

Program Reincarnation: A Pragmatic Approach to Solving to the Multicore Software Problem

Saman Amarasinghe

Abstract

As multicore processors become the dominant hardware platform, the burden of taking advantage of the available processing power of the multicores will fall squarely on the software community. However, the software community carries with it the immense burden of maintaining the flexibility and performance of decades' worth of legacy code. Legacy programs often have little in common with today's development practices; they were written in different language dialects and targeted a different class of computer architectures. Thus, migrating these programs to harness the power of multicore architectures will become a critical need. With current technology and practices, it is largely intractable to manually upgrade a legacy application to modern performance standards.

We propose a pragmatic approach to parallelizing legacy applications. We believe that no single technology will be able to do the parallelization in a fully automatic or transparent manner. However, we can bring together a host of technologies that can drastically improve the ability of a programmer to effectively and efficiently transform a legacy application into a modern parallel program. Dubbed "Program Reincarnation", the proposed tool will assist the programmer in replacing the program's code ("the body") while preserving the original specification ("the soul"). The proposed technique employs a combination of static and dynamic program analysis to extract the simple, high-level block diagram from the optimized and obfuscated legacy code. This comprehensive approach is broadly applicable to program understanding, documentation, refactoring, and automatic parallelization.

Introduction

Over the last few decades, programmers have written a staggering amount of code. These billions of lines of code have profoundly impacted the human race. Today, programs and code are everywhere – in computers, cell phone, cars, cameras and cash registers. While computer technology is going through an incredibly rapid growth period, most computer code seems to enjoy a longevity mainly associated with a mature field. It is not uncommon to actively use computer programs written two decades ago. Furthermore, even the latest software seems to include a large amount of program code written a long

time ago. For example, the Microsoft Windows vulnerability MS-03-011 affected Windows 95 to Windows 2003, suggesting that the code written before 1995 was still in use in a program produced 8 years later. In contrast, it is hard to even find a working computer older than five years, let alone a modern computer built with parts designed two years ago.

This reliance on old code, written using old languages and outdated methods mainly targeting now-defunct machines, is creating a massive obstacle to the rapid growth of computers. Most of these legacy programs cannot take advantage of the exponential growth of computational power in modern processors. Some of the performance-critical legacy programs, ones that were highly optimized for the architecture of the day, will even experience slow-downs or compatibility issues in newer processors. Legacy code has had an even larger negative impact in computer architecture. As legacy applications are extremely important, commercial microprocessors have been “wasting” the transistor budgets offered by Moore’s Law to improve the performance of legacy programs at any cost, even if the performance gains are marginal. Monolithic superscalar processors are a good example of this trend.

Recently, processor vendors have finally started to break this trend by moving to multicores. Multicores provide much better peak performance per die area, but require programs to be explicitly parallel. This trend, while providing much higher performance to modern programs, puts legacy applications at a disadvantage as they will no longer get a performance boost from modern processors.

In order to take advantage of modern processors, it is necessary to rewrite these legacy binaries using modern languages and techniques. Apart from performance, there are many other benefits of updating the legacy code base. Most of these programs are written in older languages without the benefit of many powerful language features. Furthermore, during the last decade we have made huge strides in software engineering. Rewriting the legacy code base using these techniques can make them more efficient, malleable, portable, secure, and fault tolerant.

However, rewriting a legacy binary is a daunting task. While the application may be in wide use, it is very difficult and time consuming to understand the exact algorithm implemented and special cases handled by the application. The original programmers are unavailable in many cases. Information that helped the original programmer such as specification and requirement documents, simulations, mathematical proofs, tests etc. are also unavailable for most of these applications. In some cases, the source code may no longer be available. Even if the source is available, the tool chain and the libraries have often diverged over the years, making it impossible to build the application from source. Today, recreating the specification of a legacy application is more error prone and takes longer than the coding task itself.

Program Reincarnation

We believe that it is possible to drastically reduce the cost of recreating a program. In Program Reincarnation we will provide a tool chain to help the programmer replace the program code ('the body') while adhering to the original program's specification ('the soul'). This tool chain will take advantage of the availability of the source code and a working program to help guide the programmer through the reincarnation process.

The overall design flow in a comprehensive system for program reincarnation is given in Figure 1. The design flow will be controlled by the Assisted Application Reincarnation Tool. First, the execution profile of the instrumented binary as well as static analysis of the source code is fed to an inference engine. This engine will build an annotated representation of the program. This representation will be used for many tasks. The block diagram and the derived specifications will be presented to the programmer. In addition, the programmer will be provided with hints on refactoring of the program. When possible, the system will attempt to automatically parallelize the application. Appropriate tests will be generated to help discover the program invariants as well as check the compliance of the reincarnated application.

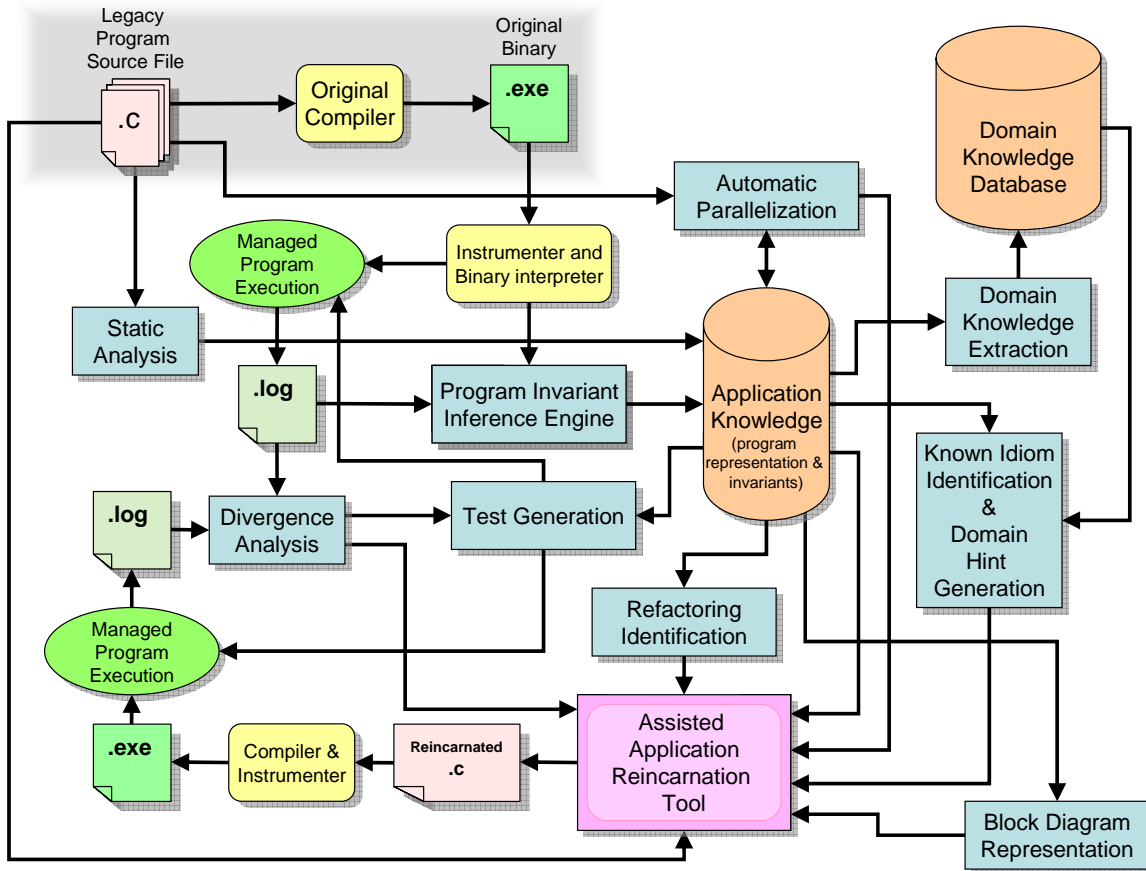


Figure 1 Design flow for Program Reincarnation.