Spike: An Optimizer for Alpha/NT Executables

Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin
Digital Equipment Corporation

Spike: An Optimizer for Alpha/NT Executables

Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin

spike@vssad.hlo.dec.com

Digital Equipment Corporation

Abstract

Spike is a profile-directed optimizer for Alpha/NT executables that is actively being used to optimize shipping products. Spike consists of the Spike Optimization Environment (SOE) and the Spike Optimizer. Through both a graphical interface and a command-line interface, the Spike Optimization Environment provides a simple means to instrument and optimize large applications consisting of many images. SOE manages the instrumented and optimized images as well as any profile information collected for those images, freeing the user from many tedious and error-prone tasks typically associated with profile-directed optimization. SOE also simplifies the collection of profile information with Transparent Application Substitution (TAS). With TAS, the user invokes the original version of the application and the instrumented or optimized version of the application is transparently executed in its. SOE uses the Spike Optimizer to optimize images. The Spike Optimizer performs code layout to improve instruction cache behavior [Pettis90], hot cold optimization [Cohn96] and register allocation. The optimizations are targeted at large call-intensive applications, where loops span multiple routines, and each routine contains complex control-flow. For this class of applications, Spike provides significant performance improvement, reducing execution time by as much as 20%.

1. Introduction

Profile-directed optimization is rarely used in practice because of the difficulty of collecting, managing, and applying profile information. Spike solves most of these problems, allowing the user to easily optimize large applications composed of many images. In this section we describe the general procedure for using profile-directed optimization, its difficulties, and how Spike overcomes these difficulties.

The first step in profile-directed optimization is to *instrument* each image in an application so that when the application is run, profile information is collected. Instrumentation is most commonly done by using a compiler to insert counters into a program during compilation [Multiflow], or by using a post-link tool to insert counters into an image [Atom, Pixie]. Statistical or sampling-based profiling is an alternative to counter based techniques [DCPI, MORPH]. Some compiler-based and post-link systems require that the program be compiled specially, so that the resulting images are only useful for generating profiles. Many large applications have lengthy and complex build procedures. For these applications, requiring a special rebuild of the application to collect profiles is an obstacle to the use of profile-directed optimization.

Spike directly instruments the final production images so that a special compilation is not required. Spike does require that the images be linked to include relocation information. However, including this extra information does not increase the number of instructions in the image and does not prevent the compiler from performing full optimizations when generating the image.

Most large applications consist of multiple images, a single executable and many dynamically linked libraries (DLL). Instrumenting all the images can be difficult, especially since the user doing the profile-directed optimization may not know all of the images in the application. Spike relieves the user of this task by finding all the DLLs that are used by the application, even if they are loaded dynamically (i.e. with a call to LoadLibrary).

After instrumentation, the next step in profile-directed optimization is to execute the instrumented application and collect profile information. Most profile-directed optimization systems require that the user first explicitly create instrumented copies of each image in an application. Then the user must assemble the instrumented images into a new version of the application, and execute it to collect profile information. As the profile information is generated, the user is responsible for locating all the profile information generated for each image, and merging that information into a single set of profiles. Our experience with users has shown that requiring the user to manage the instrumented copies of the images and the profile information is a frequent source of problems. For example, the user may fail to instrument each image, or may attempt to instrument an image that has already been instrumented. The user may be unable to locate all the generated profile information, or may incorrectly combine the profile information.

As described in Section 2, Spike frees the user from these tedious and error-prone tasks by managing the instrumented copy of each image as well as the profile information generated for each image.

After profile information is collected, the final step is to use the profile information to optimize each image. As with instrumentation, the typical profile-directed optimization system requires the user to explicitly optimize each image, and to assemble the optimized application. Spike uses the profile information collected for each image to optimize all the images in an application and assembles the optimized application for the user.

2. Spike Optimization Environment

The Spike Optimization Environment (SOE) provides a simple means to instrument and optimize large applications consisting of many images. SOE can be accessed through a graphical interface or a command-line interface that provides identical functionality. The graphical interface, called the Spike Manager, is described in Section 2.1. The command-line interface allows SOE to be used as part of a batch build system such as `make`.

In addition to providing a simple-to-use interface, SOE keeps the instrumented and optimized versions of each image and the profile information associated with each image in a database. When an application is instrumented or optimized, the original versions of the images in the application are not modified; instead SOE puts an instrumented or optimized version of each image into the database. SOE uses Transparent Application Substitution (TAS) to execute the instrumented and optimized version of an application when the user invokes the original version, as described in Section 2.2.

The Spike Optimization Environment allows the user to instrument and optimize an entire application using the following procedure:

1. *Register*: The user selects the application(s) that are to be instrumented and optimized. The user only needs to specify the application's main image. Spike then finds all the implicitly linked images (DLLs loaded when the main image is loaded) and registers that they are part of the application.
2. *Instrument*: The user requests that an application be instrumented. For each image in the application, SOE invokes the Spike Optimizer to instrument that image. SOE places the instrumented version of each image in the database. The original images are not modified.
3. *Collect profile information*: The user runs the original application in the normal way, e.g. from a command-prompt, from the Explorer, or indirectly through another program. Transparent Application Substitution invokes the instrumented version of the application in place of the original version. Any images dynamically loaded by the application are instrumented on the fly. Each time the application terminates, profile information for each image is written to the database and merged with any existing profile information.
4. *Optimize*: The user requests that an application be optimized. For each image in the application, SOE invokes the Spike Optimizer to optimize the image using the collected profile information and places the optimized version of each image in the database.
5. *Run optimized version*: The user runs the original application and TAS substitutes the optimized version, allowing the user to evaluate the effectiveness of the optimization.
6. *Export*: SOE exports the optimized images from the database, placing them in a directory specified by the user. The optimized images can then be packaged with other application components.

2.1. Spike Manager

The Spike Manager is the principal user interface for using the Spike Optimization Environment. The Spike Manager displays the contents of the database, showing the applications registered with Spike, the images contained in each application, and the profile information collected for each image. The Spike Manager enables the user to control many aspects of the instrumentation and optimization process, including specifying which images are to be instrumented and optimized, which version of the application is to be executed when the original application is invoked, etc.

2.2. Transparent Application Substitution

Transparent Application Substitution is a mechanism for transparently executing a modified version of an application, without replacing the original images on disk. SOE uses TAS to load an instrumented or optimized version when the user invokes the original application. With TAS, the user does not need to do anything special to execute the instrumented or optimized version of an application. The user simply invokes the original application in the usual way (e.g. from a command prompt, from the Explorer, or indirectly through another application) and the instrumented or optimized application is run in its place.

TAS performs application substitution in two parts. First, TAS makes the NT loader use a modified version of the main image and DLLs. Second, TAS must make it appear to the application that the original images were invoked.

TAS uses debugging capabilities provided by NT to specify that whenever the main image of an application is invoked, the modified version of that image should be executed instead. In each image, the table of imported DLLs is altered so that instead of loading the DLLs specified in the original image, each image loads their modified counterparts. Thus, when the user invokes an application, NT loads the modified versions of the images contained in the application. Some applications load DLLs with explicit calls to LoadLibrary. TAS intercepts those calls and instead loads the modified versions.

The second part of TAS makes the modified version of the application appear to be the original version of the application. Applications often use the name of the main image to find other files. For example, if an instrumented image requests its full pathname, TAS instead returns the full pathname of the corresponding original image. To do this, TAS replaces certain calls to kernel32.dll in the instrumented and optimized images with calls to hook routines. Each hook routine determines the outcome the call would have had for the original application, and returns that result.

3. Spike Optimizer

The Spike Optimizer is used to instrument and optimize images [Amitabh, Wilson96]. The optimizer is invoked by SOE and can also be invoked directly by the user. There are several phases when instrumenting or optimizing an image. During the first phase, the optimizer finds all of the code contained in an image. NT images mix code and read-only data in the same section, so the optimizer must analyze the flow paths from known code to find as much of the code as possible. If it cannot be determined if part of a section is code or data, it is handled conservatively to preserve correctness.

Next, the optimizer finds all references to addresses that must be updated when parts of the image are moved. These are commonly called relocatable addresses. For Alpha/NT images, these are PC relative branches, data in memory that refer to other data or code, and instructions which load literals that are addresses of code or data. For references to data, the optimizer must also identify the section to which the address refers, so that the address can be changed as the section is moved in memory. For a reference to code, the optimizer must identify the specific instruction that is referenced, because the optimizer can rearrange individual instructions.

After finding all the code, locating all the PC relative branches and the instructions to which they refer is straightforward. Addresses that are data or are literals in instructions can be found because they are pointed to by relocations. Relocations identify code and data that contain addresses that must be adjusted if the system must load an image in to memory at an address other than at its preferred address.

The Spike Optimizer uses a linear list of Alpha machine instructions, annotated with a small amount of additional information, as its intermediate representation (IR). On top of the IR, the optimizer builds a complete compiler-like representation for the image, including a call graph, flow graphs, routines and basic blocks. Images can be very large; for example the largest image in Unigraphics, a CAD application from EDS, contains 36 Mbytes of code in 60,000 routines. Thus, the optimizer's representations must be extremely space efficient.

The Spike Optimizer performs an interprocedural dataflow analysis to summarize register usage within the image [Goodwin97]. This enables optimizations to use and reallocate registers. The interprocedural dataflow is very fast, requiring less than 20 seconds on our largest applications. Memory dataflow is much more difficult because of the limited information available in an executable, so the optimizer only analyzes references to the stack.

Instrumentation or optimization insert, delete, or modify the IR. After instrumentation or optimization is complete, the new IR is output as Alpha instructions and references to relocatable addresses in data are updated to reflect the new layout of the image.

Program	Full Name	Type	Code layout workload	HCO workload
VC (c1)	Microsoft Visual C	compiler front-tend	Compiler test suite	N/A
VC (c2)	Microsoft Visual C	compiler backend	Compiler test suite	5000 lines of C code
SQLSERVR	Microsoft Sqlserver 6.5	database	TPC-C	cached TPC-B
ACAD	Autodesk Autocad	mechanical cad	SDUG benchmark	SDUG benchmark
EXCEL	Microsoft Excel 5.0	spreadsheet	BAPCO	BAPCO
USTATION	Bentley Systems Microstation	mechanical cad	rendering	rendering
WINWORD	Microsoft Word 6.0	word processing	BAPCO	BAPCO
TEXIM	Welcom Software Texim 2.0	Project management	N/A	BAPCO
MPEG	Digital Light & Sound Pack	Mpeg decoder	Mpeg file	N/A
MAXEDA	OrCad MaxEDA 6.0	electronic cad	N/A	BAPCO
PTC	ProEngineer	Mechanical cad	Misc.	N/A
EXCHANGE	Microsoft Exchange	Mail server	Misc.	N/A
VORTEX	SpecInt95	database	SPEC ref	SPEC ref
GO	SpecInt95	game	SPEC ref	SPEC ref
M88KSIM	SpecInt95	simulator	SPEC ref	SPEC ref
LI	SpecInt95	lisp interpreter	SPEC ref	SPEC ref
COMPRESS	SpecInt95	compression	SPEC ref	SPEC ref
IJPEG	SpecInt95	JPEG	SPEC ref	SPEC ref
GCC	SpecInt95	compiler	SPEC ref	N/A
PERL	SpecInt95	interpreter	SPEC ref	N/A

Table 1: Benchmark programs and their workloads

3.1. Instrumentation

The Spike Optimizer instruments an image by inserting counters into the image. Each counter records the number of times a particular piece of code executes. Using these counters, the optimizer can determine the number of times each basic block and control-flow edge in the image executes. Spike uses a spanning-tree technique proposed by Knuth [Knuth73] to reduce the number of counters required to fully instrument an image. For example, in an if-then-else clause, counting the number of times the if and then statements are executed is enough to determine the number of times the else statement is executed as well. Register usage information is used to find free registers for the instrumentation code, reducing the number of saves and restores necessary to free up registers. Instrumentation typically makes the code 30% larger. As part of the profile, Spike also captures the last target of a jump or procedure call that cannot be determined statically. We are adding the ability to collect block counts using statistical sampling with the DCPI continuous profiler [DCPI], which will eliminate the need to instrument an image and will greatly reduce the cost of profiling

Spike's profile information is persistent; small changes to an image do not invalidate the profile information collected for that image. Profile persistence is essential for applications that require a lengthy or cumbersome process to generate a profile, even when using low cost methods like statistical sampling. For example, generating a good profile of a transaction processing system requires extensive staging of the system. With persistence, the user can collect a profile once, and continue to use it for successive builds of a program as small changes are made to it. It is also possible to merge a profile generated by an older image with a profile generated by a newer image.

3.2. Optimizations

Spike performs three optimizations, code layout, hot cold optimization, and register allocation. Profile information is used to guide each optimization.

Code layout reduces the number of misses in the instruction cache and the total number of VM pages touched by the application. Spike uses the Pettis and Hansen algorithm [Pettis90, Hwu89], which has three phases. First, the basic blocks in each routine are rearranged so that frequently executed paths are straight-

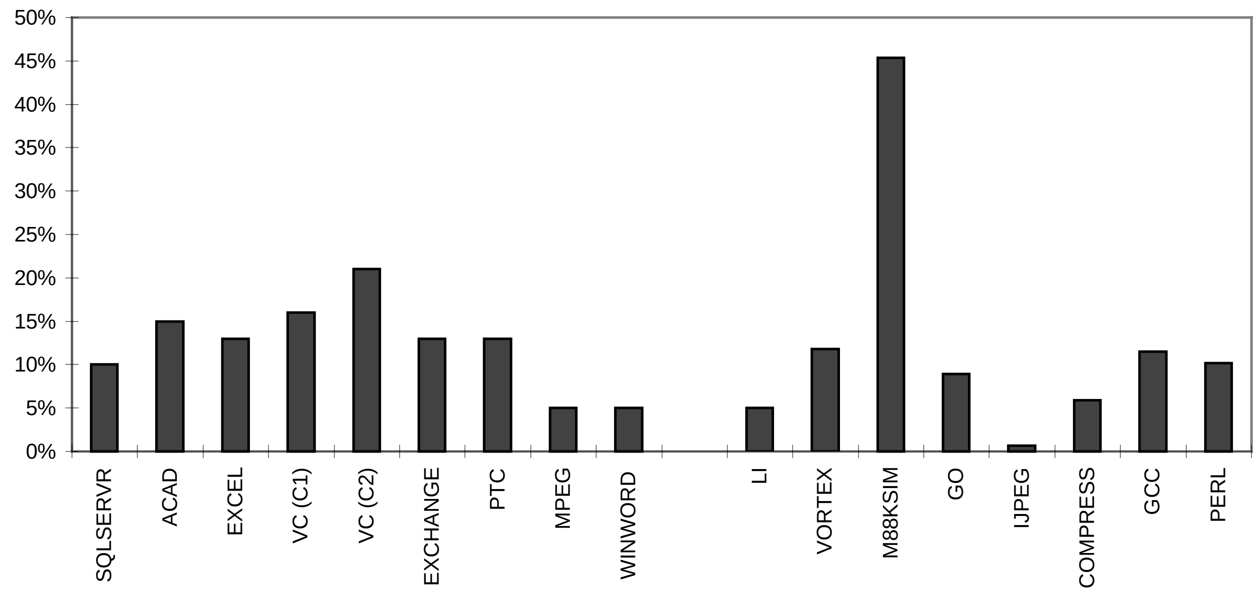

Figure 1: Speedup from code layout

line code; a simple greedy algorithm is used. By reducing the number of taken branches, the processor is able to fetch instructions efficiently and cache lines are better utilized. Next, each routine is separated into a hot and cold section. The hot section consists of the frequently executed basic blocks while the cold section contains the infrequently executed basic blocks. Finally, the hot routines are placed so that frequently called routines are near the caller and cold routines are collected at the end of the image. Splitting and placing routines reduces the chance that routines that call each other will have addresses that clash in the instruction cache.

When examining the hot paths through a routine, it is apparent that many instructions are executed on behalf of the cold paths and are therefore rarely necessary. Hot cold optimization (HCO) [Cohn96] exploits this opportunity. After Spike partitions a routine into hot and cold sections, HCO moves instructions that are dynamically dead or unnecessary in the hot section into the cold section. Stubs are introduced to hold compensation code where necessary.

Many of the opportunities exploited by HCO appear to be poor register allocation decisions, which could be improved with profile information. We are currently implementing a register allocator. Early results show path length reductions of up to 11% in the SPEC95 integer benchmarks.

4. Performance Results

Spike's performance is evaluated using a set of large NT applications that are typical of the applications run on a high performance personal computer, and the SPEC95 [SPEC] integer benchmarks. Table 1 describes each application. All of the programs are compiled with the same highly optimizing backend that is used on Alpha UNIX and VMS systems [Blickstein92].

Figure 1 shows the execution time reduction provided by the code layout optimization. The SPEC95 programs used the training data for profiles and speedups were measured on the reference data. The other programs were trained and measured on the same data. Some experimentation with Excel and VC has shown that the speedup is not very sensitive to training data, as long as it is chosen carefully. Spike speeds up most large applications by at least 5%, and often gets 10% or more. Programs that spend a significant amount of time in inner loops usually get the least benefit, but even the MPEG player has a 5% speedup

Figure 2 shows the speedup for each application after applying HCO, broken down by optimization. The measurements and some of the programs for HCO were done differently from the code layout measurements because they were collected at a different time. As noted in Table 1, the workloads for some of the applications in the HCO measurements were different from the workloads used for code layout. All the programs, including SPEC95, were trained and measured on the same data. The HCO speedups are measured as path

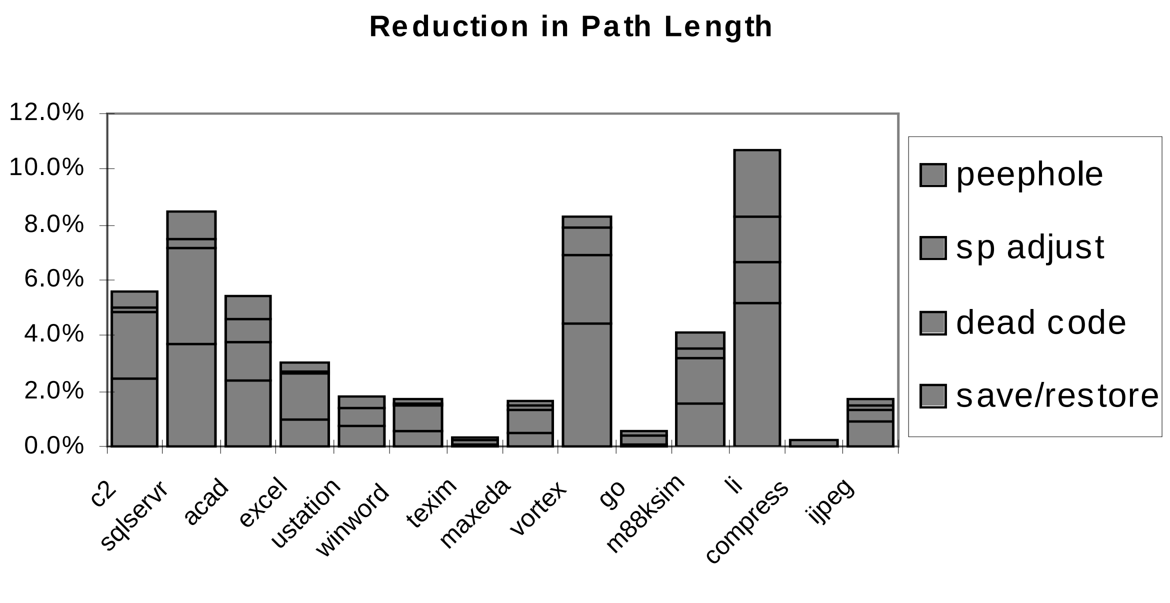

Figure 2: Reduction in path length for HCO broken down by optimization

length reductions, which is the reduction in the number of instructions executed. Using path length reduction allows us measure small effects that would otherwise be obscured by run to run variation. Speedups are in addition to those provided by code layout. For programs that do not spend most of their time in inner loops, HCO usually provides a 5% speedup. About half the speedup comes from removing dynamically dead code, which is moving operations out of frequently executed paths. Most of the rest comes from save/restore, using profile information to improve some register allocation decisions.

5. Summary and Conclusions

Spike is a complete system for optimizing Alpha/NT executables that has eliminated most of the barriers to using profile-directed optimization. Spike provides a simple to use graphical interface, making it easy for a user to instrument and optimize large applications consisting of multiple images. Transparent Application Substitution simplifies profile collection and image management, and persistent profile information enables old profiles to be used even after modifications are made to a program. The optimizations performed by Spike are effective in reducing execution time of large PC applications, producing speedups of 5-20% across a wide range of applications.

6. Acknowledgment

In creating Spike, we were inspired by the elegant solution to the profile-directed optimization problem implemented in FX!32 [Hookway97].

7. References

[Amitabh] A. Srivastava and D. Wall. “Link-Time Optimization of Address Calculation on a 64-bit Architecture,” *In Programming Language Design and Implementation*, Orlando, FL, June 1994.

[Atom] A. Srivastava and A. Eustace. “ATOM:A System for Building Customized Program Analysis Tools,” *In Programming Language Design and Implementation*, Orlando, FL, June 1994.

[Blickstein92] D. Blickstein, et al, “The GEM optimizing compiler system,” Digital *Technical Journal*, 4(4):121-136.

[Cohn96] R. Cohn and P. G. Lowney, “Hot Cold Optimization of Large Windows/NT Applications,” MICRO-29, pp. 80-89, Paris, France, December 1996.

[DCPI] http://www.research.digital.com/SRC/dcpi/

[Goodwin97] D. Goodwin, “Interprocedural dataflow analysis in an executable optimizer,” To appear in:

Conf. In Programming Language Design and Implementation, Las Vegas, Nevada, June 1997

[Hookway97] R. Hookway, "DIGITAL FX!32: Running 32-Bit x86 Applications on Alpha NT," *COMPCON*, San Jose, CA, Feb. 1997

[Hwu89] W.W. Hwu and P.P. Chang, "Achieving high instruction cache performance with an optimizing compiler," Proceedings 16th Annual Symposium on Computer Architecture, Jerusalem, Israel, June 1989

[Knuth73] D. Knuth, *The Art of Computer Programming*: Vol. 1, Fundamental Algorithms, Addison Wesley, 1973

[MORPH] http://www.eecs.harvard.edu/morph/

[Multiflow] Lowney et al. "The Multiflow Trace Scheduling Compiler," The Journal of Supercomputing, 7, pp. 51-142, 1993.

[Pettis90] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning" *in Proceedings ACM SIGPLAN Conf. on Programming Language Design and Implementation* '90, pp. 16-27, White Plains, NY, June 1990

[Pixie] MIPS Computer Systems. "UMIPS-V Reference Manual (pixie and pixstats)." Sunnyvale, CA, 1990.

[SPEC] http://www.specbench.org/

[Wilson96] L.S. Wilson, C.A. Neth, M.J. Rickabaugh, "Delivering binary object modification tools for program analysis and optimization," volume 8,1 of *Digital Technical Journal*, pages 18-31