Model-free Optimization

Gio Wiederhold, DARPA SISTO

April 1992
published in: *DARPA Software Technology Conference* 1992
Meridien Corp., Arlington VA, pages 83-96.

Abstract
We consider how programming-in-the-large differs from programming-in-the-small. We then focus on resolving a critical conflict that arises when building large software systems. Large systems have both the requirement to be adequately efficient in execution and sufficiently flexible to be maintainable in a changing world. For our purpose their scale is assumed to be such that multiple processors, multiple applications, and multiple types of software modules will be simultaneously active. After a general background discussion, two critical points are elucidated:

1 The high level application process must obtain performance estimates as well as results from lower modules to create efficient schedules for each application type.
2 There will be no single optimization model or overall schedule, but this task will be as distributed and parallelized as the computations themselves.

The general conclusion is that higher level abstractions are needed in all aspects of software technology to overcome the problems of programming-in-the-large and to use modern hardware technology effectively.

1. Background

The 1992 DoD Software Technology Strategy (SWTS) emphasizes the importance of reuse of software. To achieve the goals '... a considerable challenge remains relating to validation and acceptance of assets developed outside the control of the user organization' [SWTP:92, 4.1.4]. Techniques and standards for specification, documentation, integration, and assurance have to become more effective.

Techniques which are successful in moderate scale software have not, or not yet, been able to demonstrate their effectiveness in large-scale projects. Top-down decomposition fails when large systems must satisfy multiple, often conflicting objectives. Careful system design prior to implementation fails when the time needed to complete the implementation exceeds the interval in which the environment and requirements change. Multi-layered approaches using the virtual machine paradigm show major performance deficits, so that the layers are often pierced in practice. Object-oriented programming has not yet proven scalability to mega-objects, probably because of the model problem, as elucidated through most of the remainder of this paper. There also remains the problem of mixed paradigms:

1 Objects as fully independent objects, accessible only via methods; an effective view when modeling information processing systems : the SMALLTALK approach.
2 Objects as smart data structures, manipulable externally; an effective view for the design community: the PDES approach.

Without new languages, handling objects of type 1 induces great overhead for small objects. Methods to obtain fields from records, or worse, values from constants, cannot compete with effective code sequences created by compilers. For engineering design systems, at least, it appeared that minimal implemented objects should start at a scale which is beyond the level handled effectively by compiled procedures [WWN:89].

It seems difficult to design languages and compilers which can span the full range of atomic object to mega-object management. Furthermore, by placing all work into one language model, we reduce accessibility to new programming notions, as those based on declarative approaches, constraint-based code-generation, and also limit reuse of proven 'Commercial-of-the-shelf' modules.

1.1 Megaprogramming

We hence focus now on composition at the large object level. Megaprogramming is defined as software development by composing components rather than individual lines of code. The need for effective assembly of large software applications has been frequently elucidated. We have successful experience in the use of subroutines, and, in some cases, as numerical and statistical libraries, quite large sub-modules. However, these successes have not been easy to generalize.

1.1.1 Precursors

Numerical and statistical packages, with high-level glue languages are well established and in general use, and embedded in many packages as the SPSS, BMDP, LINPACK, a.o., [Brown:83], [Catarci:90], [Press:86], [SPSS:75]. A likely hypothesis for their success is that numerical software is easy to specify, since it builds on a long tradition of formal specification. A contributing, positive factor in numeric scientific processing is that the datastructures in numerical codes use are simple and regular. With regular structures scalability can be achieved by simply increasing the cardinality and dimensionality of the arrays representing such data. However, in multi-dimensional problems the demands for memory and cycles increase more rapidly (by $\mathcal{O}(n^{dim})$) than even our rapid technical advances in hardware capacity ($\mathcal{O}(n^2/year)$ [HPCC:91]. More recently, scientific computation has started to deal with scale by use of more complex representations. Successive partitioning of spatial areas where discontinuities occur and the imposition of complex grids to represent surfaces are logically more sensible, but their complexity introduces new problems [Olig:91].

Non-numeric software has had to deal with a greater variety of data representations since its inception. Records, lists, trees, and graphs represent primitive concepts used to compose larger structures. Automatic programming systems recognize these primitives, and can provide mappings [Smith:90]. But the associated semantics are hard to specify, and assuring the correctness of transforms and analyses demands processable specifications.

Databases have also achieved success in scaling their collections to significant size. Here again the regularity of their structures and the availability of an algebra permits transforms with predictable results. Standards as SQL have made both structure and operation specification commonly available, although not easy to use. The simple structures only provide for the representation of simple functional dependencies, while most programs must deal with relationships among complex objects. The actual processing semantics for such databases are provided in the users' programs. There is a need for having stuctural constraints implicitly placed into the structure as dependencies or explicitly provided as meta-data [Stonebraker:86]. A number of extensions which are intended to handle more complex datatypes are being developed, but it is obvious that these stress the simple relational infrastructure.

Where software modules can be well specified, composition has become feasible. Programs under UNIX are linearly composable if they absorb and produce a character sequence. Many numeric routines are composable if they produce numeric matrices. The Xerox IDL package provides composable statistical processing primitives. The GENESIS program [Batory:89] defines an assembly mechanism for database systems. Assembly of image-processing modules is provided for rapid configuration of planning experiments [Hunt:90].

In all these systems the user here is assumed to know the semantics of the modules, and the data structures are prespecified to match well. Because the number of modules is limited the user can be expected to have a good conceptual understanding of their semantics. Scalability to hundreds of candidate modules is unlikely. When only a limited range of compositions is valid, it is feasible to check for syntactic correctness of compositions. Such checking is inhibited when users are able to interpose applicaction-specific code. The use of glue code is frequent when datastructures do not match fully, when the application demands filtering of information, and in general when modules written in a different context are being reused.

1.1.2 The Vision

To support more general notions of composition, interface formalisms have to be provided [Callahan:91]. These interfaces need machine-processable specifications, to help in the composition task. The use of formal interfaces is especially critical for reconfiguration and redistribution of software modules [Purtilo:92]. Often multiple processors will be involved. The actual composition of modules is likely to still be carried out by some programmer, but the code to be generated consists of linkages to these interfaces, some glue code to adapt mismatched interfaces, invocations of display functions on the users' workstations, and means to deal with the asynchrony of execution of the computing modules in such a system [Friedman:86] .

1.1.3 Languages

It is obvious that we do not want our programmers to actually write the code to connect modules — it is essential that bindings be symbolic and that the mappings to actual processors and code be handled by a compiler. The language needed (we'll call it M here) does not exist, although we find partial models [Cannata:92]. The language should be compilable to avoid high overhead costs of initiating computations in multi-processor situations. It must be easily recompilable to deal with configuration changes.

Procedure-oriented languages are fine for writing the modules, but do not deal well with externally specified interface structures. So-called 4GLs do **not** provide a model, since they are typically domain-specific and oriented towards being user-friendly, by incorporating domain knowledge. Some of the modules, especially those performing user interaction, may be well served by such a 4GL, but the language we need must support composition of these modules in a domain-independent manner.

The M languages will use the formal interfaces as their atomic units, rather than the access functions for records, arrays, and individual values in local memory of procedural languages [WWC:90]. Complementing the languages must be the objects for composition: large libraries of reusable software. Allocation of storage in memory is rescaled to allocation over computing networks. Partitioning and managing information flow, i.e., *transduction* of data among computing nodes, are critical issues here.

We find then that there are issues in programming-in-the-large that differ fundamentally from programming-in-the-small, even more than football differs from table tennis. There are more players, more space, there is more specialization, and the amount of money involved changes the way the business is performed. Unfortunately, in an academic environment programming-in-the-large has been hard to present, while industry, working without concepts and tools, has been forced to depend on brute force approaches to deal with the issues. In the next section we present some problems to be addressed when moving to megaprogramming. By understanding some specific issues better, we hope that the family of associated problems will be attacked and become manageable. Having languages for large scale composition should also improve academic access to these problems, and enable dissemination of any research results.

The remainder of this paper addresses only two related issues from the gamut of issues in large scale software: model-free performance estimation and distributed scheduling. These alone do not solve the problem, but issues of performance have been given less attention than many other issues in establishing megaprogramming. Many conceptual advances will be discarded if megaprogramming cannot achieve credible performance.

2. The Modeling Problem

We have to learn to deal with structures that are variously called *large objects*, *reusable library programs*, *mega modules*, *domain-specific software modules*, etc. Although the concepts represented by these notions do not fully overlap, we want to address here an issue common to all, and will refer to such entities as *mega-objects*. As the mega-objects grow in size and scope it becomes difficult and not rational to require that the programmer assembling them be able to understand their internals. Many programming philosophies, in fact, take the paternalistic view that the programmers should not be able to inspect the internal workings, lest they be tempted to take advantage of internal structure, and thus commit the system to retaining this structure. A similar visibility barrier should exist when such mega-objects are used by automatic programming schemes.

However, without having a model of the internal structure, it becomes impossible to assemble these modules into an effective structure. Take for example the task of finding reviewers for ACM TODS [Wetal:89], although the performance problem was not addressed there. The top level application sentence defining this task is:

```
'Find-reviewers':
   Find competent reviewers for a submitted paper on topics x,y,z,
            who are not friends of the authors, and who are responsive.
```

This is not yet much of a mega-program, but ontologically sufficiently complex to elucidate the problem. The subtasks needed to solve this sentence are

1 A competency program module, '`Provost`', which assigns values to books, journal papers, etc., sums theses values per person object, and ranks these objects. The data used are an author-based view on a large bibliographic database.

2 A subject match module, '`Expert`', which takes the keywords, locates them in the acyclic graph (DAG) of subjects provided by the ACM COMPUTING REVIEWS (expanded in the database topic area). Finding more subsumed or general terms in the DAG provides keywords which will retrieve better or more entries when searching another view on the same bibliographic database [Chaudhuri:90]

3 We can't compute friends due to absence of actual data, but can invoke a '`Colocation`' module, which uses temporal reasoning over intervals in order to find overlapping periods of people, authors and reviewers, at institutions [WJL: 91]. Here

another, publication-based view on the bibliographic database is used. The application makes the pragmatic jump of considering colocated people as potential friends.

4 The '`Responsiveness`' module, given a list of people, checks my email log for matching messages, taking into account some of the vagaries of email addresses.

We see that in order to process this sentence, we invoke four distinct modules, all related to a subexpression in the source statement. These modules use different computational methods, and some of these have been sufficiently problematic that they led to distinct publications. The modules, in turn, use three distinct databases, or more relevant, five database views. A highly integrated program to solve the application sentence would need to be written and maintained by a committee of experts.

2.1 Understanding Models and Performance

However, an essential problem has not been addressed. The programmer who actually wrote the '`Find-reviewer`' application understood the underlying modules sufficiently well that she invoked the modules in order '`Expert`', '`Provost`', '`Colocation`', '`Responsiveness`'. The '`Expert`' module is chosen initially because it provides the best partitioning of the search space, and has methods to do so effectively. The '`Provost`' module cannot provide a useful partitioning, unless a competency limit is given. It would return a a very large result, as would the other modules.

But if the same modules were used to find the best faculty candidate in a scientific area, then the '`Provost`' module is likely to provide the best initial partitioning. The '`Expert`' module would return too many candidates per academic area. We model again mentally the invoked methods and our estimates of the sizes of underlying database to produce a reasonable schedule.

2.2 Optimization Principles

A major task of an optimizer is *rearranging* primitive components. In programming languages code is moved out of loops, constant expressions are precomputed, and subroutine invocation is bound. Optimizers have a formalized model of the time taken for execution time of a component and estimate how often it needs to be invoked under certain conditions. The execution time may depend on cardinality information or estimates. Based on this knowledge, an optimal schedule is produced. The base concepts for arranging modules for optimal execution do not differ from those used in programming languages, databases [WKNSBS:86], [Stonebraker:86], and artificial intelligence processing [Smith:85].

2.3 Sequencing Mega-objects

A crucial problem in scaling software and keeping it efficient is that optimizers depend on models of the computations being performed. The models are evaluated with parameters being extracted or estimated, as the count of loop executions, the selectivity of a database predicate, the partitioning of the search space.

However, our Mega-objects, as other objects, should hide methods and cardinality information internally. It is argued, in fact, that it is dangerous if the programmer knows the internals [Bertino:92]. We can substantiate this argument with the example above. If the programmer's model is that it is harder to find responsive reviewers than expert reviewers, she might have placed the modules in an alternate sequence, and invoked the '`Responsiveness`' module, checking for email correspondents first. Unfortunately, such poor sequences can change computation costs by factors of many thousands [Swami:89].

2.4 Distributing Execution

Large programs will also execute at multiple nodes within our computer networks. A parallel schedule has more freedom than a linear one. It can create execution graphs subject to dependency constraints [Qian:87] [RSK:91] . Overlap of execution is a further major factor to achieve good performance [Bodorik:88]. Dealing with objects adds some further complexity [Härder:88], [Kim:90].

Here we begin to exceed the likely competency of any programmer. Envisaging efficient parallel schedules in intelligent systems is yet more complex [Cohen:89]. For our TODS example it seems likely that a schedule: '`Expert`',{'`Provost`', '`Colocation`', '`Responsiveness`'} is optimal, since the subset produced by '`Expert`' can be used by the other modules in parallel. But we were dealing with a toy problem, and ignore performance hits due to parallel access of shared database resources by '`Provost`' and '`Colocation`'.

2.5 Tentative Executions

In a resource-rich environment competing modules can be initiated, and the best-performing chosen, while poor performers are aborted [Roy:91]. We might have an alternate source for competency than the '`Provost`', say '`Income`' which ranks the salaries of the candidate reviewers. This assumes some faith in the compensation system, but the paper-production measure also has its detractors. The programmer's model is stressed even more when alternates are feasible.

2.6 Flexibility of Schedules

Having modules is motivated by the ability to replace modules as computing science provides improvements. The data resources used by the modules also change; they often grow by leaps and bounds. Sometimes r esponse times change little with data volume, sometimes they grow exponentially. The technology tradeoffs in networks also change as advances in communication and computation jockey for position.

The advantage of modularity in software is lost if we fix schedules based on initial assumptions. Rearranging modules during production for the sake of efficiency is rarely done, since subsequent programmers cannot easily induce which arrangements are due to computational dependencies and constraints, and which were motivated by performance models.

2.7 Hence: Automation of Scheduling

When building large-scale systems we cannot and should not expect the programmer to provide the model of all the underlying modules. When maintaining large-scale systems nobody tracks growth and changes to determine when and when performance is affected, at least until it has become a disaster. If we have a programming language to handle high-level statements, as given in our example sentence, we can envisage having a compiler for the task or arranging modules optimally. However, our compilers, when employing rearranging techniques, depend on a formal model of the underlying methods, as well as on frequency and size parameters, to produce good schedules.

Such models are not available when mega-objects are used or reused. Furthermore, jointly optimizing the execution sequence of modules which use a variety of computational techniques cannot be driven by a single model. For our example application we would have had to build an optimizer which understands ranking in the '`Provost`' module, pruning by generalization and specialization in the '`Competency`' module, temporal reasoning for '`Colocation`', and email matching for '`Responsiveness`'. But even that optimizer will be baffled when we add the '`Income`' module to our scheme.

3. Model-free Scheduling

We now come to essential argument of this note: scalability of software, while retaining efficiency of execution, requires *model free optimization*. The efficiency of modules is not the only condition for having efficient systems. An application will be composed of many modules and the sequence and overlap of their execution is the major factor to achieving good performance.

Large systems employ mega-objects to gain a higher level of abstraction in semantics. For instance, to avoid friends I only need to understand '`Colocation`', but not the temporal reasoning within the server module needed to produce instances of colocated persons.. The use of abstraction cannot be limited to the function of the modules, it transcends all aspects.

An optimizer in this world must also work at a higher level of abstraction. What are the needed abstractions here?

We propose, that instead of obtaining base parameters for its model from the lower layers, this optimizer must import estimates of performance, that do not depend on understanding the type of computation below. Such estimates are best composed of expected values and their variance or uncertainty, so that the high level optimizer can also judge risk [WWC:91].

The expected value and variance is applied to two parameters: cost within the module and result size, leading to a quadruple, as shown in Fig. 1.

`Estimate`	Mean	Variance
Time Cost	$\overline{cost}$	$sd(cost)$
Result Cardinality	$\overline{size}$	$sd(size)$

Figure 1. Estimates to be produced by mega-objects.

The essential point to be generalized here is then: to build integrated systems with high-level functionality requires that the support functions that enable efficient processing must also be moved to that higher level.

For objects that implement multiple methods, each method must produce corresponding estimates. This approach doubles then the number of entry points to an object, or requires that each method has an additional associated parameter: `Compute/Estimate`. We will see below that the latter choice provides more flexibility.

4. Multiple Schedulers

Another important point must be made explicitly now: there is not a single hierarchy in such a computing architecture. The set of available program and information resources and their potential relationships forms a network. For any application a different set of modules can be assembled.

Typically an application is partitioned top-down into a hierarchy that represents a decomposition of its specific functional objectives. The methods needed to support the objectives determine the selection of modules. The information flow specifies their invocation schedule [Collet:91]. For each application very different schedules will optimize the invocation of these modules. Although the objective may be identical, the set of modules and their invocation schedule will differ.

4.1 Diversity of Performance Objectives

Optimization objectives are not the same for distinct applications. Some application must give more weight to flexibility, and others to performance. The performance measure for some applications may be minimal response time, and for others low resource usage.

A scheduler may be able to insert certain modules solely to obtain faster responses, for instance, modules that perform prefetching or caching. Caching can be further improved by not only keeping source data, but by also precomputing and storing intermediate results at higher levels of abstractions. Such modules consume memory and storage resources to gain performance. The extent of prebinding will differ as well; if the environment is stable the communication paths can be fixed, rather than dynamically evaluated.

Real-time constraints can require drastic changes in optimization strategies. To gain predictability a real-time scheduler will invoke alternate modules having estimates with lower variance, even if their mean performance is worse. Modules specifically created to better serve real-time needs may be available. Some such modules may create results which are less less precise, but obtained faster. Especially time-based simulations may produce simple results which are less precise when the time horizon is farther away.

It is clear that a single scheduler that can achieve the full range of possible performance objectives will be excessively complex. Again we return to the theme of modularity of services: We expect that there will be multiple schedulers available, and that an application choses one of them, namely the one that is likely to satisfy its objectives best and at the least cost. The application client now can consider performance as an objective on the same abstraction level as functional requirements.

4.2 Distributed Estimation

Our proposed schedulers can live at many nodes. Model-free schedulers are not affected by having new methods or entire computational nodes inserted into the set of system resources. Multiple layers of services will appear in such an architecture, and estimates as well as data will be passed through the layers. Access to the lower layers is often needed to obtain cardinality estimates. But if aggregations are produced at intermediate layers, cardinality estimation may be possible without low-level accesses.

Overall this architecture avoids centralization. Individual applications are optimized for their own performance objectives. Lower level service modules may serve multiple applications simultaneously. Their estimates, if obtained at execution time, can reflect reduced performance due to the committed load, so that the application schedulers can avoid overloaded modules, if functionally possible.

Although limiting the optimization to individual applications is likely suboptimal in terms of total resource consumption, there does not appear to be an important conflict. Only excessive locking of resources to benefit a specific application must be avoided and perhaps controlled.

4.3 Replication

Many types of computational modules can be replicated, if access to the input data is easy. Replication can even be beneficial within one application. For instance multiple simulations can perform what-if analyses at several time-windows or for several assumptions.

Database server modules cannot be replicated easily, so here it is important to separate computational and retrieval functions, so that their performance remains high. This observation argues for keeping database systems simple, and allocating advanced analyses to distinct modules.

4.4 Cost of model-free optimization

The overhead for generating estimates and transmitting will be relatively low for large server tasks. However, for small tasks, returning even a simple estimate can be as costly as returning the results. But the model-free scheduler cannot know in advance the magnitude of the task. We place that decision on the server as well, by providing two options for responding to an `Estimate` request.

A critical shortcut

To address that problem, we propose a shortcut. For large tasks, only the four values of the estimate are returned. For small tasks, the actual results are returned with the estimates. The boundary of large and small tasks can be made seamless.

To permit a server module to make a realistic, low-variance estimate, its input will be specified to the best extent possible. This means that no new distinct interaction may be needed to supply input in order to compute the result once the input for the estimate is given. For small tasks, having complete input is quite likely.

Early availability of input enables the shortcut and seamlessness. When the cost of computation is low, the final result can be returned immediately. Low means here less than the overhead cost of invoking the module later again with a `Compute` entry. If the result is immediately returned the associated cost estimate is precisely zero. Both variances are zero too.

This approach can keep the estimation overhead modest, while providing results already during the optimization phase, reducing subsequent costs. No special distinction need be made for small versus costly execution modules. We have now a seamless scheme for small and large server tasks.

4.5 Blurring Optimization and Execution

Since some execution is initiated during optimization, the traditional boundary of having an optimization and an execution phase disappears.

To what extent optimization is best carried out as a precompilation phase, without known inputs, is unclear and awaits experience. If the cost of scheduling and composing large modules can remain relatively small then execution-time optimization can be tolerated. On parallel machines, in fact, multiple candidate schedules can be started, and the less successful ones, when evident, aborted [Roy:91].

5. Discussion

We will now summarize the notions presented here, and place them in a broader context. Then we identify areas where further research seems needed.

5.1 The Computational Environment

This model of computation assumes a resource-rich environment, as envisaged in [HPCC:91]. Responsiveness of applications is more important than efficient use of computing cycles. There is no global optimization over all the modules, outside of the adaptation caused by applications rejecting use of overloaded modules. The optimization is oriented towards application performance, the critical issue in large-scale software. The estimation process envisaged can also serve real-time needs. Satisfaction of real-time constraints requires a resource-rich environment, but provides opportunity for economical execution of less demanding tasks.

5.2 Alternative Architectures

There are alternative architectures which address the goals of having efficient large-scale programs in networks. Some of these complement the notions presented here.

ALGRES extends SQL to provide intermediate modules using main-memory data structures, which are loaded from mass-memory storage in an external database using explicit statements [Ceri:90].

The Starburst Project recognizes the need for its optimizer to be extensible to new methods [Widom:91]. It provides this extensibility by letting the implementor of new operations also add rules pertaining to those methods. However, in a world where new methods are effectively remote — encapsulated in objects, located on remote nodes — placing the information for optimization at a high-level central node is not appropriate. Furthermore, since distinct applications will use different initial nodes, one cannot even identify such a central node. Replication of optimization information is, of course, feasible, but that means that nodes will be loaded with optimization rules and models for component nodes they might never use.

A broader view is taken by people working in Enterprise Integration [Collett:91]. The CARNOT project at MCC using a single knowledge-base, within CYC to manage diverse resources [Cannata:92] . Rules in CYC resolve semantic mismatches. A development of an existing language for parallel processing, ROSETTE, is used to define the application workflow. Since the optimizer cannot access the resources, it produces estimates of data volumes internally, and minimizes expected data transmission costs only. It creates scripts for each node, so that transduction of information is direct between nodes, avoiding a central communication load.

Domain-specific architectures focus on modules which incorporate domain semantics. They reduce the need for programmers redesigning common structures. Domain knowledge can also help in producing estimates which are appropriate, relieving the module programmer of repeating that task.

Associated with some standards efforts is research in application composition from large objects [OMG:91], [KQML:92]. Unfortunately, there is inadequate experience to have confidence in potential standards. Implementations and demonstrations are needed. These must be scaled up to gain confidence in applicability to mega-programming.

5.3 The Research Tasks

The critical element missing today in a software architecture for dealing with mega-objects is the high-level language needed to compose applications. The composition task can be greatly aided by having tools as libraries with formal specifications, and graphical tools for the selection and insertion of modules [Hunt:90] .

To create a library of such modules we can initially wrap existing programs to serve within this architecture [Callahan:91]. The wrapper will have to provide an interface specification for linkage as well as provided the estimates for module scheduling. Human interface code in old programs, needed to make them user-friendly can be omitted, greatly reducing their complexity. We foresee that the input-output code will move to the end-users' workstations, which will then communicate with the root nodes of the application tasks [Green:91]. The workstations can adapt to the ever-changing user-interface paradigms, greatly reducing the maintenance demands on computational modules.

Scalability of methods does not only require implementation and testing, but also underlying models, so that extrapolations and new combinations of components will have predictable performance.

6. Conclusion

We are envisaging today many large computing systems. Building the *best* integrated system today is foolhardy. First we have to assemble a great many domain-experts, and link their work through defined interfaces. Components and interfaces become obsolete before a system is implemented, requiring continuing efforts. Once the system is produced it will require cooperation of many experts to maintain it afterwards. The task of restarting fresh with the old approaches is uninspiring and risky. We have too many systems operating today which represent the *best* technology of 20 years ago.

Modularity and reuse is an essential direction. Individual modules of large integrated systems are unlikely to be reusable, unless they satisfy simple and formal specifications. Most existing modules depend on their environments; and the environments depend on their modules. Only if we have a partitionable and evolvable software architecture will we able to achieve having large and maintainable systems with the limited number and great diversity of individual contributors we have available. Model-free module scheduling is one aspect of such a flexible and adaptable software environment.

This note has focused on performance issues in mega-programming and provided a direction for its resolution. The proposed solution involves the use of an abstraction of performance measures commensurate with the abstraction scale of mega-objects. We believe that the same paradigm shift is needed on all aspects of large-scale software, including functional specification, data structure description, interface formalisms, intermodule constraint handling, compilation, and transaction management.

Acknowledgement

I thank Robert Molter for thoughtful reading of an earlier draft of this paper. Barry Boehm helped in formulating my thoughts through mutual discussions.

References

Note that due to my background these references are oriented towards the database literature.

[Batory:88] D.S. Batory et al.: "GENESIS: An Extensible Database Management System" *IEEE TSE*, Vol.14 No.11, Nov.1988, pp.1711–1730.

[Bertino:92] E. Bertino: "Data Hiding and Security in Object-Oriented Databases"; *IEEE Data Engineering Conf. 8*, Tempe, AZ, Feb.1992.

[Bodorik:88] P. Bodorik and J.S. Riordon: "Distributed Query Processing Optimization Objectives"; *IEEE Data Engineering Conf. 4*, Feb.1988, Los Angeles.

[Brown:83] Brown,V.A., Navathe,S.B., and Su,S.Y.W.: "Complex Data Types, and a Data Manipulation Language for Scientific and Statistical Databases"; *Proc. of the Second International Workshop on Statistical Database Management*, Menlo Park CA, Sep.1983, pp.188–195.

[Callahan:91] J. Callahan and J. Purtilo: "A Packaging System for Heterogeneous Execution Environments"; *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp.626–635.

[Cannata:92] Philip E. Cannata, Darell Woelk, Wei-Min Shen, and Christine Tomlinson: "Query Translation and Optimization for Enterprise Information Integration"; *1992 Workshop on Heterogenous Databases and Semantic Interoperability*, Feb.1992, Pam Drew and Marek Rusinkiewicz (editors), US West Advanced Technologies, Boulder CO.

[Catarci:90] Catarci,T., D'Angiolini,G., and Lenzerini,M.: "Concept Description Language for Statistical Data Modelling"; *VLDB* 16, Brisbane, Aug.1990, Morgan Kaufman, pubs.

[Ceri:90] Stefano Ceri, S. Crespi, R. Zicari, G. Lamperti, and L. Lavazza: "ALGRES: an advanced database system for complex applications"; *IEEE Software*, Jul.1990.

[Cohen:89] S. Cohen and Ouri Wolfson: "Why a Single Parallelization Strategy is Not Enough in Knowledge Bases"; *ACM Principles of Databases Symp.*, Philadelphia, Mar.1989.

[Collett:91] C. Collett, M. Huhns, and Wei-Min Shen: "Resource Integration Using a Large Knowledge Base in CARNOT"; *IEEE Computer*, Vol.24 No.12, Dec.1991, pp.55-62.

[Green:91] M. Green and R. Jacob: "Software Architectures and Metaphors for Non-WIMP User-Interfaces"; *Computer Graphics*, Vol.25 No.3, Jul.1991, pp.119–235.

[Friedman:86] Fred Friedman, Arthur M. Keller, Gio Wiederhold, Mike R. Berkowitz, John Salasin, and David L. Spooner: "Reference Model for ADA Interfaces to Database Management Systems"; *IEEE Computer Society Data Engineering Conference 2*, Los Angeles, CA, February 1986.

[Härder:88] Thomas Härder, H. Schoning, and A. Sikeler: "Parallelism in Processing Queries on Complex Objects"; *Internat. Symp. on Parallel and Distributed Systems*, IEEE CS, Austin TX, Dec.1988.

[HPCC:91] Committee on Physical, Mathematical, and Engineering Sciences *Grand Challenges: High Performance Computing and Communications*: Supplement to President's FY 1992 Budget, 1991.

[Hunt:90] Neil Hunt: "IDF, a Graphical Dataflow Processing Language for Image Processing"; *Proc. IEEE Conf. on Systems, Man and Cybernetics*, IEEE, Nov.1990.

[Smith:90] D.R. Smith: "KIDS: A Semi-Automated Program Development System"; *IEEE Transactions on Software Engineering*, Vol.16, No.9, Special Issue on Formal Methods, September 1990, pp. 1024–1043.

[Kim:90] K-C. Kim: "Parallelism in Object-Oriented Query Processing"; IEEE Data Engineering Conference 6, Feb.1990.

[Khoshafian:88] S. Khoshafian,P. Valduriez, and G. Copeland: "Parallel Query Processing for Complex Objects"; *IEEE Data Engineering Conference 4*, Feb.1988, (Los Angeles).

[KQML:92] Bob Neches, manager, KRSS standards group: "A Knowledge Query and Manipulation Language"; Information Sciences Institute, Marina delRey CA, draft to appear Mar.1992 .

[Lee:90] Byung Suk Lee: *Efficiency in Instantiating Objects from Relational Databases through Views*: PhD Thesis; Stanford Report CS-90-1346, Dec.1990.

[Neches:91] Robert Neches, Richard Fikes, Tim Finin, Tom Gruber, Ramesh Patil, Ted Senator, and William R. Swartout: "Enabling Technologies for Knowledge Sharing'; AI Magazine, Vol.12 no.3, Fall 1991.

[Olig91] Joseph Oliger: "A system and Language Environment for Scientific Computing"; Classic Project Manuscript CLaSSiC-91-27, Computer Science Department, Stanford University ,1991.

[OMG:91] The Object Management Group: "The Common Object Request Broker, Architecture and Specification"; OMG Doc. 91.8.1, Aug.1991, DEC, HP, Hyperdesk Co., NCR, Object Design, Sunsoft.

[Press:86] Press,W.H., Flannery,B.P., Teukolsky,S.A., and Vetterling,W.T.: *Numerical Recipes, The Art of Scientific Computing*: Cambridge Un.Press, 1986.

[Purtilo:92] J. Purtilo and E. White: "A flexible program adaptation system: case studies in ADA; to appear in *Journal of Systems and Software*: Univ. of Maryland, CSD Tech Report 2522.

[Qian:87] X-L. Qian and D.R. Smith: "Reformulation : An Approach to EFficient Constraint Validation"; VLDB 13, Brighton, UK, Sep.1987.

[Roy:91] Shaibal Roy: *Parallel Execution of Database Queries* PhD Thesis, Stanford, CSD, CS-91-1397, Aug.1991.

[RSK:91] Marek Rusinkiewicz, Amit Sheth, George Karabatis: "Specifying Interdatabase Dependencies in a Multidatabase Environment"; *IEEE Computer*, Dec. 1991, pages 46–53.

[Smith:85] D.E. Smith and M.R. Genesereth: "Ordering Conjunctive Queries"; *Artificial Intelligence*, Vol.26, 1985, pp.171–215.

[Stonebraker:86] Michael Stonebraker(editor): *The* INGRES *papers: Anatomy of a Relational Database system*: Addison-Wesley, 1986.

[SPSS:75] Nie,N., Hull,C.H., Jenkins,J.G., Steinbrenner,K., and Bent,D.H.: "*Statistical Package for Social Sciences*: McGraw-Hill, 1975.

[Swami:89] Arun Swami and Anoop Gupta: "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques"; *ACM SIGMOD 89*, (Portland OR), May.-Jun.1989, pp.367–376.

[SWTP:92] Director of Defense Research and Engineering: *Software Technology Strategy*: Department of Defense, Draft, Dec. 1991,

[Wiederhold:91] Gio Wiederhold: "The Roles of Artificial Intelligence in Information Systems"; *Methodologies for Intelligent Systems*, Ras (ed): Lecture Notes in Artificial intelligence, Springer Verlag, 1991, pp.38-51

[Wiederhold:92] Gio Wiederhold: "Mediators in the Architecture of Future Information Systems"; *IEEE Computer*, March 1992, pp.38–49.

[WJL: 91] Gio Wiederhold, Sushil Jajodia, and Witold Litwin:"Dealing with Granularity of Time in Temporal Databases"; in R. Anderson et al., eds: *Lecture Notes in Computer Science*, Vol. 498, Springer-Verlag, 1991, pp.124–140.

[Widom:91] J. Widom, R.J. Cochrane, and B.G. Lindsay "Implementing Set-Oriented Production Rules as an Extension to Starburst" *VLDB 91*, Barcelona, Sept.1991., Morgan Kaufman, pubs.

[WKNSBS:86] Wiederhold, Arthur M. Keller, Sham Navathe, David Spooner, Murray Berkowitz, Bill Brykczynski, and John Salasin: "Modularization of an Ada Database System"; *Proc. International Pre-VLDB Symposium*, Beijing PRC, S.B.Yao,S-H.Sa,Z-S.Zhong(eds), China Assoc.for Science and Technology, Aug. 1986, pp.202-232; extended abstract in Proc. 6th Advanced Database Symposium, Information Processing Society of Japan, Aug. 1986, pp.135-142.

[WRBLQ:90] G. Wiederhold, P. Rathmann, T. Barsalou, B-S. Lee, and D. Quass: "Partitioning and Combining Knowledge"; *Information Systems*, Vol.15 no.1, 1990, pp. 61–72.

[WWC:90] Gio Wiederhold, Peter Wegner, and Stefano Ceri; *Towards Megaprogramming*: Stanford Un. Report CS-90-1341 and Brown Un. report 90-20, Oct.1990; accepted for *Comm. ACM.*

[Wetal:89] Gio CM Wiederhold, Michael G. Walker, Waqar Hasan, Surajit Chaudhuri, Arun Swami, Sang K. Cha, XiaoLei Qian, Marianne Winslett, Linda DeMichiel, and Peter K. Rathmann: "KSYS: An Architecture for Integrating Databases and Knowledge Bases"; in Amar Gupta (ed.) *Integration of Information Systems: Bridging Heterogenous Databases*', IEEE Press, 1989, pp.72–112.

[WWN:89] Gio Wiederhold, Marianne Winslett, and Lt Nicholas Naclerio: "Layering an Engineering Information System"; *Digest of Papers, IEEE CS Spring COMPCON 34*, February-March 1989, pp.444–449.