Vulcan

Binary transformation in a distributed environment

Amitabh Srivastava
Andrew Edwards
Hoi Vo

April 20, 2001

Technical Report
MSR-TR-2001-50

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This report replaces MSR-TR-99-76, an earlier version of the same material.

Vulcan
Binary transformation in a distributed environment

Amitabh Srivastava
Microsoft Research
One Microsoft Way
Redmond, WA
amitabhs@microsoft.com

Andrew Edwards
Microsoft Research
One Microsoft Way
Redmond, WA
andred@microsoft.com

Hoi Vo
Microsoft Research
One Microsoft Way
Redmond, WA
hoiv@microsoft.com

ABSTRACT

Distributed computing on the Internet presents new challenges and opportunities for tools that inspect and modify program binaries. The dynamic and heterogeneous nature of the Internet environment extends the traditional product development process by requiring program development tools like these, which were once used only internally, to work in live environments too. The concept of compilation process must be expanded along with the capabilities of the binary tools. This paper presents *Vulcan*, a second-generation technology that addresses many of these challenges. Vulcan provides both static and dynamic code modification and provides a framework for cross-component analysis and optimization. It provides system-level analysis for heterogeneous binaries across instruction sets. Vulcan works in the Win32 environment and can process x86, IA64, and MSIL binaries. Vulcan scales to large commercial applications and has been used to improve performance and reliability of Microsoft products in a production environment.

1. INTRODUCTION

In recent years, binary instrumentation and optimization tools (hereafter called "binary tools") have been effectively used to understand and improve the performance of significant programs[23][24]. Because they are new, binary tools are typically not well integrated with the existing compiler framework. (They often rely on slightly modified executable formats, so that relocation information is retained, and code and data can be easily distinguished in the executable.)

At the same time, the dynamic and heterogeneous nature of Internet computing has challenged the traditional compilation model, presenting great opportunities to expand the role of binary tools in improving performance and reliability of software. Vulcan is a second-generation technology that addresses many of the challenges of this new generation of computing. To understand the role of Vulcan, we first describe how the traditional compilation framework has changed.

In the past, the compilation process has focused simply on turning source code into executables, balancing compilation speed against code optimization. The static compiler turns source file into object files, followed by the linker that combines the object files to produce the final executable. Very little program information is preserved after the link stage, mostly for debugging and support.

The following binary modification stage was not designed as part of the original compilation process. Binary tools "hacked" their way into the compilation process by intercepting and transforming executables that had already been compiled and linked. The binary modification stage thereby provided language independence and a natural environment for whole program analysis and architecture specific transformations without requiring recompilation.

The new distributed computing model of the Internet presents new challenges for software development tools: its heterogeneous nature forces applications to be built with components in multiple instruction sets, and its dynamic nature extends the traditonal product development process to live environments.

The heterogeneous nature of the Internet requires certain compilation phases like code generation and optimization to be delayed until run time. Programs can be compiled to architecture-independent languages like MSIL[1] (Microsoft Intermediate Language[19]), with final optimization and code generation performed at run time. As shown in Figure 1, parts of a heterogeneous application may still exist in MSIL while other parts may

[1] MSIL is Microsoft's intermediate language for the managed environment. A number of languages such as C#, VB, Cobol etc. can be compiled to MSIL. MSIL is converted to native code by JIT compilers.

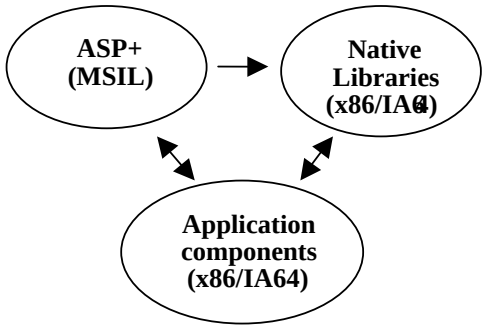

Figure1: Heterogenous application

already exist in the native instruction set. Architectures like IA-64 also let x86 binaries co-exist with IA-64 binaries. Finally, an application may be distributed over multiple machines with different instruction sets. In this case, each MSIL component will ultimately be translated to the instruction set of the local machine.

The heterogeneous environment of the Internet poses new challenges in understanding whole-system behavior: all components of an application in different formats must ultimately be analyzed within a single framework. Systems that have been ported to multiple architectures are not sufficient to analyze heterogeneous programs because each port of the system analyzes parts of the program independently on its architecture.

The Internet places strong demands for reliability, performance, and continuous operations in the presence of open-ended extensibility. As the complexity of the deployment environment cannot be fully reproduced inside our test labs, many development operations like debugging, program verification, and the identification of performance bottlenecks must occur on live systems. Because of the severe memory and time constraints in the deployment environment, binary tools will require both static and dynamic modification capabilities. Moreover, to operate in a live distributed environment, binary tools will require modification capabilities triggered from remote systems.

All our existing tools have been designed to operate at a particular stage of the compilation process and cannot be simply extended to operate elsewhere. For example, past binary systems like ATOM operate statically at post-link time and cannot operate under the stringent memory and time constraints of run time. Conversely, systems like Dynamo have been engineered to perform specific optimizations at run time with very low overhead, but lack the general capabilities granted by the more general infrastructures of static systems.

Vulcan is a second-generation infrastructure that is designed for research and development in the new Internet environment. Vulcan provides a single rich infrastructure that addresses the needs of the new environment without increasing the complexity of writing transformations and building tools. The key features of Vulcan include:

- Static and dynamic binary code modification capabilities.
- System-level analysis for heterogeneous programs. Vulcan currently works with binaries in x86, IA64, and MSIL in the Win32 environment.
- Uniform abstraction for different component types in heterogeneous environments. Vulcan provides a flexible API both for instrumentation and modification. Vulcan's API is similar in philosophy to the ATOM's API although ATOM provides API only for instrumentation.
- Dynamic modification of an executing program, triggered locally or from a remote machine in a distributed environment.
- Abstraction across multiple components to provide a single representation for cross-component optimization.
- Operation on large commercial applications like Microsoft Office, Windows 2000 and SQL 2000. Vulcan has been used to improve the performance and reliability of Microsoft products.
- Opportunities for new classes of transformations like partial compilation, mixed-instruction set binaries, and cross-component optimization in heterogeneous environments.

In this paper, we describe the design and implementation of Vulcan. We discuss its programming abstractions, its performance, and the new classes of transformations it enables.

2. THE ARCHITECTURE OF VULCAN

The Vulcan infrastructure is designed to address the development, testing, and operational requirements of product development. It provides a rich infrastructure for experimentation, rapid prototyping, and product implementation. Vulcan includes:

- A uniform representation for **heterogeneous systems** to simplify the definition of transformations and to allow whole-system analysis.
- **Externalized program representations** for use by new tools, analyses, and transformations in areas like optimization, correctness, testing, support, and debugging.
- Infrastructure for **cross-component optimizations** across component boundaries.

2.1 Heterogeneous systems

The basic design of Vulcan is based on the observation that although there are many different formats for encoding a program, conceptually each stores similar information like code, data, read-only data, symbols, *etc*. Vulcan builds high-level abstraction layers to uniformly represent different input forms. This eliminates

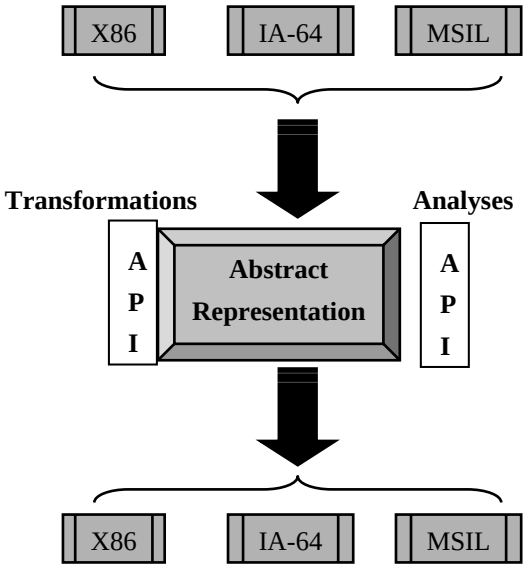

Figure 2: The Vulcan infrastructure

differences due to various encoding formats and provides a uniform view of the program to the user.

Figure 2 shows the basic architecture of Vulcan. Vulcan builds an abstract representation of the program from the application binaries. In this abstract representation, all addresses have been converted into logical pointers making the representation very malleable. Vulcan permits modification and analysis of the program representation by externalizing the abstraction.

To represent different instruction sets in the abstract representation, Vulcan uses a machine model consisting of infinite registers and a stack, with instructions like add, subtract, and push. This abstract representation can be queried for analysis and modified for instrumentation and optimization. (The original representation is also available for machine-specific analysis and modification.) Vulcan writes the final binary from the abstract representation. In the case of dynamic modification, Vulcan's input and output are attached to an executing process.

2.2 Externalized program representations

Vulcan presents a simple program representation for analysis and transformation. Opaque types expose its six basic abstractions: System, Program, Component, Procedure, Basic Block, and Instruction. A System in Vulcan is a collection of Programs. Each Program is a list of Components. Each Component has a symbolic representation consisting of symbols, data blocks and code. The Component also contains abstractions unique to different input formats, such as Win32's PE file format, where it abstracts the header information with methods to read, write, query and translate the contents.

Code in Components is represented as a linear list of Procedures; Procedures as a list of Basic Blocks; and Basic Blocks are a list of Instructions. Higher-level abstractions like control flow graphs (CFG) and data flow graphs (DFG) can be built using Vulcan's basic abstractions. As shown in Figure 3, each of these abstractions exports an API for navigation, query and

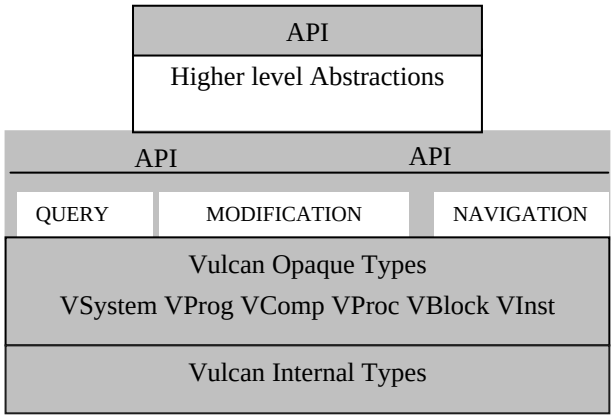

Figure 3: Vulcan program representations

modification. Vulcan also provides several additional abstractions such as Data Blocks to represent data, and Symbols to represent the elements of the symbol table (including the names of functions, parameters, and locals).

Appendix A shows the code of a peephole optimization that performs branch chaining using the Vulcan API. This code is architecture-independent and will work on a Program that contains different types of Components. Vulcan provides primitives for memory addresses, branches, and register contents that lets users write architecture independent code. For instance, Intel's x86 architecture has many different addressing modes for referencing memory while RISC architectures like Compaq's Alpha have a simple base register with a displacement. However, Vulcan's Effective Address primitive lets the user ignore the architecture specific details. Vulcan also automatically provides the memory address that is being referenced.

Vulcan can also dynamically modify a running application from a remote machine. To trigger branch chaining from a remote machine, only a main routine needs to be changed, as shown in Figure 4. The main routine uses the Open () primitive to obtain all the running

Programs (processes) on that machine, then iterates over all the Programs and invokes Peephole on each of them.

```
int main(int argc, char** argv) {
  // argument is machine name
   VSystem *pSystem = VSystem::Open(argv[1]);
   for (VProg *pProg=pSystem->FirstProg();
          pProg;
          pProg->Next())
        Peephole (pProg);
   return 0;
}
```

Figure 4: Branch chaining from a remote machine

Vulcan also exports an instrumentation interface for building tools. This interface allows a specified procedure call to be inserted before or after a Program, Component, Procedure, Basic Block, or Instruction. The user provides the code for this procedure in a separate library that is loaded and called as the program executes. This procedure can be passed standard arguments like int and char, as well as additional types including the contents of a register, an effective memory address, or whether a branch will be taken.

Vulcan records the transformations specified by the user and applies them only when the Commit () primitive is invoked. The user has full control on when and how often to invoke the Commit () primitive.

2.3 Cross-component optimization: merging multiple components

Applications are divided into Components—(dynamic linked libraries or dlls)— for several reasons. First, dlls enable code sharing, since a number of applications can share the same dll. Second, dlls enable small patch releases, since only dlls whose code has changed must be shipped as part of the patch release. Finally, dlls provide a unit to separate code shipped by different organizations. These issues argue for dividing the applications into a large number of dlls.

Unfortunately, dividing a program into a large number of dlls may also hurt performance. If optimizations are limited to a single dll, they are less effective when the dll is small; we cannot pack code on a single page for working set optimizations if the code is split between different dlls. Poor packing also degrades application startup time. Similarly, procedure calls that span dlls cannot be inlined. However, by merging dlls, these same optimizations can operate more effectively across component boundaries.

For program management reasons, the number of dlls that constitute an application is often decided early in the development process, and this decision can be hard to change. Vulcan provides the ability to merge multiple components late in the process, separate from management concerns, and provide a single program representation for the merged components. Transformations may now be applied to the merged component and achieve cross component optimization effects. The initial results are very promising and may be subject of a separate paper.

Vulcan can be used for cross component optimization without merging the components. This mechanism is discussed in Section 6.

3. VULCAN IMPLEMENTATION

This section describes how Vulcan statically modifies application binaries and dynamically transforms running application. These details are internal to Vulcan and are transparent to the user.

3.1 Static modification

In the static mode, Vulcan takes as input the components that make up the program. Depending on the type of each component, Vulcan invokes an appropriate reader to parse the binary file, disassemble the instructions, and build the abstract representation. Vulcan uses the available meta-information (stored in program database file or PDB) to construct the program representation. The PDB file contains procedure names, symbol table information, variable type information, *etc*. Vulcan can build the representation lazily and at low cost, without incurring the dynamic translation overhead of other schemes that transform executables[26]. Vulcan discovers the basic block boundaries within functions and also determines which parts of the binary are statically dead. The representation of each basic block is just a pointer to the raw bytes of the image. It also converts physical addresses to logical addresses to facilitate manipulation. Vulcan also creates a link between each block and its logical successor block, so that reordering the blocks will not affect the logic of the program.

Vulcan maintains the correspondence between the abstract instructions and the original bytes in the binary so that applications can query for the original bytes. It initially keeps each instruction close to its original form as many program analysis tools and simulators require access to instructions in their original form. This also allows efficient translation of instructions in the final assembly phase. When optimizers do not require the original architecture-specific instructions, they may convert them into architecture-independent instructions. By converting the instructions, the representation loses

the one-to-one mapping with the original form but allows more architecture-neutral analyses and transformations.

Once the representation has been built, it can be queried for analysis and modified for instrumentation and optimization. Vulcan builds the abstraction lazily, so a tool that just inserts probes at procedure level or reorders basic blocks will not incur the cost of the full instruction abstraction.

After all the transformations have been applied to the program, Vulcan generates the updated binary from the abstract representation in the appropriate instruction set. Vulcan adds the necessary code to preserve the logical control flow. Vulcan computes the addresses, assembles instructions, writes the new binary file, and also creates an updated PDB file so the updated program can be debugged and even read back into Vulcan.

3.2 Dynamic modification

Vulcan can also dynamically modify running applications. As in the static mode, Vulcan builds the abstract representation lazily. It reads the executing image directly from the memory of the target process to build its abstraction. Modifying executing programs in the presence of multiple threads and variable-size instructions requires special treatment. This section describes how Vulcan addresses these cases in an efficient manner.

Instructions and basic blocks cannot be safely moved around in a running program because their addresses may have already been stored in dynamic data structures. If the transformation does not increase the instruction size or the basic block size, they do not have to be moved. Vulcan detects such cases and appropriately modifies the instructions in place. In presence of multiple threads, this change must be done atomically; Vulcan suspends all the threads in the process before making changes to the instruction.

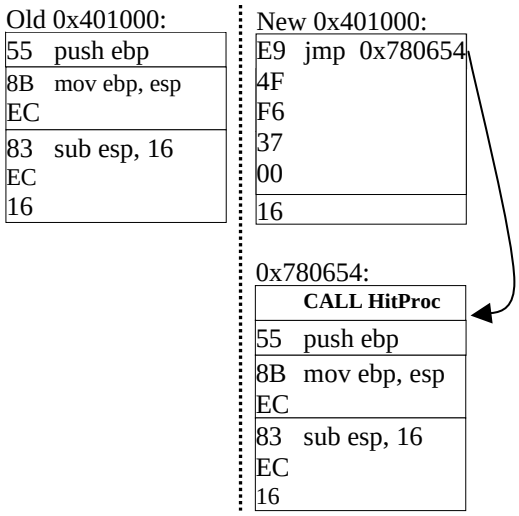

Figure 5: Dynamically inserting a procedure call

Modified basic blocks may not fit in the original space, and modified variable-width instructions may not fit in the space occupied by the original instructions. In this case, Vulcan creates a copy of the procedure or the block[2] and then links that modified block into the target process. Vulcan also redirects control flow to the copy of the procedure or block by replacing its beginning with a jump to the modified copy, as shown in Figure 5. This replacement is also done atomically while all of the threads of the target process are stopped. If any of the threads were executing inside the code being replaced, Vulcan fixes the thread before restarting all of the stopped threads.

Vulcan supports iterative profiling by allowing instrumentations probes to be added and removed at different times. Vulcan allows probes to be inserted and removed at Procedure, Basic Block, and Instruction granularity. PARADYN [16] allows calls to be inserted only at procedure granularity.

3.3 Remote dynamic modification

Vulcan can also dynamically modify applications running on a remote machine. As described in the previous section, Vulcan normally builds the abstract representation directly from the executing image, but Vulcan can also build the abstract representation from the binary of the running application on disk. However, this requires an additional step of relocating the program to the address of the running program.

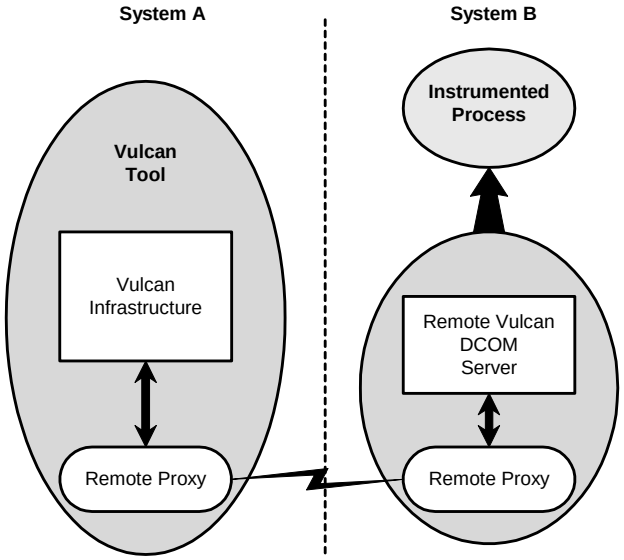

Figure 6: Modification from a remote machine

As shown in Figure 6, reading memory from the remote process is handled through a distributed common object model interface (DCOM). DCOM handles cross-machine as well as cross-platform communication, and it also

[2] On x86, the common technique is to use the one-byte **int 3** instruction that generates an interrupt. As it is 1-byte it can eliminate growing the basic block but it introduces large time overheads and complexities in interrupt handling.

enforces security. The DCOM interface for remote Vulcan is small (around a dozen methods) and allows remote reading and writing of memory ranges, as well as enumerating processes, threads, and components. It also lets Vulcan create, suspend, and adjust threads. (This interface is completely implemented using public and documented Win32 APIs.) This process requires a Vulcan proxy library on the machine where the application is executing.

4. VULCAN PERFORMANCE

To measure the performance of Vulcan, we tested Vulcan against four large commercial applications: WinWord, a word processing application, Excel, a spread sheet application, Mso9, a shared code library, and SQL, a database application. Figure 7 gives the details of each of the programs: number of basic blocks, number of instructions, size of the binary and its associated PDB file.

Program	Winword	Excel	Mso9	SQL
Procs	24k	25k	35k	41k
Blocks	547k	563k	359k	438k
Insts	2,039k	2,023k	1,377k	1,767k
exe (MB)	8.1	6.5	5.2	6.6
pdb (MB)	22	17	22	20

Figure 7: Program details

For each application we measured the peak working set and the time it took Vulcan to read and write the application binary. These measurements do not include the additional memory and time taken by the user's transformation.

The peak working set of Vulcan, as shown in Figure 8, for building a complete abstract representation of an application, then writing it out, is about five to eight times the size of the executable. As Vulcan normally operates lazily, Vulcan's memory and time overhead will be lower for many transformations. Figure 8 also shows the peak working set is 10-30% lower if Vulcan does not build the Instruction abstraction.

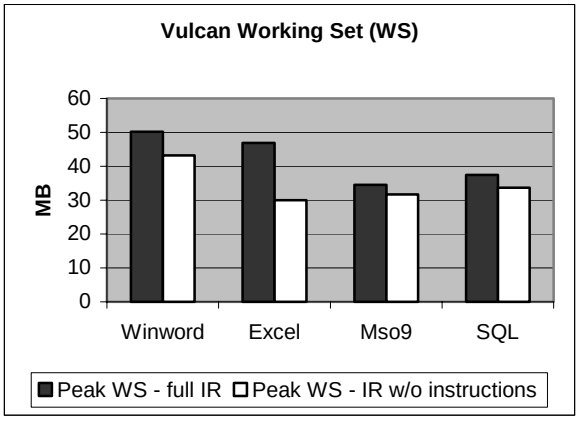

Figure 8: Vulcan Working Set

Vulcan's Instruction abstraction can dominate its memory requirements; having all occurrences of the same instruction sharing the same representation, reduces this. Figure 9 shows that only 17-21% as many instructions need to be represented in these large applications, containing 1.3 million to 2.0 million instructions[3]. This is why we did not see a bigger reduction in working set in Figure 9 when Vulcan did not build the Instruction abstraction.

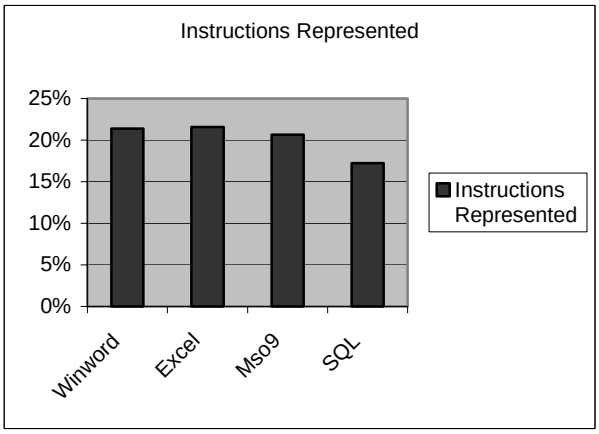

Figure 9: Unique instructions in applications

[3] All pc-relative instructions are assumed unique.

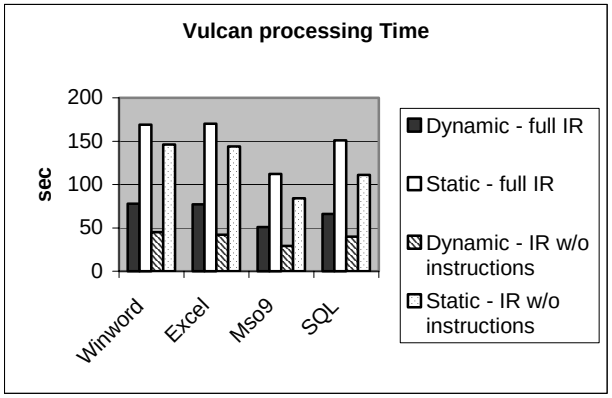

Figure 10: Vulcan processing time

Figure 10 shows the time taken by Vulcan to process an application both in static and dynamic mode. In the static case, it includes the time taken to read the binary, build the abstract representation, link the binary and write the binary to disk. The figure also shows the time if the abstract representation was built without the Instruction abstraction. For the dynamic modification, there is no link stage since the modification can be made incrementally at very low cost, and an updated binary does not have to be written to disk. Figure 10 shows the time Vulcan takes in dynamic mode both with and without the Instruction abstraction.

5. VULCAN STATUS

Vulcan currently works in Win32 environment and can process x86, IA64, and MSIL binaries. Vulcan can process large Microsoft applications such as the Microsoft Office suite, Windows NT system components, and SQL Server.

Vulcan also provides a library of higher-level abstractions that includes control-flow graphs, data-flow graphs, call graphs, and inter-procedural flow graphs. A wide variety of tools have been built using Vulcan to improve performance and reliability of Microsoft products. Some uses of Vulcan are enumerated below.

Program analysis and measurement: Vulcan has been used for building various profiling tools like basic block counting, edge-count profiling and whole-program path profiling**Error! Reference source not found.**. Vulcan has been used to profile parameters in procedure calls [10]. Vulcan is also used for understanding other aspects of programs, such as data reference profiling.

Program optimization: Various optimizations have been implemented using Vulcan. These include optimizations for code locality, data locality, procedure inlining, and cross component optimizations. Vulcan has also been used for architecture specific optimizations [1] and for building dynamic optimizations [4].

Software management: Vulcan has been used to build binary matching algorithms. Binary matching [25] identifies parts of a binary that has changed from a previous version. This information is useful in building smaller patches, prioritizing test cases for test case selection, and mapping stale profile data to new binaries.

Vulcan has also been used for code obfuscation and software watermarking.

Reliability: Vulcan has been used to improve the reliability of software. Vulcan-based tools can measure code coverage and improve test effectiveness by injecting faults. A tool for detecting instruction hazards in IA64 binaries [6] has also been built using Vulcan.

6. FUTURE WORK

Vulcan enables new classes of transformations to be built because it allows transformations to analyze multiple components in different forms, and programs to be modified both statically and dynamically. We describe some of these new transformations in this section. This is work in progress and the results will be the subject of future papers. These transformations will be important in the new Internet environment.

6.1 Partial compilation and mixed-instruction set binaries

In heterogeneous environments, programs are often compiled into architecture-independent languages like MSIL. This allows code generation to be delayed until the characteristics of the machine have been determined at runtime. Rather than compiling the entire binary to the native instruction set, it can be beneficial to compile performance-critical parts to native code while keeping the rest in a more compressed representation.

Vulcan can also be used to generate mixed-instruction set binaries. For example, IA-64 binaries are much larger than x86 and MSIL binaries. Gwennap [8] therefore suggests mixing IA-64 code with x86 code. Gwennap's analysis suggests that by carefully choosing which routines to code in IA-64 and which to leave in x86, applications may achieve more than 90% of full native performance while increasing code size by only about 20% over x86.

By using profile information from previous runs or from the current run, Vulcan can choose to compile time-critical routines to native code while keeping infrequently used routines in MSIL. Vulcan can process the binaries statically at install time or post-run time[4], or dynamically

[4] Post run time occurs between the different execution runs of the application. It has lesser time and memory constraints than run time.

at run time on the client machine. User-specific profile information could also be collected dynamically using Vulcan.

6.2 Cross component optimization: without merging components

Merging dlls for cross-component optimization has disadvantages if a particular dll is a candidate to merge with in more than one set of dlls. The dll would have to be assigned to a particular set to break the tie. However, Vulcan provides another mechanism for cross module optimization that does not require merging components: Vulcan can simultaneously read and represent multiple components. Without merging, the transformation can analyze all the components and build its own data structures across component boundaries. The transformation may update any of the representations as needed. Vulcan will write the updated components. For example, cross component inlining can be implemented[5].

7. RELATED WORK

A large number of systems have been developed to analyze and optimize homogeneous programs. Common analysis techniques used are address tracing, binary code modification, and simulation. Some related systems are discussed in this section.

Pixie [17] started a class of basic block counting tools by inserting a fixed sequence of instructions into basic blocks to record their execution frequencies. Epoxie [26] extended the technique by using relocations to eliminate dynamic translation overheads. QPT [12] further extended the technique by reducing the number of basic blocks that need to be instrumented. Purify [9] instruments memory references to detect out-of-bounds memory accesses and memory leaks.

OM [23][24] allows general transformations to be applied to a binary by converting the binary to an intermediate representation that can be easily manipulated. It used a Register Transfer Language (RTL) as a machine-independent representation and has been implemented on MIPS[17], Alpha[21], and x86 architectures. EEL [13] uses a similar technique and provides an editing library for SPARC architectures. Spike [7] and Alto [5] are optimizers for the Alpha architectures. Dynamo [3] dynamically optimizes HP PA-RISC binaries at run time.

ATOM [22] extends OM by providing a flexible instrumentation interface for Alpha and Intel x86 systems, but does not support modification of binaries. Etch [20] provides a similar instrumentation system for x86 as does BIT [14] for Java byte codes. PARADYN [16] allows dynamic instrumentation of a running process and has been ported to several architectures. It allows calls to be inserted only at procedure granularity.

All of the above systems operate at a particular point in the compilation process and cannot be easily extended to other phases. Moreover, none of the previous systems work in heterogeneous environments. Some of them have been ported to multiple architectures, but each port operates as an independent system.

8. CONCLUSION

Vulcan provides a rich infrastructure for the next generation binary tools that address the requirements of the new Internet programming paradigm. Vulcan allows whole-system analysis in heterogeneous environments where components are held in different forms. Vulcan can perform both static and dynamic code modification. Its resource usage scales with the complexity of the transformation. Vulcan-based tools have been used for research and development on large commercial applications.

Vulcan's general infrastructure also opens opportunities for building new transformation like partial compilation, iterative profiling, and cross-component optimization. Some of these topics will be the subject of future papers.

ACKNOWLEDGEMENTS

Many people have helped bring Vulcan to its current stage: the original Vulcan team included Bruce Kuramoto, David Gillies, Greg Eigsti, Hon Keat Chan, John Lefor, John Liu, Ken Pierce, Richard Shupak and Ronnie Chaiken. Robert Bickford, Carlos Gomes, William Frank, and Jay Finger later joined the effort. John Liu implemented the MSIL input to Vulcan. Ronnie Chaiken, David Gillies, and Robert Bickford implemented the IA64 input and the library for higher-level abstractions. A large number of early adopters patiently worked with us and contributed to the richness of Vulcan. Our special thanks to the summer interns who over the past two years continuously found innovative ways to use Vulcan. John DeTreville, Ben Zorn, and Scott McFarling provided perceptive comments.

[5] For cross module inlining, both the components will have to be updated when either of the components is updated with newer versions.

APPENDIX A: VULCAN API EXAMPLE

We illustrate the Vulcan API by showing its use in writing a peephole optimization. Figure 11 shows the code for a peephole optimization that performs branch chaining. The program to be optimized can be passed as an argument to the *main* routine. As Vulcan can operate both statically and dynamically, the argument to the main routine can be a program or a process identifier of a program that is already running.

After processing the arguments, main creates the Program abstraction given the program name or process id. *Peephole* is called to optimize the Program. Peephole has four nested loops: one iterating over the Components in the Program, one iterating over the Procedures of each Component, one iterating over the Basic Blocks in each Procedure, and the innermost iterating over the Instructions in each Basic Block. To navigate an abstraction, Vulcan provides the First(), Next(), and Prev() methods. *Peephole* examines each Instruction, and if it is an unconditional branch, *BranchChain* is called to chain the branches. BranchChain finds the target Basic Block of an unconditional branch using the BlockTarget() method. If the first instruction of the target Basic Block is also an unconditional branch, the branches are chained. Finally, after all the transformations have been performed, the Component is written out using the Write() primitive.

```
void BranchChain (VInst* pInst) {
  for(;;pInst = pInstTarget) {
      VBlock *pBlkTarget = pInst-> BlockTarget();
      VInst *pInstTarget = pBLkTarget-> FirstInst();

      if (COp::IsUnCondBranch(pInstTarget->Opcode()))
          pInst->SetBlockTarget( pInst-> BlockTarget());
      else
          break;
   }
}
```

```
void Peephole(VProg* pProg) {
  for(VComp *pComp = pProg->FirstComp();
        pComp; pComp = pComp->Next()) {
     for(VProc *pProc = pComp->FirstProc();
           pProc; pProc = pProc->Next()) {
       for(VBlock *pBlk = pProc->FirstBlock();
             pBlk; pBlk = pBlk->Next()) {
         for(VInst *pInst = pBlk->FirstInst();
               pInst; pInst = pInst->Next()) {
           if
(COp::IsUnCondBranch(pInst.Opcode()))
               BranchChain(pInst);
         }
       }
     }
     pComp->Write();
  }
}

int  main(int argc, char** argv) {
  VProg *pProg;
  if (IsProcessId(argv[1])) // Is argument a live process
     pProg = VProg::Open( GetProcessId(argv[1]) );
  else // or the name of a program?
     pProg = VProg::Open(argv[1]);
  Peephole(pProg);
  return 0;
```

Figure 11: Branch Chaining

9. REFERENCES

[1] Ronald Barnes, R. Chaiken and D. M. Gillies, Feedback-Directed Data Cache Optimizations for the x86, *2nd ACM Workshop on Feedback Directed Optimization (FDO)*, November 1999.

[2] B. Buck and J. K. Hollingsworth, An API for Runtime Code Patching, *Journal of Supercomputing Applications (to appear)*, 2000.

[3] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: A Transparent Runtime Optimization System, *Proceedings of SIGPLAN' 97 Conference on Programming Language Design and Implementation*, June 2000.

[4] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, David M. Gillies. Mojo: A Dynamic Optimization System. Submitted for publication, October 2000.

[5] K. De Bosschere and S. Debray. Alto: a Link-Time Optimizer for the DEC Alpha. Technical Report TR-96-16, Computer Science Department, University of Arizona, 1996.

[6] David M. Gillies. Halo: A Hazard Location Tool for IA64. MSR-TR-2000-37, April 2000.

[7] David W. Goodwin. Interprocedural Dataflow Analysis in an Executable Optimizer. *Proceedings of SIGPLAN' 97 Conference on Programming Language Design and Implementation*, June 1997.

[8] Linley Gwennap. Intel's Merced and IA-64: Technology and Market Forecast. MDR Technical Library Special Report, 1998.

[9] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. *Proceedings of Winter Usenix Conference*, January 1992.

[10] John Kalamatianos, David Kaeli, and Ronnie Chaiken. Parameter Value Locality of Window NT-based Applications. *1st ACM Workshop on Feedback Directed Optimization (FDO)*, November 1998.

[11] James Larus. Whole program paths, *Proceedings of SIGPLAN' 99 Conference on Programming Language Design and Implementation*, 1999.

[12] James Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software, Practice and Experience*, vol 24, no 2, pp 197-218, February 1994.

[13] James Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. *Proceedings of SIGPLAN' 95 Conference on Programming Language Design and Implementation*, June 1995.

[14] Han Lee and Benjamin Zorn. BIT: A Tool for instrumenting Java bytecodes. *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997.

[15] James Larus. Whole Program Paths. *Proceedings of SIGPLAN' 95 Conference on Programming Language Design and Implementation*, May 1999.

[16] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* **28**, 11, pp.37-46, November 1995.

[17] MIPS Computer Systems, Inc. *Assembly Language Programmer's Guide*, 1986.

[18] Andy Padawer, Microsoft P-code technology. 1992. In Microsoft Developer Network, http://premium.microsoft.com/msdn/library/background /html/msdn_c7pcode2.htm

[19] Jeffery Richter. Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web. http://www.microsoft.com/DirectAccess/products/microsoft.net/framework.asp

[20] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Chen, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[21] Richard L. Sites, ed. Alpha Architecture Reference Manual, Digital Press, 1992.

[22] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *Proceedings of SIGPLAN' 94 Conference on Programming Language Design and Implementation*, June 1994. Also available as WRL Research Report 94/2, March 1994.

[23] Amitabh Srivastava and David Wall. A Practical System for Intermodule Code Optimization at Link Time. *Journal of Programming Language*, 1(1):1-18, March 93.

[24] Amitabh Srivastava and David Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. *Proceedings of SIGPLAN' 94 Conference on Programming Language Design and Implementation*, June 1994. Also available as WRL Research Report 94/1, March 1994.

[25] Zheng Wang, Ken Pierce, Scott McFarling. BMAT – A Binary Matching Tools for Stale Profile Propagation. *The Journal of Instruction-Level Parallelism*, vol. 2, March 2000. Also available as MSR-TR-99-83, November 1999.

[26] David W. Wall. Systems for late code modification. In Robert Giegrich and Susan L. Graham, eds, Code Generation – Concept, Tools Techniques, pp. 275-293, Springer-Verlag, 1992.