System Optimizer™ Technology Overview

Abstract

The speed of computer hardware and the costs of software development are both increasing exponentially. These two opposing forces are taken for granted in the computer industry. The fact that software development is unable to keep pace with advances in hardware technology is a major industry-limiting factor. We explore some aspects of this dichotomy and propose a new technological method of reducing these negative effects while obtaining significant increases in computer performance in the real world.

A proposed business strategy shows marketing approaches to build a profitable company based on this technology. Finally, a series of steps illustrate an initial development path as well as a plan for continued growth into the future.

Introduction

The capabilities of modern computer hardware are increasing at an exponential rate. The efficiency of the software programs that run on this new hardware is not improving at all. In fact, there is a strong tendency in the software community to ignore software efficiency issues in favor of simply "throwing hardware" at a problem. This mind set can be easily justified by simple economics: over time, the cost of hardware performance goes down while the cost of software development increases.

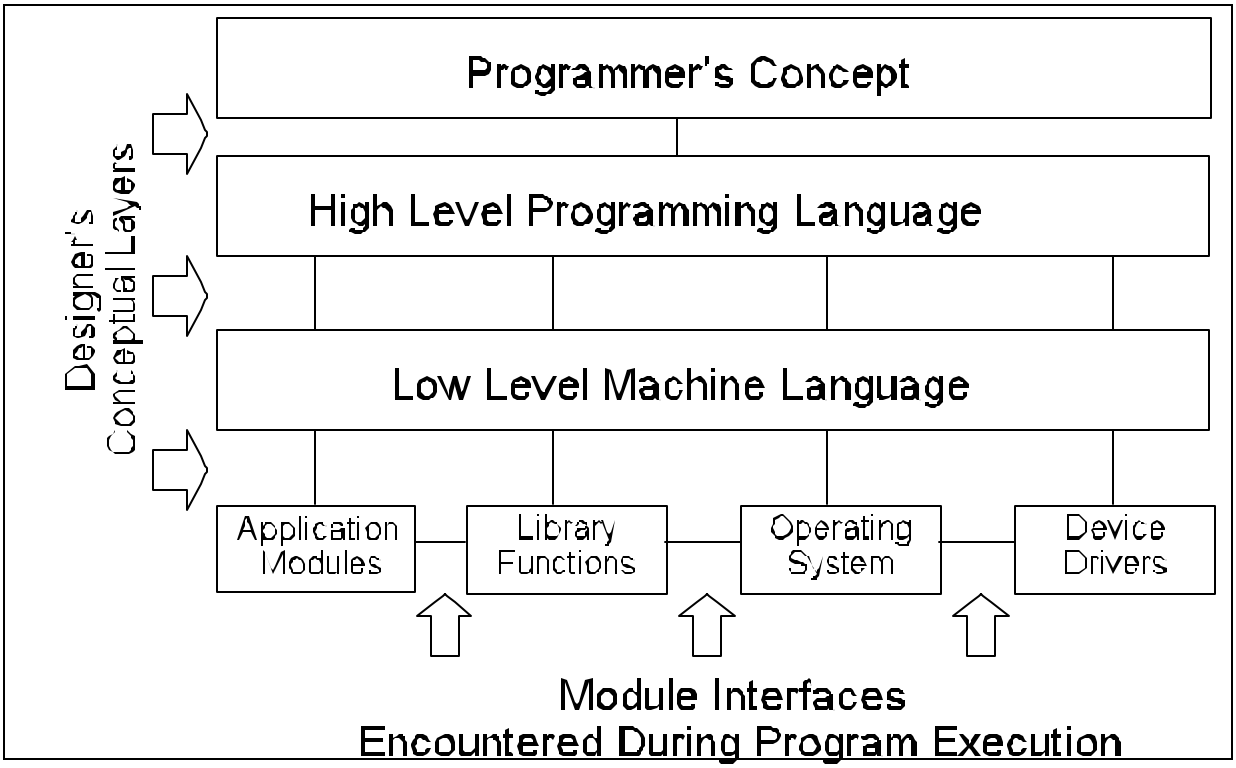
The cost of software development and the amount of time required to develop and test an application leads to many undesirable characteristics that have become an accepted part of modern software. Because there are no clear alternatives to the use of these development techniques, the attendant disadvantages remain largely unappreciated.

Advantages and Disadvantages of Selected Software Development Techniques

Technique	Advantages	Disadvantages
Standard Third-Party Library Modules	• Documented Interface • Known Performance • Known to be Correct	• Unknown efficiency • General-Purpose Design includes Unnecessary Features
Application Modules	• Parallel Development • Parallel Testing • Improve Project Management	• Inefficient Execution • Legacy Technology
Code Re-use	• Reduced Development Time • Increased Reliability	• Inefficient Execution • Legacy Technology
Target "Least-Common-Denominator" Hardware	• Broad Usability	• Inefficient Execution in many cases
Continuous Version Upgrades	• Customers Get New Features • Build on Existing Framework	• Core Functions Become Progressively Outdated.

Modularization Considered Harmful

A key technique used by software developers is modularization. Breaking a program into conceptual modules enables a parallel development effort to be conducted by different teams of programmers. Standardized interfaces between these independently developed modules allow these independent groups of people to ensure correct operation of the overall project. The availability of libraries of standard software modules reduces project complexity and development. The reuse of proven modules also helps minimize the introduction of undetected errors.



Unfortunately, this same modularity is the fundamental cause of the progressive loss of program efficiency. During program execution, each time a new "standard interface" is encountered the processor must spend time transferring control to the other module as well as transforming data to match the interface requirements and then back again. These transformations and control changes do not constitute useful steps toward the goal of the application. They are artifacts introduced by the design process to help ensure that the human beings writing the program are able to do it right.

> The key concept here is that the hardware executing the program has *no need whatsoever* for the modularity that made the software development possible.

Referring to the diagram above, it is not uncommon for many layers of modularity to exist in real-world applications. Each layer adds its own unnecessary transformations and inefficiencies. The translation of an application designer's intent into an executable program is handled by programming language compilers that implement the top-to-bottom links in the diagram. Traditional optimizers are tasked with making this

translation as efficient as possible for a particular processor. The flow of information horizontally across the module interfaces near the bottom of the diagram is the responsibility of the computer processor. Traditionally, the way to speed up this flow is to buy a faster processor. Each of these module interfaces represents a significant speed penalty due to the unnecessary processing required only for the sake of the interface itself.

Direct Interface	Modular Interface
	Transform Data to Match Interface
	Transfer Control to Function
Perform Useful Function	**Perform Useful Function**
	Transform Results to Match Interface
	Return Control to Main Program
	Move Results to Destination

In addition, the general-purpose nature of software modules (especially in third-party libraries) has led module developers to embed many "features" into the code that are often unnecessary and counterproductive for the specific application. In striving for generality many completely unrelated functions are often combined into a single module. These modules are often bloated, inefficient implementations of otherwise simple tasks. The modular design philosophy thus leads to single modules with multiple, unrelated goals.

Conversely, the "sealed black box" aspects of such modules frequently lead experienced developers to "roll their own" duplicate functions because they are afraid to rely on the unknown code contained in off-the-shelf libraries. This results in particular functions being redundantly implemented in many modules. In this case, the modular approach places the same function in multiple modules!

Legacy Software's Burden

In an ideal world new features would be seamlessly added to the existing base of mature software. As bugs are detected and fixed, older software becomes more robust and reliable with time, forming a strong foundation for new development.

In the real world, however, old mistakes become permanently embedded right along with early, fundamental features. These flaws are entombed in layer upon layer of work-arounds and alternative algorithms for resolving historical problems that may never exist in the future. Modern programs still contain instructions for dealing with the Pentium FDIV bug, despite the fact that such flawed processors have not been manufactured for more than six years. Enhanced instruction set features such as Intel's MMX can provide significant performance improvements. Specific applications such as video games make use of the extensions, but mainstream applications are slow to move away from the minimal instruction set of the x86 architecture.

The addition of support for new hardware features and the removal of work-arounds for old problems are

traditionally expensive and time consuming. Software developers see very little reason to devote their scarce resources toward efforts that yield no apparent benefit.

Years of applying this "If it ain't broke, don't fix it" mentality has become the recipe for the mediocrity of the current software.

Traditional Approaches to Optimization

Program optimizers have existed in various forms since the days of the first compilers. Great strides have been made in matching high level languages with the instruction sets of various computers. All modern compilers have options that, for example, allow a tradeoff between speed and memory usage. Even simple optimizations achieve significant performance improvements over "brute force" code generation, and some academic compilers and linkers automatically achieve extremely efficient execution within a *single application* program.

Performance analysis tools from a number of companies are commonly used to find bottlenecks or inefficient sections in programs as they are being developed and tested. The developers then examine the offending section and *manually* revise the software. Although these techniques are an important first step, an interesting effect of "hot spot elimination" techniques is to produce the *illusion* of overall efficiency while leaving fundamental inefficiencies spread evenly throughout the computer system.

All traditional approaches to optimization are geared toward enabling the program developer to improve his own product. Even though his product may rely heavily on interfaces to modules provided by other vendors, these remain outside the control of the developer. In contrast, the proposed System Optimizer™ technology allows improvements to be made in the linkages between program modules regardless of their original source. This includes improving the performance of the interface between the program, the operating system, and hardware-specific drivers.

An additional benefit for the user will be increased system reliability. Many crashes are due to the accidental presence of an incompatible module in an otherwise working system. This leads to the common situation where the user "re-installs the program" to make it work again. The System Optimizer™ analysis of the linkages between modules will detect these previously hidden errors.

The System Optimizer™

All of the development techniques described above are a necessary part of modern software development. What is needed is a method of minimizing the negative effects of using these techniques into the future. The proposed System Optimizer™ technology provides a method of curing the ills of the development process after the fact. Traditionally developed, existing programs are fed into the System Optimizer™ and improved programs are automatically generated.

System Optimizer™ technology analyzes and modifies the actual executable program files on a user's system. This technology will be capable of performing fully automated improvements to a wide variety of program files. Instead of concentrating on a single file, the executables of an entire system are treated as part of a combined whole. Looking at the big picture allows programs, libraries, operating system files and device drivers to be unified into an efficient body of code that could not have been created by any one of the developers working by himself.

Software developers must design for as general a market of potential users as possible. Because the System Optimizer™ applies changes to a specific hardware/software gestalt environment it can make specialized improvements that would be impossible for the developers to consider.

Most programs do not use the MMX extensions available on modern Intel processors. Since the System Optimizer™ applies improvements to the complete inventory of application software installed on a particular system, it is able to ensure that the capabilities of the particular hardware are used to their greatest advantage.

The initial goal of the System Optimizer™ technology is to provide an improvement in real-world performance equivalent to doubling the speed of the processor. This is approximately equivalent to leaping forward 18 months in hardware technology.

Business Strategy

System Optimizer™ technology presents six distinct but complementary marketing opportunities.

- **Consumer Products**. The Optimizer software can be marketed directly to consumers as a shrink-wrapped retail product or for download via the web. It would be used in a manner similar to a disk de-fragmenter to improve the performance of the computer.
- **Developer Products**. The Optimizer software would be sold (licensed) to software developers and used as a tool by them to improve the performance of their products.
- **OEM Services**. Major computer manufacturers (i.e. HP, Dell, Gateway, and IBM) bundle software from many manufacturers with their systems. Our strategy in this case would be to provide an Optimization Service to these manufacturers. The result would be an optimized suite of applications for delivery with the system instead of the traditional software bundle. As computers move toward turnkey devices, optimized systems could help promote brand loyalty and provide increased capabilities using less expensive hardware.
- **Translation Services**. The detailed analysis performed as part of the Optimizer process can be used to generate code for use on a different computer processor. This is conceptually similar to "porting" an application from the PC to the Macintosh. This would not be a fully automated process in the beginning, so it would be offered as a service to large customers with a large existing base of legacy code. The recommended customers would be manufacturers of telephone switches: Nortel, Motorola and Ericsson.
- **Technology Licenses**. Optimizer software can be licensed for incorporation directly into the operating system's program loader. The obvious candidate here is Microsoft, but other developers of real-time and embedded operating systems may see even greater advantages.
- **Embedded Systems**. Optimizer software can be targeted for any device which embodies a single-use firmware load. Primary examples would be appliances such as cell phones, GPS receivers, and video games. The Optimizer offers of significant performance improvements without new hardware.

As with all software products there are ample opportunities to introduce different versions covering a spectrum of capabilities and price points. Future hardware and software development in the industry

provide the basis for ongoing revision upgrades which represent continuing revenue streams.

From a business standpoint it is wise to target the Windows software environment and Intel Pentium® processors for the initial products. The core technology will be applicable to other platforms including Linux, Windows CE, Sun, MIPS and Alpha.

Steps Toward the System Optimizer

In order to develop the System Optimizer™ technology and prepare it for market it will be necessary to take a series of steps.

1. Write a static analyzer for executables that can be tailored to detect optimization candidates.
2. Analyze a variety of real-world computer systems to establish a targeted strategy.
3. Write a reliable Re-linker that can modify executables without introducing errors.
4. Select a target operating system and suite of software for use in bench marking the development and demonstrating the performance of the technology.
5. Write a control-flow and data-flow analyzer to further refine optimization candidates. Critical here is the ability to detect potentially unsafe optimizations.
6. Test techniques for introducing specific optimizations. Develop methods to use traditional peephole or RTL optimizations across module boundaries.
7. Use advanced hardware monitoring to select new optimization candidates.

In researching the possibilities of System Optimizer™ technology several areas for improvement have already been discovered.

1. There are many retail applications that are delivered with virtually no optimizations whatsoever. In particular, companies often leave debugging and diagnostic options turned on in their products to provide information for customer support. If the application isn't crashing, eliminating this extraneous code will present immediate benefits to the consumer.
2. Many statically linked application and library functions which make no calls may be replaced by in-line code.
3. Function calls with constant parameters are good candidates for splitting into multiple, specialized functions with fewer parameters.
4. Functions that begin with a conditional should be moved at least partially in-line.
5. Various forms of parameter passing and local variable allocation should be eliminated.
6. Advanced instruction sets should be part of the optimizer's output generation capabilities.
7. Many production programs are released with obvious optimizations such as instruction reordering disabled in an attempt to support source-line and source-module correlations with particular instructions.

An Overview of the Project

No assembly language programmer who looks at a trace of the execution of compiled code says "Gee, that's good code." They invariably say "I could do that better, but I don't have time." Even then, these observers are still looking at the small picture. They do not have the tools to help them identify overall meaningful instructions as different from useless instructions.

Assuming a correctly functioning program, let us envision the sequence of instructions actually executed by a processor. In an ideal world we should be able to identify two kinds of machine instructions. "Meaningful" instructions are ones that are necessary to move toward the goal of the application. "Useless" instructions are ones which do not contribute toward the goal. It is easy to identify with certainty some useless instructions: the NO-OP, and the unconditional branch, for example, are useless by definition and never contribute toward the goal. (Practical considerations necessitate the existence of instructions such as unconditional branches but their frequency of occurrence can be reduced to arbitrarily low levels). Most instructions may be meaningful in some contexts and useless in others. Good examples are redundant LOADs or STOREs: in one instance they are useless, in the next they are meaningful.

Possibly the most surprising result arising from this concept is that all software operations that manipulate the stack are useless. PUSH and POP operations are used exclusively for parameter passing, and CALL, RETURN and Software Interrupt instructions are simply elaborate forms of unconditional branches. These are all artifacts left over from the modular construction techniques required by the human design process. They do not actually contribute to the solution of the problem at hand. (Note that this does not refer to stack operations relating to hardware interrupts or exception management).

The major conceptual difficulty is that it is not possible to identify guaranteed-meaningful instructions from the low level looking up. Only a view of the big picture offers any assurance that the results of a particular computation are not subsequently discarded, thus rendering all those computing steps useless.

The goal of our optimizer will be to reduce the percentage of useless instructions actually executed by a processor in the real world. Practical considerations dictate that some useless instructions will remain, but we should be able to reduce their impact to any desired level.

After eliminating useless instructions the next step is to identify groups of instructions that can be transformed into a faster group that accomplishes the same goal. Classic examples include the optimization of register usage. This step uses the same methods proven in traditional optimizing compilers, but it is applied across module boundaries. It is here that new instruction set features can be generated as needed.

Finally, the resulting instruction sequences should be written to an updated version of the executable file. In line with the classic trade-off between speed and memory, this will invariably result in a somewhat larger but much faster program.

Conclusion

Modern computer hardware is rapidly increasing in speed but software development is lagging further and further behind. Many of the methods used to improve the productivity of software development efforts actually result in more inefficient programs. This progressive loss of efficiency is masked by the tremendous increases in hardware performance.

We presented a proposal for a System Optimizer™ which can make a significant improvement in computer performance using existing hardware and software as a basis. The technology would build on and improve, not replace, current software. Several marketing and business strategies were outlined and multiple opportunities for additional development were mentioned.

The System Optimizer™ represents a unique opportunity to improve the overall performance of existing computer systems in one broad stroke. Using this technology, users can see immediate benefits without installing new hardware or engaging in piecemeal upgrades of software packages.