Adaptable Binary Programs

Susan L. Graham Steven Lucco Robert Wahbe

Computer Science Division, 571 Evans Hall
UC Berkeley, Berkeley CA, 94720

Abstract

To accurately and comprehensively monitor a program's behavior, many performance measurement tools transform the program's executable representation or *binary*. By instrumenting binary programs to monitor program events, tools can precisely analyze compiler optimization effectiveness, memory system performance, pipeline interlocking, and other dynamic program characteristics that are fully exposed only at this level. Binary transformation has also been used to support software-enforced fault isolation, debugging, machine re-targeting and machine-dependent optimization.

At present, binary transformation applications face a difficult trade-off. Previous approaches to implementing robust transformations result in significant disk space and run-time overhead. To improve efficiency, some current systems sacrifice robustness, relying on heuristic assumptions about the program and recognition of compiler-dependent code generation idioms. In this paper we begin by investigating the run-time and disk space overhead of transformation strategies that do not require assumptions about the program's control flow or register usage. We then detail simple information about the binary program that can significantly reduce this overhead. For each type of information, we show how it enables a corresponding type of binary transformation. We call binary programs that contain such enabling information *adaptable binaries*. Because adaptable binary information is simple, any compiler can generate it. Despite its simplicity, adaptable binary information has the necessary and sufficient expressive power to support a rich set of binary transformations.

We evaluated a prototype adaptable binary transformation system under the Ultrix 4.2 operating system and the MIPS processor architecture. Using the C SPEC92 benchmarks, we assessed adaptable binaries in two ways. First, we demonstrated that the information necessary to build adaptable binaries can be compactly recorded, increasing space overhead by only 9% for the SPEC92 benchmarks. Second, we measured the run-time overhead of previous approaches to implementing robust binary transformations, and showed that adaptable binaries enable a significant reduction of this overhead.

1 Introduction

Program development and monitoring tools are frequently implemented in two parts. The first part *instruments* a target program, transforming that program to monitor its behavior. The transformed program interacts with the second part of the monitoring tool, a run-time library that records information and takes action in response to events in the transformed program. To accurately and comprehensively monitor a program's behavior, many monitoring tools must instrument

[0]This work was supported in part by the National Science Foundation (CDA-8722788), Advanced Research Projects Agency (ARPA) under grant MDA972-92-J-1028 and contracts DABT63-92-C-0026 and N00600-93-C-2481. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Email: {`graham, lucco, rwahbe`}`@cs.berkeley.edu`

the program's executable representation or *binary*. For example, to measure memory system performance, a profiling tool must be aware of register spilling decisions made by the high-level language compiler, and instruction organization decisions made by the linker. At the binary level, all such decisions have been resolved. By transforming binary programs to monitor program events, tools can precisely analyze compiler optimization effectiveness, memory system performance, pipeline interlocking, and other dynamic program characteristics that are fully exposed only at this level.

Instrumentation of programs at the binary level also simplifies the engineering of program monitoring tools. Binary instrumentation can be made compiler-independent, facilitating the analysis of programs written in different high-level languages. Applying transformations to the binary eliminates the complexity and cost of recompilation. Further, by applying transformations to the binary rather than higher-level representations, instrumentation code cannot alter compilation decisions. Finally, binary transformation can be applied to code, such as system libraries, for which source is typically unavailable.

For these reasons, a number of performance analysis tools are implemented as binary transformations [BL92, BKW90, Digc, GH90, Wal91]. For example, `pixie` calculates instruction frequencies and floating point interlocks based on profile data and the instruction sequence of each basic block [Digc]. `QPT` uses control flow analysis to support collection and compression of address traces and profiles [BL92]. To aid programmers in identifying memory hierarchy bottlenecks, `MTool` compares actual and estimated execution times for different regions of a program [GH90]. Borg, Kessler, and Wall use binary transformation to generate and analyze very long multi-program address traces [BKW90].

Binary transformation has also been used to support software-enforced fault isolation, debugging, machine re-targeting and machine-dependent optimization. By inserting code to efficiently monitor indirect control transfers and memory updates, software-enforced fault isolation insures that program errors in one module do not corrupt data in other modules [WLAG93]. Instrumented memory references have been used to implement data breakpoints, detect memory leaks, and trap reads of uninitialized data [HJ92, Cen, WLG93]. Several systems have used binary transformation to re-target a program to a new architecture [HB89, SCK+93, Ech92, BKKM87]. Finally, optimizations such as code motion, dead-code elimination, register allocation, and instruction scheduling have been applied to binary programs, exploiting the global information available at the binary level [Joh90, SW93, Wal86, Wal92].

At present, binary transformation applications face a difficult trade-off. Robust and compiler-independent techniques for disassembly, insertion of instrumentation code, and obtaining free registers for use by the instrumentation code incur significant disk space and run-time overhead. To improve efficiency, some systems sacrifice robustness, relying on heuristic assumptions about the program's control flow and register usage and recognition of compiler-dependent code generation idioms. This reliance on heuristic information limits the scope and effectiveness of a binary transformation application.

In this paper we begin by investigating the run-time and disk space overhead of transformation strategies that do not require assumptions about the program's control flow or register usage. We then detail simple information about the binary program that can significantly reduce this overhead. We refer to binaries that include this information as *adaptable binaries (AB)*.

Adaptable binaries contain three classes of information. First, *control information* allows transformation tools to distinguish code from data, identify basic blocks, and identify targets of indirect control transfer instructions. Second, transformation tools can use *relocation information* to insert, delete, and reorder machine instructions. Finally, *register information* helps identify free registers that are available for use by instrumentation code.

We have implemented and evaluated adaptable binaries under the Ultrix 4.2 operating system and the MIPS architecture. Using the C SPEC92 benchmarks, we evaluated adaptable binaries in two ways. First, we measured the additional disk space required by adaptable binaries. We show that the information necessary to support adaptable binaries can be compactly recorded, increasing space overhead by only 9% for the SPEC92 benchmarks. For comparison, the average space overhead of the standard symbol table included in unstripped executable files is 81%. Debugging information increased executable file size by an average of 123%.

Second, we measured the run-time overhead of previous approaches to implementing robust binary transformations, and showed that adaptable binaries significantly reduce this overhead. These measurements of the basic binary transformation operations show that adaptable binaries can significantly improve the performance of many binary transformation applications.

The rest of the paper is organized as follows. Section 2 defines the fundamental operations required by binary transformation applications and presents the overhead of different robust implementation techniques. Section 3 details the information and program analysis required to build adaptable binaries and discusses our prototype implementation. Section 4 surveys the related work in this area.

2 Implementation Issues

In this section, we present transformation operations required by essentially all binary transformation applications and outline a number of corresponding implementation strategies. We demonstrate that, because existing binaries lack control, relocation and register information, these strategies must rely on conservative assumptions about the program. In the next section we define adaptable binaries, our solution to addressing these issues. Adaptable binaries contain the minimum information required for efficient and robust binary transformation applications.

To quantify the impact of different implementation strategies, we measured the run-time *transformation overhead.* Transformation overhead is the amount of time spent executing instructions that support the instrumentation code. For example, instructions that save and restore registers in order to obtain them for instrumentation purposes are counted as contributing to the transformation overhead. Neither the time spent in the original program nor the time spent in instrumentation code is part of the transformation overhead. Hence, transformation overhead isolates the runtime cost of supporting binary transformation operations.

Our experiments used the C SPEC92 benchmarks and were performed on a DecStation 5000/240 with 32 Megabytes of memory. The standard system libraries were not instrumented.

2.1 Disassembly

The fundamental difficulty in disassembling programs is reliably distinguishing code from data. A number of compilation environments place data in the binary's *code segment.* Under some object formats, since only the code segment is made read-only and sharable, it is a natural place for read-only data such as program constants. Unfortunately, many systems continue this practice even under object formats that provide appropriate types of *data segments.* If data is mistaken for code, the transformed program will be incorrect. If code is mistaken for data, parts of the program will not be instrumented. This problem is exacerbated on architectures with variable length instructions; code sequences are not constrained to begin on word boundaries, making disassembly dependent on the assumed starting point of the code.

In the presence of indirect control transfers, there is no robust method for distinguishing data from code. Most control transfer instructions have statically resolvable targets; however, the target of an indirect control transfer instruction is specified via a register and can only be determined at runtime. Common programming language abstractions such as function pointers, case statements, and continuations are typically implemented using indirect control transfer instructions.

On architectures with word-aligned fixed length instructions, a clever and robust solution to the problem of data in the code segment, first employed by `pixie`, is to duplicate the code segment, and instrument only the duplicated code segment. The duplicated code segment is assumed to contain only code and thus some data may be instrumented. Except for immediates in control transfer instructions, all addresses are left unaltered. Thus, all load and store addresses will reference data in the original code segment. Indirect control transfer instructions, on the other hand, must have their addresses *dynamically relocated* to the instrumented code segment. Dynamic relocation uses a translation table, built at transformation time, that maps old target addresses to new target addresses. The relocation table is the same size as the original code segment. Why not just leave addresses used to access data in the text segment unaltered and adjust addresses used for control transfers? Because it is not possible, without the use of heuristics, to determine at transformation time whether an address is going to be used as a data address or as a control address. The addition of the duplicated code segment and the dynamic relocation table triples the size of the binary.

Figure 1 gives the code sequence used to perform dynamic relocation. The extra memory access instructions are necessary to obtain a scratch register. We inserted this code sequence before

Benchmark		DYN-RELOC	OUT-OF-LINE
`052.alvinn`	FP	55%	112%
`026.compress`	INT	18%	104%
`056.ear`	FP	28%	98%
`023.eqntott`	INT	20%	86%
`008.espresso`	INT	10%	102%
`001.gcc1.35`	INT	58%	154%
`022.li`	INT	49%	170%
`072.sc`	INT	34%	71%
Average		**34%**	**112%**

Table 1: Transformation overhead, relative to native execution time, for dynamic relocation and out-of-line insertion strategies for inserting instrumentation code before each load and store in the program.

`Memory` ← `temp-reg`
Save temporary register `temp-reg`.
`temp-reg` ← `target-addr - code-start-addr`
Calculate offset of current jump target address.
`temp-reg` ← `temp-reg + table-base-addr`
Add jump target address offset to base of dynamic relocation table.
`target-reg` ← `Memory[temp-reg]`
Load new jump target address from relocation table.
`temp-reg` ← `Memory`
Restore temporary register `temp-reg`.

Figure 1: Assembly pseudo code for dynamic relocation. Without register information to determine which registers are live, it is necessary to save and restore a temporary register. If the `code-start-addr` and `table-base-addr` are known at transformation time, one arithmetic instruction could be saved.

every indirect control transfer in the original version of each of the programs listed in Table 1. The DYN-RELOC column of Table 1 shows that the average performance overhead of this approach was 34% on our example programs.

On architectures with variable length instructions, such as the Intel x86, the solution employed by `pixie` does not work, since correct disassembly depends on the assumed starting point of each code fragment. Robust transformations on architectures with variable length instructions require *dynamic disassembly*. The transformation system must monitor indirect control transfer instructions, performing dynamic disassembly at previously unseen target addresses. In a study by Cmelik and Keppel, overhead for dynamic disassembly on the SPARC architecture was approximately 600% for a small set of C programs [CK93].

2.2 Edit Operations

Edit operations are used to insert, delete, and reorder machine instructions. These operations are fundamental to all binary transformation applications, since by definition such applications modify the program to alter or monitor its behavior. Edit operations may change the addresses of instructions and data objects, requiring that all references to these objects be updated. Computed addresses and instruction addresses stored in the data segment cannot be reliably identified and updated statically. For example, a common implementation strategy for case statements is to use a *jump table* stored in the data segment. The jump table stores the code address for each arm of the case statement. There is no reliable way to distinguish a jump table from other kinds of data.

A simple solution to this problem, employed by

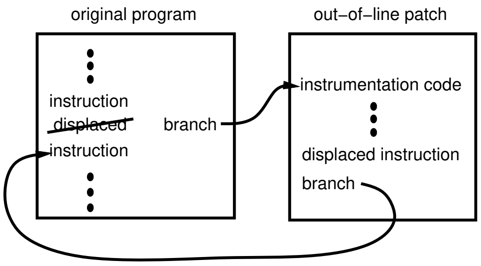

Figure 2: Simple example of out-of-line insertion of instrumentation code.

a number of applications [Digc, LB92, Wal91], is to use the dynamic relocation technique described above. As shown in Table 1, the average run-time overhead of this approach was 34% on our example programs. In addition to the space and run-time overhead of dynamic relocation, system that employ dynamic relocation must deal with indirect function calls from uninstrumented system codes. For example, the `signal` system call in UNIX takes the address of a signal handler. Since dynamic relocation necessarily leaves addresses unaltered until translation at the indirect control site, the address passed to the `signal` system call will be the *old* address of the signal handler. Since the `signal` code can not be instrumented, no dynamic relocation is performed and the application will have incorrect behavior.

As an alternative to directly inserting instrumentation code, a binary transformation tool can use *out-of-line insertion* to transfer control to instrumentation code without altering existing instruction addresses [Kes90]. To logically insert instrumentation code before a particular instruction, the instruction is replaced with a control transfer instruction to the instrumentation code. Before returning to the original instruction stream, the displaced instruction is executed. Figure 2 depicts a simple example of out-of-line insertion.

The OUT-OF-LINE column of Table 1 gives the transformation overhead for inserting instrumentation code before every load and store instruction. On average, the OUT-OF-LINE incurs 112% execution time overhead.

2.3 Register Operations

Instructions inserted by a binary transformation tool may need to use machine registers to compute intermediate values or memory addresses. We call such registers *temporary registers* because they are not live across instructions from the original program. Some applications, such as the memory system simulation program `MemSpy`, require a large number of temporary registers [MGA92]. In addition, many binary transformation applications can benefit from registers *reserved* for their exclusive use. For example, `pixie` uses a reserved register to hold the base of its dynamic relocation table. Wahbe, Lucco, and Graham show that reserving registers for a data breakpoint facility can significantly reduce run-time overhead [WLG93].

Temporary registers are simple to obtain. On entry to the instrumentation code the required registers are saved; these registers are restored upon exit from the instrumentation code.

To reduce the run-time penalty of obtaining temporary registers, a binary transformation tool can use live register information to avoid saving and restoring registers whose values are no longer needed by the original program. However, calculating live register information requires an accurate control flow graph, not possible in the presence of unknown indirect control transfers. Further, without inter-procedural register usage information, transformation applications must assume that all registers not mentioned in the procedure are live, that procedure calls use and define all registers, and that all registers are live on exit from the procedure.

Table 2 shows the cost of obtaining 2, 4, and 8 temporary registers before every load and store instruction in our benchmark programs with and without register usage information. For example, Table 2 shows that the execution time overhead of obtaining 4 registers with accurate register usage information is only 3%, compared to 157% without such information.

Finally, allocating reserved registers also can introduce significant run-time overhead. All uses of the reserved register in the original program must be removed. This requires either obtaining other free registers or mapping the reserved register to a fixed memory location. Because programs can manipulate the stack in unpredictable ways (such as allocating space using the C library call `alloca`), the stack can not be used to hold register values, further complicating reserved register allocation.

3 Building Adaptable Binaries

In this section, we define the minimum information required for efficient and robust support of

Benchmark		Without Register Information Two Registers	Without Register Information Four Registers	Without Register Information Eight Registers	With Register Information Two Registers	With Register Information Four Registers	With Register Information Eight Registers
`052.alvinn`	FP	44%	91%	199%	1%	1%	13%
`026.compress`	INT	43%	83%	311%	2%	3%	12%
`056.ear`	FP	63%	133%	289%	0%	0%	4%
`023.eqntott`	INT	69%	153%	337%	0%	1%	8%
`008.espresso`	INT	54%	281%	668%	2%	4%	14%
`001.gcc1.35`	INT	51%	91%	195%	3%	9%	19%
`022.li`	INT	68%	310%	892%	3%	5%	11%
`072.sc`	INT	45%	114%	289%	1%	3%	12%
Average		**55%**	**157%**	**398%**	**2%**	**3%**	**12%**

Table 2: Transformation overhead, relative to native execution time, of obtaining the specified number of scratch registers at each load and store instruction.

disassembly, edit, and register operations on binary programs. An adaptable binary is any executable program representation that contains this information. We also outline the implementation of our adaptable binary system, which uses this information to support binary transformation.

3.1 Adaptable Binary Information

The necessary information falls into three categories. *Control information* provides enough information to construct a control flow graph for each procedure, providing support for distinguishing code from data and identifying basic blocks. *Relocation information* supports editing operations by allowing references to instruction and data objects to be statically updated. Finally, *register usage information*, in conjunction with the control information, supports live register analysis, significantly improving the performance of register operations.

This information can not be derived reliably from current binary programs. Fortunately, the necessary information is readily available in most compilers and can be compactly recorded using conventional binary symbol tables. To conserve space, only information that is impossible to derive is stored in the binary; our adaptable binary system reconstructs complete control, relocation, and register usage information at transformation time.

3.2 Control Information

Three components of the control flow graph can not be derived reliably and are maintained in the adaptable binary:

- The beginning and ending address for each procedure.
- Entry addresses for each procedure.
- The possible targets of indirect control transfers.

Indirect control transfer targets are specified using *target groups*. Target groups are named sets of addresses; an address may belong to any number of target groups, and target groups need not be unique. Target groups are used for two reasons. First, because many indirect control transfers have the same set of possible target addresses (*e.g.* procedure returns), target groups provide considerable space savings. Second, target groups are used to support separate compilation. The targets for certain classes of indirect control transfers, for example exception handling, might reside in files not yet processed by the compiler. Because new addresses can be added to target groups at any time, they provide a convenient level of indirection in naming target addresses.

Given the above control information, our adaptable binary system locates basic blocks using the following algorithm:

1. Initialize work list to entry address of the program[1]. Repeat steps 2 and 3 until the work list is empty.

[1] Since the operating system requires the entry address to begin program execution, we can safely assume that it is specified in all binaries.

2. Remove an address `address` from the work list and create a corresponding basic block or split an existing block. Add instructions to the current basic block, beginning at `address`, until either a control flow instruction is encountered or there are no more instructions.

3. Given a control flow instruction, if any of the target addresses have not been processed as in Step 2, add the addresses to the work list.

Unlike conventional algorithms that linearly process the program to discover basic blocks [ASU86], the above algorithm guarantees that no data is inadvertently processed as code. Unreachable areas in the code segment are simply treated as data. Given the basic blocks, the next step is to build the control flow graph.

On architectures without *delayed control transfers*, building the control flow graph is a simple task [ASU86]. In the presence of branch delay slots, especially annulled delay slots, building the control flow graph is more difficult than for higher-level language programs, but still straightforward given the adaptable binary information [LB92].

3.3 Relocation Information

Edit operations, such as inserting instrumentation code, can change the address of instructions and data objects. Adaptable binaries must provide information to update these references at transformation time, eliminating the need for techniques that incur run-time overhead, such as dynamic relocation or out-of-line code insertion.

Traditional linkers combine one or more object files into a single executable, relying on relocation information to locate and update all program references. Each relocation entry consists of a pointer to the affected reference and the operations required to update it. Traditionally, the linker discards the relocation information following creation of the binary. Adaptable binaries retain this standard *inter-file* relocation information.

In theory, inter-file relocation is not sufficient to support transformations that change the relative order of instructions within a file. For example, consider the indirect control transfer depicted in Figure 3. Because conventional inter-file relocation information assumes that the relative placement of instructions within a file remains unchanged, support for updating the reference `CodeLabel` is provided, but not, typically, the constant offset `-16`. If instrumentation code is inserted between `CodeLabel` and `CodeLabel-16`, the constant offset must be adjusted.

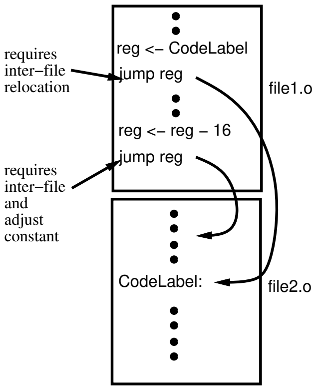

Figure 3: Two indirect control transfer instructions. The first `jump` requires only inter-file relocation information. Because the second `jump`'s target address is computed using an offset (`-16`) relative to a relocated label (`CodeLabel`), inter-file relocation is not sufficient.

As another example where inter-file relocation is not sufficient, consider Figure 4. If the size of the instrumentation code placed before `CodeLabel1` is different than the instrumentation code placed before `CodeLabel2`, there is no easy way to adjust the offset `-16`.

As a rule, adaptable binaries must contain sufficient relocation information necessary to update addresses if the relative position of instructions is changed. To date, we are not aware of any binaries that make use of computed target addresses as described above. Hence, rather than complicate the relocation information, we simply prohibit adaptable binaries from using these types of computed addresses.

3.4 Register Usage Information

As outlined in Section 2, register operations that allocate temporary and reserved registers can benefit from live register information. For each procedure, adaptable binaries store the following information:

- *livereg-gen* Registers whose value is used in the the procedure.

- *livereg-kill* Registers whose value is defined in the procedure.

- *livereg-live-on-exit* Registers whose value is live on exit to the procedure.

```
if ()
   reg = CodeLabel1
else
   reg = CodeLabel2
reg <– reg – 16
jump reg


CodeLabel1:

CodeLabel2:
```

Figure 4: Example instruction sequence which complicates binary transformation applications. If the size of the instrumentation code inserted before `CodeLabel1` and `CodeLabel2` is different, there is no easy way to adjust the computed address `reg - 16`.

In the presence of callee-saved registers, *livereg-gen* and *livereg-kill* can not be precisely derived. Consider the following function:

function f()
  / Save callee-saved register before using it. */*
  Memory ← `reg`
  ... function body ...
  / Restore callee-saved register. */*
  `reg` ← Memory

If there are assignments to memory in `f()`'s body, conventional program analysis would conclude that `f()` both uses and defines the value in `reg`. With respect to the call site, however, the value of `reg` is preserved across calls to `f()`, and is thus semantically neither used nor defined. When the compiler emits this register spill code, it can easily and precisely construct *livereg-gen* and *livereg-kill* to reflect this state. Because highly optimized programs can violate standard calling conventions, without precise *livereg-gen* and *livereg-kill* information, a transformation application using `reg` would be forced to unnecessarily spill the register before calling `f()`.

3.5 Our Adaptable Binary System

We have built a prototype adaptable binary system. The system supports both ecoff and a.out binary formats and can read OSF/1, SunOS, and Ultrix binaries. At present, it only includes disassembly modules for the MIPS and Sparc architectures.

We modified `gcc` to output adaptable binary (AB) information. Because the AB information was readily available in `gcc` data structures, adding this functionality to `gcc` required less than 800 lines of C code and less than a week's work.

3.6 Disk Space

Table 3 presents the disk space overhead of the control, relocation, and register usage information required by adaptable binaries. Our prototype stores adaptable binary information using the conventional binary symbol table and compresses this data using a variation of Liv-Zempel compression [LZ77]. We found that using a standard compression algorithm rather than semantic compression allowed simplified layouts of the adaptable binary information and yielded similar disk space savings. For comparison, we also measured the space overhead of the standard symbol table included in all unstripped executables and the symbol information required during debugging. As can be seen from Table 3, compressed adaptable binary information requires significantly less space than standard symbol table information.

4 Related Work

A number of systems have added information to binaries to facilitate transformation systems. Johnson modified the Stardent linker `ld` to retain inter-file relocation information [Joh90]. In addition, Stardent compilers were restricted from performing certain machine-dependent optimizations and placing data in the code segment. The Stardent system did not consider control or register information.

Like Johnson's system, `epoxie` retains inter-file relocation information [Wal91]. Rather than modify the linker, `epoxie` can use any linker that supports incremental linking. Incremental linking, (e.g. `ld -r`), retains relocation information, allowing already combined object files to be repeatedly linked.

The Mahler system performs transformations in the linker, providing transformation applications with inter-file relocation information [Wal92]. Like Stardent's `ld` and `epoxie`, no control information is included in the binary. Mahler performs several optimizations, including global register allocation. To accomplish this, *register actions* are included which tell the linker how to modify the binary if it decides to promote a variable from memory to a register. Register actions provide, however, only

Benchmark		Adaptable Binary Information	Standard Header	Standard Header + Debug Symbols
`052.alvinn`	FP	5%	95%	100%
`026.compress`	INT	8%	94%	107%
`056.ear`	FP	9%	82%	107%
`023.eqntott`	INT	9%	91%	131%
`008.espresso`	INT	9%	47%	151%
`001.gcc1.35`	INT	9%	92%	124%
`022.li`	INT	9%	93%	181%
`072.sc`	INT	11%	50%	81%
Average		**9%**	**81%**	**123%**

Table 3: Disk space overhead for adaptable binary information. For context, we also present the disk space overhead for the standard header included in all unstripped programs, as well as the symbol overhead when program's are compiled for debugging.

limited support for determining live register information.

In addition, systems have used compiler-dependent heuristics to approximate the control, relocation and register information present in adaptable binaries [Wal91, GH90, LB92, SW93]. Control and relocation information is synthesized by pattern matching for compiler-dependent instruction idioms. For example, to find jump tables, `MTool` searches for the instruction sequence it assumes will implement case statements [GH90]. Larus and Ball exploit register usage conventions when allocating registers, relying on programmer input to discover cases in which these assumptions are invalid [LB92].

Relying on compiler-dependent heuristics reduces the scope and effectiveness of a binary transformation application. For highly optimized code, finding a reliable heuristic might be difficult or impossible. Programs frequently use libraries written in different high-level languages or generated by different compilers. In these cases, different and potentially incompatible heuristics might be necessary. Because there are no constraints on the compiler, it is impossible to insure that the heuristics cover all possible sequences of generated code. For example, the Ultrix C compiler will violate normal MIPS register usage conventions when asked to do global register allocation[Digb]. Finally, a binary transformation tool that relies on compiler-dependent heuristics must be updated and tested with each new release of the compiler.

Adaptable binaries avoid these difficulties by including extra information in the executable representation of a program. Once annotated with this information, a binary program or library can support a large range of binary transformation applications, regardless of its origin.

On the other hand, support for adaptable binaries requires modifications to existing compilers. Even small changes to compilers, such as the ones described in this paper, may be politically difficult to get adopted. Until compilers export adaptable binary information, binary transformation systems must choose between added efficiency at the cost of compiler-dependent heuristics.

Finally, researchers have performed transformations on higher level program representations. For example, modifying assembly or object files addresses the problem of deriving relocation information. It does not address the need for control and register usage information.

Further, instrumenting a program at higher levels of abstraction poses the problem that many classes of program events are influenced by compilation decisions not fully resolved until the binary is created. For example, instrumenting a compiler's intermediate form can significantly affect the code generated. On many systems, the assembly representation contains high-level pseudo instructions, hiding machine instruction selection. On MIPS systems, the assembler performs machine instruction selection, delay slot filling, and instruction scheduling [Diga]. The relative order of procedures as well as final addresses are not established until the object files are processed by the linker.

5 Conclusion

We have described adaptable binaries, a technique for supporting robust and efficient binary transformation. First, we identified a set of operations that are fundamental to binary transformation. At least one of these operations is necessary for every binary transformation application that we surveyed. Second, we defined the minimum information required for efficient and robust support of these operations. We detailed the subset of this information that can not be derived from current binary programs and demonstrated that this subset of information can be added to executable files with negligible space overhead. Finally, we demonstrated quantitatively that augmenting binary programs with this information can dramatically reduce the run-time overhead of instrumented programs.

Adaptable binaries establish the necessary and sufficient information that any compiler must provide to support the basic binary transformation operations. Once annotated with this information, a binary program or library can support a large range of binary transformation applications, regardless of its origin. We hope that this work will influence compiler-writers and maintainers to make the output of adaptable binary information a standard compiler option.

Authors

Susan L. Graham is a professor of Computer Science at the University of California, Berkeley. Her research interests include programming language implementation, software development environments, and high-performance computing.

Steve Lucco is an assistant professor of Computer Science at Carnegie Mellon University. His research interests include software-enforced fault isolation, new organizations for operating systems, and high-performance computing.

Robert Wahbe is a Ph.D. candidate in Computer Science at the University of California, Berkeley. His current research focuses on mechanisms to allow databases and operating systems to safely incorporate third-party extensions written in traditional programming languages such as C and Fortran.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[BKKM87] A. Bergh, D. Margenheimer K. Keilman, and J. Miller. HP 3000 emulation on HP precision architecture computers. *Hewlett-Packard Journal*, December 1987.

[BKW90] Anita Borg, R.E. Kessler, and David W. Wall. Generation and analysis of very long address traces. In *International Symposium on Computer Architecture*, pages 270–279, May 1990.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing. In *Proceedings of the Conference on Principles of Programming Languages*, pages 59–70, 1992.

[Cen] CenterLine Incroporated. *TestCenter Manual.*

[CK93] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, University of Washington, 1993.

[Diga] Digital Equipment Corporation. *Ultrix v4.2* `as` *Manual Page.*

[Digb] Digital Equipment Corporation. *Ultrix v4.2* `cc` *Manual Page.*

[Digc] Digital Equipment Corporation. *Ultrix v4.2* `pixie` *Manual Page.*

[Ech92] Echo Logic, Inc. News Release, May 1992.

[GH90] Aaron Goldberg and John Hennessy. MTOOL: A method for detecting memory bottlenecks. Technical Report TN-17, Digital System Reseach Center, December 1990.

[HB89] C. Hunter and J. Banning. DOS at RISC. *Byte Magazine*, pages 361–368, November 1989.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 Usenix Winter Conference*, pages 125–136, 1992.

[Joh90] S. C. Johnson. Postloading for fun and profit. In *Proceedings of the Winter USENIX Conference*, pages 325–330, 1990.

[Kes90] Peter B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 78–84, White Plains, New York, June 1990. Appeared as SIGPLAN Notices 25(6).

[LB92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconsin-Madison, March 1992.

[LZ77] J. Liv and A. Zempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[MGA92] Margaret Martoonosi, Anoop Gupta, and Thomas Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[SW93] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 264–275, July 1986. Appeared as SIGPLAN NOTICES 21(7).

[Wal91] David W. Wall. Systems for late code modification. Technical Report TN-19, Digital Western Research Laboratory, June 1991.

[Wal92] David W. Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems*, 14(3), July 1992.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, pages 203–216, December 1993.

[WLG93] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12, 1993. Appeared as ACM Sigplan Notices 28 (6).