory is to let each instruction comprise one word, and place successive instructions consecutively in long tanks. This has the disadvantage that after the control has read one instruction it cannot receive the next instruction until one major cycle afterwards. Since most of the manipulation will be performed on numbers which are being stored temporarily in short tanks this involves an unnecessary waste of time. The alternative adopted, is to space the instructions in such positions in the instruction tanks, that when instruction is completed, the next is in the correct position for it to be obeyed. [1]

This approach (which required each machine instruction to carry the address of its successor) developed into Turing's doctrine of *Optimum Programming* in which data and programs were laid out in storage in such a way as to minimise fetch latencies. This required a very difficult programming style which Christopher Strachey explicitly eschewed in the 1953-54 design of the Ferranti PEGASUS because 'it tended to become a time-wasting intellectual hobby of the programmers' [2, p.79]. Maurice Wilkes had his own reasons for rejecting optimum programming: in his 1967 Turing Award lecture he rather imperiously said: 'I felt that this kind of human ingenuity was misplaced as a long-term investment, since sooner or later we would have truly random-access memories. We therefore did not have anything to do with optimum coding in Cambridge' [3]. In the modern age, though, Turing has had his revenge; the techniques of optimum programming live on in the scheduling and cache management algorithms at the heart of any good optimising compiler. The difference now is that only the compiler implementer has to be aware of the intricacies.

Turing also considered programmer productivity, but again in terms of machine throughput:

> If the coding of each problem were attempted 'ab initio', the time taken to prepare a problem for computation might seriously reduce the effective speed of the machine. [1]

He proposed the pre-computation of important functions that would then be available for lookup. This idea pre-dates the invention of the closed subroutine by Wilkes, Wheeler and Gill [4] but Turing was perfectly aware that understanding problems and their solutions in software:

> ... we regard the preparation of these tables as being of real value since we believe that the successful use of electronic computing equipment will depend more upon the development of an efficient method of organising routines and their integration into large scale computing problems than any other single factor. [1, p.7]

Some 65 years later, we now work in a world in which machine cycles are abundant yet software engineers often struggle to achieve correctness, safety and cost goals. The discipline of software engineering has grown up as an attempt to adapt successful practices from traditional engineering disciplines to the complexities of software, augmenting them with discipline specific notions of re-use and re-targetability.

3. The stratification of software engineering

These days the majority of applications are 'fast enough', and the focus tends to be on programmer productivity and software correctness, rather than on machine performance. The profession of software engineering has stratified into a series of specialisations including analysts, software architects, project managers, specialists such as user interface designers and, at the lowest level, coders. The *Software Engineering Body Of Knowledge* is an ISO technical report [5] which attempts to encapsulate the broad spread of skills possessed by experienced software engineers. The current (third) revision defines 15 modules of which only two (Computing Foundations and Mathematical Foundations) are part of the core of a typical computer science undergraduate programme.

3.1. Design patterns and the OO metaphor

The emphasis on process and abstraction in SWEBOK is is familiar to other areas of engineering, for example construction architects do not lay bricks, much less manufacture them or dig the clay from which they are made. The building industry has long been a fertile source of metaphors for the construction and comprehension of computing systems; a prominent example being the *Design Patterns* [6] movement which began with Christopher Alexander's writings on the pattern language of towns and buildings [7] (itself inspired by programming language grammars) and which directly inspired work in the late 1980's and 1990's on software patterns at algorithmic, coordination and enterprise architecture levels [8].

Object-oriented programming styles coupled to design and architecture patterns have facilitated a move to design-by-composition in which software systems may be assembled from parameterisable modules. There is no doubt that the strict separation of concerns and controlled access to object state has been instrumental in allowing re-use of software, although it is less clear that inheritance (to some, the defining characteristic of OO design) is significant: indeed some authors warn against over-use of inheritance since it exposes some of the internals of a class to its derived classes.

Object orientation as a metaphor is a success, but conventional implementation techniques do not come for free. In most OO languages, objects are created dynamically on the heap, and method despatch is largely specified at runtime. (The obvious counter-example is C++ which retains the static core of C.) This dynamic behaviour is certainly helpful at system level, but the *reductio ad absurdam* as represented by Smalltalk-80's everything-is-an-object doctrine requires even a simple integer value to request heap allocation with a significant memory overhead. The associated execution overheads inevitably limit throughput. In more pragmatic languages such as Java, C# and C++ a spectrum of approaches are provided: in particular the distinction between primitive value types that map efficiently onto the underlying hardware and 'pure' object primitive types. Skilled programmers can thus avoid some of the more obvious traps, but neophytes are often caught out.

An undergraduate student recently presented us with an application that featured a real-time graph of internal activity. Over time, the application's performance slowed to a crawl. The student had, as required, used the Java Swing API method `drawrect()` to draw a 3×3 box for each data point. The student had created an individual `Rectangle` object for each data point, and passed that to the graphics routine rather than directly supplying the coordinates. The resultant stream of object creations crippled the application; removing the redundant `new Rectangle` constructor from the parameter list to `drawRect()` resulted in acceptable performance even though to the student it seemed less object oriented.

Contemporary languages such as Java and C# further obscure runtime performance because Just-In-Time (JIT) compilation and garbage collection modulate the throughput of programs over time; it is difficult to achieve general agreement over the absolute performance of Java programs.

4. A case study: the GLL algorithm

In this section we examine the performance implications of object oriented style of programming on implementations of generalised parsing using standard

Java environment.

Parsing is central to many software engineering tools; the dynamic hints, error checking and sophisticated code refactorings available within the Eclipse IDE rely on accurate parsing of code in parallel with code editing. Most programs which reason about programs, i.e. metaprograms, proceed from the source code and will require parsers. Metaprogramming benefits from general parsing since it frees the user from the hand optimisation of grammars that usually accompanies the use of conventional near-deterministic parser generators such as Bison and ANTLR.

The doyen of metaprogramming tools is ASF+SDF [9], a conditional term rewriting environment coupled to a GLR-style generalised parser, has been used for many applications [10]. GLR is a generalisation of Knuth's LR parsing which exploits Tomita's observation that two concurrent context free parse stacks may be merged when they have the same state on top. This reduces the (potentially exponential) explosion of parse stacks in a general parser to a polynomial bound. The merged stack structure is called a *Graph Structured Stack* (GSS). The (potentially infinite) set of derivation trees resulting from a parse is represented using a *Shared Packed Parse Forest (SPPF).*

More recently a generalisation of recursive descent parsing which uses a GSS to represent the concurrent call graph has been described[11]. GLL runs in worst case cubic time and in linear time on LL(1) grammars. The GLL parser generation algorithm comprises a declarative specification of a collection of grammar attributes defined over a grammar Γ_{EBNF} for extended BNF, and a series of low level control flow templates that are parameterised by these attributes. The grammar Γ_{target} for which a parser is to be generated is internally represented as a derivation tree Δ of Γ_{target} with respect to the grammar Γ_{EBNF}; the parser is a recursive instantiation of the control flow templates that is isomorphic with Γ_{target}.

During execution, a GLL parser maintains a bipartite graph of symbol nodes and pack nodes which represents the binarised SPPF. The out degree of a pack node is less than three. The out degree of a symbol node is bounded by the length of the string being parsed multiplied by the size of the grammar. The GSS comprises nodes representing stack states and edges labelled with SPPF nodes.

The out degree of both GSS and SPF symbol nodes may be very large, but for normal near-deterministic grammars is usually small. To maintain the worst case cubic runtime bound, it must be possible to locate GSS and SPPF nodes with specific labels in unit time, and this presents a significant challenge. One approach is to use a table of all possible elements, implemented using sparse matrix techniques [12] in which table rows are allocated on demand. This technique is impractical for realistic sized problems, since even with sparse optimisations, high memory consumption leads to swapping during the parse.

The alternative is to allocate memory only for those elements that are actually used, and then use indices or hash tables to look them up. The natural object-oriented approach to constructing these data structures defines a class for each kind of element using k individual reference fields to represent *contains k-of*

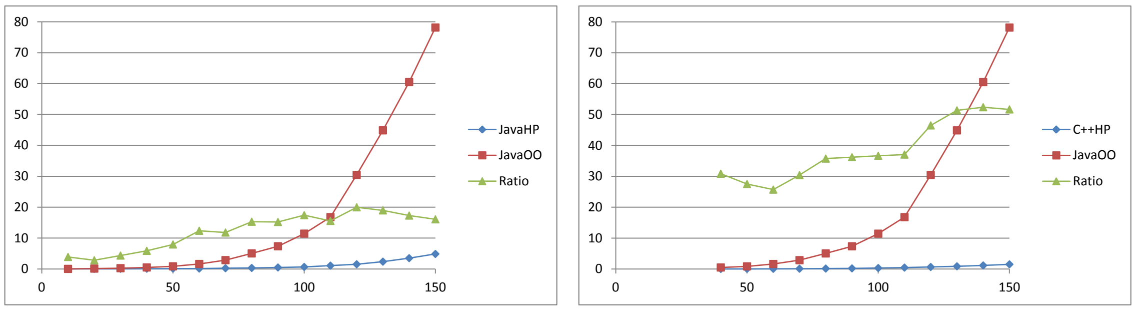

Figure 1: Relative performance of object oriented and HashPool GLL implementations

relationships for small k and some sort of collection class (in Java, the `HashMap` generic) to represent *contains many-of* relationships. Each element of the data structures then becomes a separate object, allocated on the heap. In practice, thisapproach suffers from the same performance overhead as the undergraduate graphics application mentioned above: there may be millions of objects created during a parse, each of which has many bytes of memory overhead. In addition, the intense object creation activity is likely to trigger the garbage collector, even though there will be no memory to reclaim since all nodes created by the algorithm have to remain in memory until the parse completes.

To minimise object creation, we directly manage memory from within the application. In languages such as C++ this is straightforward because we may cast structures onto arrays representing raw memory, but Java militates strongly against such usage. Our *HashPool* implementation maintains a vector of references to arrays of integers (called the *pool*) as the underlying memory model. The vector is initially populated by `null` values except for the first entry which is allocated an array of primitive integers sized so as to contain around 1,000 elements. Allocation simply involves incrementing an index by the size of the allocated element, unless the end of the array has been reached in which case a new primitive array is created in the next vector element. If the vector is full, a new larger vector is allocated and the references copied over. Java does not allow us to directly re-interpret the contents of these arrays as structured types: instead we use indexed addressing and a set of symbolic constant offsets to access individual fields. The resulting programming style is ugly and error prone, since we have effectively removed the safety net of type checking.

Figure 1 summarises the performance of these two approaches using parsers for the grammar $S ::= b \mid SS \mid SSS$, running on strings of $b^n, n = 10, 20, 30, \ldots$. This grammar is highly ambiguous, and in fact triggers the cubic upper bound for the GLL algorithm.

The left hand graph shows performance of the ObjectOriented Java implementation against the Java HashPool support library. The parser code itself is identical in each case; only the management of the internal data structures changes. Run times are in seconds; the ratio of the runtimes is between 10 and

20 for $n > 50$.

We also implemented HashPool in C++ with the results shown in the right hand graph. The performance ratio exceeds 50 for $n > 130$. The C++ implementation is a clerical translation of the Java HashPool implementation without any attempt at further optimisation; for instance, casting a C struct onto an array of integers would allow compile-time constant offset addressing to extract subfields which on some architectures would provide further speed up.

The runtime rations in each case exhibit structure which merits investigation. We believe that these non-linearities arise from the re-hashing behaviour of the Java HashMap; further experiments with pre-sized maps will allow us to test this hypothesis.

5. Conclusions and a proposal: faceted programming languages

The declarative specification of a GLL parser lends itself to reasoning about correctness, but any implementation requires the synthesis of data structures and algorithms that meet the declarative specification. A naïve object oriented implementation has the virtue of conciseness and comprehensibility, but the runtime object creation overhead is uncomfortably high. The HashPool implementation implements the same semantics, and indeed both the declarative specification and the object oriented implementation could be viewed as more abstract specifications for HashPool.

In software we have for a long time been able to trust the compiler to produce correct, efficient machine level implementations of our high language specifications: so much so that assembly level programming is often no longer taught as a primary skill at degree level. In silicon-level hardware design, engineers have historically not been able to rely on efficient synthesis, and hardware description languages (HDL) such as VHDL emphasises the development of multi-level specifications for hardware. Typically these include separate behavioural level, register transfer level and structural level descriptions, each constituting one *facet* of the overall specification. The development environment then provides tools which attempt to establish that all of these descriptions implement the same semantics and meet other *Design Rule Checker* constraints.

The object oriented software engineering required to transform our declarative GLL specification into conventional Java, and the detailed software microengineering required to further transform the implementation into directly memory managed code would have been facilitated if we had had support for facets within Java development and tools to establish their equivalence. At a simple level, Eclipse plugins might provide appropriate tooling and indeed our testing regime for the ART GLL parser generator in a sense provides crude equivalence testing. However, just as with HDLs, we believe that software engineering in general and software microengineering specifically would benefit from integrated programming language designs which allow individual classes to be implemented using declarative set-theoretic specifications, traditional OO metaphors and also as lower level code more redolent of systems-level C, and the existence of clearly defined facet environments would stimulate the development of checkers and synthesizers which would support the kinds of transformations needed for performance optimisation.

References

[1] J. H. Wilkinson, Progress report on the Automatic Computing Engine, Tech. Rep. MA/17/1024, National Physical Laboratory (April 1948).

[2] S. H. Lavington, Early British Computers, Manchester University Press, 1980.

[3] M. Wilkes, Computers then and now, Journal of the ACM 15 (1) (1968) 1–7.

[4] M. V. Wilkes, D. J. Wheeler, S. Gill, Preparation of Programs for an Electronic Digital Computer, Addison-Wesley, 1951.

[5] Software engineering – guide to the software engineering body of knowledge (SWEBOK), Tech. Rep. ISO/IEC TR 19759:2005, International Standard Organisation (2005).

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, 1994.

[7] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language, Oxford University Press, 1977.

[8] G. Hohpe, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison Wesley, 2004.

[9] M. van den Brand, J. Heering, P. Klint, P. Olivier, Compiling language definitions: the ASF+SDF compiler, ACM Transactions on Programming Languages and Systems 24 (4) (2002) 334–368.

[10] M. van den Brand, Applications of the ASF + SDF meta-environment, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, Vol. 4143 of Lecture Notes in Computer Science, Springer, 2005, pp. 278–296.

[11] E. Scott, A. Johnstone, Gll parse-tree generation, Science of Computer Programming.

[12] A. Johnstone, E. Scott, Modelling GLL parser implementations, in: B. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers, Lecture Notes in Computer Science, 2011, pp. 42–61.