©2010 International Journal of Computer Applications (0975 - 8887)

Volume 1 – No. 16

Designing Compilation Server

Mrs. Mrudula S. Nimbarte
Lecturer
Bapurao Deshmukh College of Engineering,
Sevagram, Wardha, 442102

Prof. A. P. Bodkhe
Professor and Head
Information Technology Department
Prof. Ram Meghe Institute of Technology and
Research,Badnera

ABSTRACT

In language systems that support separate compilation, the header files are internalized over and over again when the source files that depend on them are compiled. Making a compiler a long-lived server eliminates such redundant processing of header files, thus reducing the compilation time. Modern JVM implementations interleave execution with compilation of “hot” methods to achieve reasonable performance. Since compilation overhead impacts the execution time of the application and induces run-time pauses, it is better to offload compilation onto a compilation server. Compilation server is the server which compiles and optimizes Java byte codes on behalf of its clients. It provides the benefit of lower execution and pause times due to reducing the overhead of optimization. Compilation server is able to handle more than 50 concurrent clients while still allowing them to outperform best performing adaptive configuration.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Processors - Compilers

General Terms

Design, Languages, Theory.

Keywords

Compilers, Compilation Server, JVM, execution time, pause time

1. INTRODUCTION

Modern programming systems based on compilers support the notion of *separate compilation*. A program in such systems consists of *source files*, which are separately compiled and linked to form an executable, and *main files*, which supply commonly used declarations. Each time a source file is compiled, a new compiler process is created. During compilation, the process *internalizes* declarations in those main files on which the source file depends, building corresponding data structures, such as symbols and parse trees, within the process. Different language systems internalize declarations in a main file in different ways. For instance, in a C-based language system, the internalization usually consists of two steps: first, the preprocessor reads the original texts of a source file and of main files specified in it by include directives, and writes out a preprocessed source file; then, the compiler parses the resultant file. In a Modula-2 language system, on the other hand, the internalization involves only a compiler. Some Modula-2 compilers, such as the one described by [8] internalize declarations of a main file by compiling the original text, whereas others, such as the one described by [10] do so by reading a *precompiled* version of the main file, because precompiled main files can be more efficiently internalized.

Unfortunately, internal data structures built in one compiler process are never shared by another compiler process in most systems. As a result, a group of compiler processes repeatedly internalizes declarations in main files. Let us consider two typical situations that occur during the development of a program—*massive* compilation and *repetitive* compilation. A massive compilation, in which many source files are compiled in series, is caused when an attempt is made to build an executable after making modifications that influence many of the source files; it sometimes occurs even as a result of a single modification to a main file. Since each main file is ordinarily used in more than one source file, redundant internalization occurs [14].

2. MOTIVATION

2.1 Execution Model of Java Virtual Machines

Running Java programs normally consists of two steps: converting Java programs into bytecode instructions (i.e., compiling Java source to *.class* files), and executing the resulting class files [9]. Because the compiled class files are network- and platform-neutral, one can easily ship them across a network to any number of diverse clients without having to recompile them.

JVMs then execute these class files, and to achieve reasonable performance, state-of-theart JVMs, such as HotSpot [16] and Jikes RVM [2], also perform dynamic native code generation and optimization of selected methods. Instead of using an interpreter, Jikes RVM [1] includes a *baseline* (non-optimizing) and an *optimizing* compiler. The baseline compiler is designed to be fast, and easy to implement correctly, while the optimizing compiler is designed to produce more efficient machine code by performing both traditional compiler optimizations (such as common subexpression elimination) and modern optimizations designed for object-oriented programs (such as pre-existence-based inlining [6]. In addition to providing flags to control individual optimizations, Jikes RVM provides three optimization levels: O0, O1, and O2. The lower levels (O0 and O1) perform optimizations that are fast (usually linear time) and offer high payoff. For example, O0 performs inlining, which is considered to be one of the most important optimizations for object-oriented programs. O2 contains more expensive optimizations such as ones based on static single assignment (SSA) form [4].

2.2 Compilation Pause Times

In addition to affecting overall running time of applications, dynamic compilation also affects their responsiveness because it induces pauses in program execution. The use of pause times as a performance metric is popular when evaluating garbage collection algorithms [3], and it should be equally important in evaluating dynamic compilation systems. For example, H¨olzle and Ungar [11] uses the concept of absolute pause times to evaluate the responsiveness of the SELF programming system.

2.3 Memory Usage

Performing code optimizations consumes memory and thus may degrade memory system performance. Since Java programs will be optimized at run time, there are two memory costs for optimizations: (i) the data space cost, i.e., the space required by the optimizer to run; and (ii) the instruction space cost, i.e., the footprint of the optimizer. (The final size of optimized code may be larger or smaller than unoptimized code, but this size effect is much smaller than the other two)[12].

3. REASON FOR SELECTION

A common method for installing a compiler is to have one copy locally held on each client workstation. The compiler and associated utilities and library files could be maintained by system. In common installation methods system administrator need to install compiler on every server and also if any updates are there in library function provided by the specified language or the compiler. This installation can be difficult task in case of multiple clients. As client performance is always slower that the server in the network we can then also implement server client technology for compilation purpose.

So the main reason and the goal for selection of the project is to centralization of the compiler that is single copy on the server. This system will let the client use compilation power and the speed of server directly form client machine. It also let the compiler developer and the system administrator manage compiler code and the libraries provided by the language. Now administrator doesn't need to install compiler software on each client.

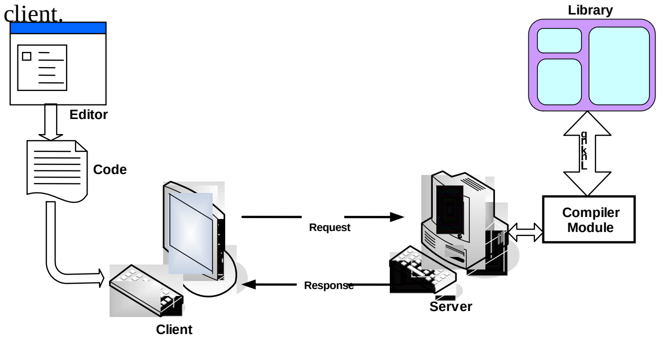

Fig.1. Basic block diagram

4. PROPOSED PLAN OF ACTION

➢ Design and implementation of compilation server:

The primary design goal of *Compilation Server* is to minimize client execution time. *Compilation Server* clients may include desktop PCs, laptops, and PDAs, and thus are likely to be limited in one form or another compared to *CS*, which would be equipped with plenty of memory, fast disk drives, and fast network connection(s). Therefore, it would be beneficial to allow the server to various required tasks.

➢ Steps to develop compilation server:
   a) Develop a small language
   b) Design compiler for that language
   c) Create editor for same language
   d) Then install compiler on server as shown in fig. 2

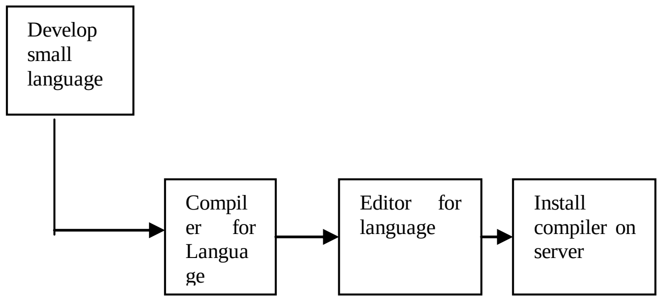

Fig.2. Preprocessing for Compilation Server

➢ Architecture of compilation server is shown in fig.3
   a) Connect editor to client side
   b) Connect editor to server also
   c) Client edit the code and send request to server for compilation
   d) Server compiles the file send back to client
   e) Client can execute the file and show the output

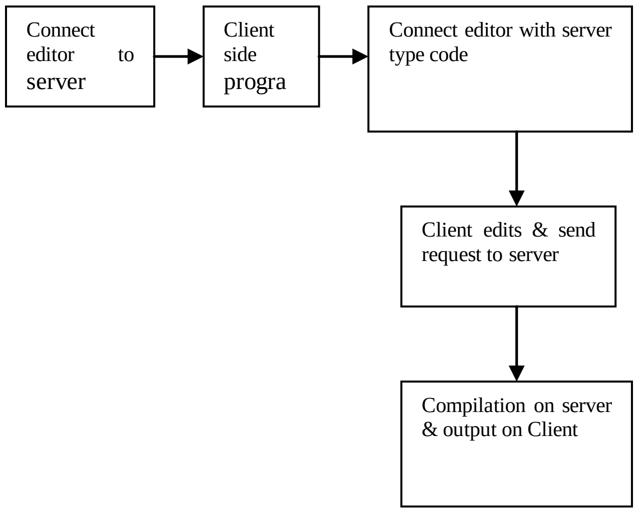
Connect editor to server
Client side progra
Connect editor with server type code
Client edits & send request to server
Compilation on server & output on Client

Fig 3. Architecture of Compilation Server

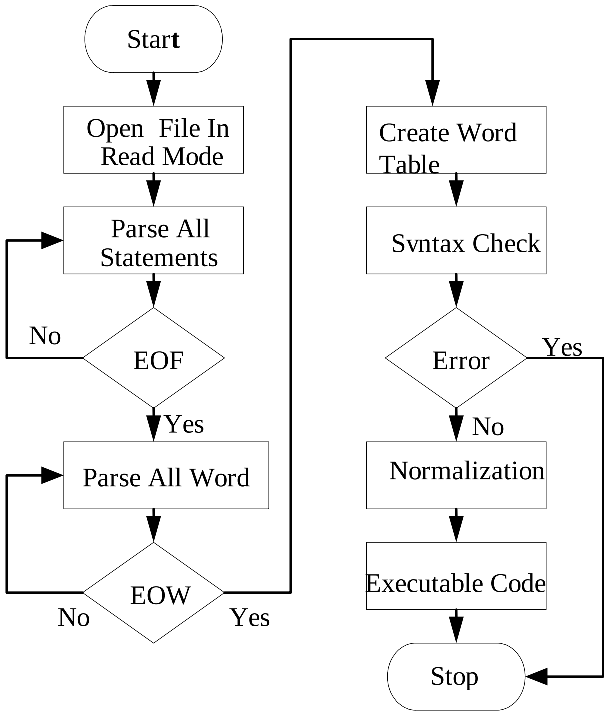

Fig 4. Behavior of Compilation Server

As shown in fig.4 client reads the file and parses all statements and words. Then it creates the word table. After that syntax will be checked. Then it comes the part of server. Server does the important part of compilation. It performs normalization and converts the file into executable one so that client can execute it.

5. ADVANTAGES

Compilation server provides the following benefits :

a) Lower execution time and pause time due to reducing the overhead of optimization

b) Lower memory consumption of the client by eliminating allocations due to optimizing compilation and footprint of the optimizing compiler.

c) It can manage concurrent clients.

d) Centralization of library function

6. RESEARCH METHODOLOGY

➢ Development Tool

VB.Net is object oriented language, and its major features are (1) class interfaces without private members, (2) run-time type descriptors, (3) garbage collection and (4) on-demand internalization of class interfaces. When we collect statistics on the main files, we will focus on those containing class interfaces; we do not count system main files. The compilation server is a long-lived compiler process that accepts and handles successive compilation requests from a client. The most important feature is that it can retain internal data structures generated while serving a request and use them to deal with subsequent requests; it requires main files to be internalized only once at most. We can thus expect it to reduce the compilation time in both massive and repetitive compilations. This project describes details with creating and using a compilation server, in which object orientation is used as both the source language and the implementation language.

➢ Operating System

Microsoft Windows 2000 or any grater versions

7. RELATED WORK

The idea of a compilation service means reducing the energy consumption of mobile devices using power models in [15]. This work is an extension of that prior work, but instead of using power models to investigate energy consumption, it presents design and implementation of compilation server and clients.

7.1 Server-based Compilation

There is related work in the area of server-based compilation for Java as well as for static programming languages such as C and C++.

_Delsart et al. [5] describe a framework called JCOD designed to perform native compilation of Java classes similar to ours but with completely different design goals. Their design is tailored for embedded devices with very limited memory, and thus focuses on improving client performance with minimal increase in code size and memory requirements. In fact, their compile-server performs only a few optimizations that reduce code size: they perform no method inlining or loop unrolling since those optimizations may increase code size. They are also concerned with producing code that is independent of the operating system and virtual machine, and to that end they implemented a generic object format that must be linked on the client, which results in high overhead.

The design philosophy of this work is that any task that can be performed on the server should be done there since servers can be expected to be much more capable machines.

Newsome andWatson [13] describe a proxy compilation scheme called MoJo in which a server compiles Java class files to C source code and then to an object file to be used by a client using GNU gcc. MoJo handles only a subset of Java and does not allow recompilation of "hot" methods but rather compiles whole class files at once. In this sense, MoJo acts more like a way-ahead-of-time compiler that batch compiles for its clients. Client execution is halted until compiled code is received.

This work differs, optimization of "hot" methods are only considerd, interleaving execution with optimization request. There has been some effort in distributing compilation of static programming languages such as C. The problem is that these approaches are trying to reduce overall compilation wait-time and is much simpler to solve since everything can be compiled.

7.2 Task Migration

The idea of offloading compilation onto a dedicated server can be considered as a specific instance of task migration.

Flinn et al. [7] describe a framework that automatically downloads tasks to a wired server based on information provided by the application and past profiles. Their work also incorporates a notion of "fidelity". For example, their system may decide based on the environment to use either the full or a short vocabulary for a speech recognition system.

Teodorescu and Pandey [17] describe a Java system that is distributed across servers and resource-limited devices. The resource-limited devices run minimal kernels that download parts of the run-time system on demand. The granularity of transferring code is a method. However, all compilation is done on the server.

8. CONCLUSION

So a compilation server is effective in reducing the compilation time both in massive compilation and in repetitive compilation. It is not simply a tool for reducing the compilation time, but can function as a central tool in a programming environment. This is because the symbols and parse trees kept in the server process represent almost all the aspects of the source files compiled.

A *compilation server* compiles client code at the granularity of methods to reduce or eliminate the cost associated with dynamic compilation. Compilation server is able to handle more than 50 concurrent clients while still allowing them to outperform best performing adaptive configuration. CS is also effective at reducing clients' end-to-end execution times, pause times, and memory consumption. Being able to migrate compilation onto a remote server using CS approach will have significant impact on the way virtual machines and optimizations are designed and implemented.

9. REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney, 2000. Adaptive optimization in the Jalape˜no JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*.ACM Press, Minneapolis, MN, 47–65.

[2] M. Burke, J. D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, and H. Srinivasan, 1999. The Jalape˜no dynamic optimizing compiler for Java. In *ACM Java Grande Conference*. ACM Press, San Francisco, CA, 129–141.

[3] P. Cheng, and G. E. Blelloch, 2001. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM Press, Snowbird, UT, 125–136.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Adeck, 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13,* 4, 451–490.

[5] B. Delsart, V. Joloboff, And E. Paire, 2002. JCOD: A lightweight modular compilation technology for embedded Java. In *Proceedings of the Second International Conference on Embedded Software*. Springer-Verlag, Grenoble, France, 197–212.

[6] D. Detlefs, and O. Agesen, 1999. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*. Springer-Verlag, Lisbon, Portugal, 258–278.

[7] J. Flinn, D. Narayanan, And M. Satyanarayanan, 2001. Self-tuned remote execution for pervasive computing. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII*. IEEE Computer Society Press, Schloss Elmau, Oberbayern, Germany.

[8] D. G. Foster, 'Separate compilation in a Modula-2 compiler', *Software–Practice and Experience* 16,101–106 (1986).

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, 2000. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[10] J. Gutknecht, ‘Separate compilation in Modula-2: an approach to efficient symbol files’, *IEEE Software*,3, (11), 29–38 (1986).

[11] U. H¨olzle, and D. Ungar, 1994. A third-generation SELF implementation: reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*. ACM Press, Portland, OR, 229–243.

[12] H. B. Lee, D. Von Dincklage, A. Diwan, and J. E. B. Moss, 2007. Design, implementation, and evaluation of a compilation server. ACM Trans. Program. Lang. Syst. 29, 4, Article 18 (August 2007).

[13] M. Newsome, And D. Watson, 2002. Proxy compilation of dynamically loaded Java classes with MoJo. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, Berlin, Germany, 204–212.

[14] T. Onodera, Reducing Compilation Time by a Compilation Server IBM Research, Tokyo Research Laboratory, 5–19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan

[15] J. Palm, H. Lee, A. Diwan, And J. E. B. Moss, 2002. When to use a compilation service? In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, Berlin, Germany, 194–203.

[16] M. Paleczny, C. Vick, and C. Click, 2001. The java hotspot(tm) server compiler. In J*ava Virtual Machine Research and Technology Symposium*. The Usenix Association, Monterey, CA.

[17] R. Teodorescu, And R. Pandey, 2001. Using JIT compilation and configurable runtime systems for efficient deployment of Java programs on ubiquitous devices. In *Ubiquitous Computing 2001, LNCS 2201*. Springer Verlag, Atlanta, GA, 76–95.