A Brief History of Just-In-Time

JOHN AYCOCK

University of Calgary

Software systems have been using "just-in-time" compilation (JIT) techniques since the 1960s. Broadly, JIT compilation includes any translation performed dynamically, after a program has started execution. We examine the motivation behind JIT compilation and constraints imposed on JIT compilation systems, and present a classification scheme for such systems. This classification emerges as we survey forty years of JIT work, from 1960–2000.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors; K.2 [**History of Computing**]: *Software*

General Terms: Languages, Performance

Additional Key Words and Phrases: Just-in-time compilation, dynamic compilation

1. INTRODUCTION

> Those who cannot remember the past are condemned to repeat it.
>
> *George Santayana, 1863–1952* [Bartlett 1992]

This oft-quoted line is all too applicable in computer science. Ideas are generated, explored, set aside—only to be reinvented years later. Such is the case with what is now called "just-in-time" (JIT) or dynamic compilation, which refers to translation that occurs after a program begins execution.

Strictly speaking, JIT compilation systems ("JIT systems" for short) are completely unnecessary. They are only a means to improve the time and space efficiency of programs. After all, the central problem JIT systems address is a solved one: translating programming languages into a form that is executable on a target platform.

What is translated? The scope and nature of programming languages that require translation into executable form covers a wide spectrum. Traditional programming languages like Ada, C, and Java are included, as well as little languages [Bentley 1988] such as regular expressions.

Traditionally, there are two approaches to translation: compilation and interpretation. Compilation translates one language into another—C to assembly language, for example—with the implication that the translated form will be more amenable to later execution, possibly after further compilation stages. Interpretation eliminates these intermediate steps, performing the same analyses as compilation, but performing execution immediately.

This work was supported in part by a grant from the National Science and Engineering Research Council of Canada.

Author's address: Department of Computer Science, University of Calgary, 2500 University Dr. N. W., Calgary, Alta., Canada T2N 1N4; email: aycock@cpsc.ucalgary.ca.