Adaptable Binary Programs

Robert Wahbe[†] Steven Lucco Susan L. Graham[†]

April 1994

CMU-CS-94-137

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported in part by the National Science Foundation (CDA-8722788) and the Advanced Research Projects Agency under grant MDA972-92-J-1028 and contracts DABT63-92-C-0026 and N00600-93-C-2481. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, ARPA, or the U.S. government.

†: Computer Science Division, University of California, Berkeley, CA 94720.

Keywords: compilation, late code modification, program instrumentation

Abstract

To accurately and comprehensively monitor a program's behavior, many performance measurement tools must 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 incur 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 complex, compiler-dependent code generation idioms. In this paper we present *adaptable binaries*, a technique for implementing robust, efficient, and compiler-independent binary transformations.

We evaluated a prototype implementation of adaptable binaries under the Ultrix 4.2 operating system and the MIPS processor architecture. Using the C SPEC92 benchmarks, we assessed adaptable binaries in three 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 significantly reduce this overhead. Finally, we measured the run-time transformation overhead of two user applications, `pixie` and `MemSpy`. For our benchmark programs, using adaptable binaries eliminates `pixie's` 110% average transformation overhead and reduces `MemSpy's` average overhead from 1296% to 33%.

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 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 [BL92]. To aid programmers in identifying memory hierarchy bottlenecks, `MTool` compares actual and estimated execution times for different regions of the 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]. By applying this transformation at the binary level, the fault isolation system eliminates the need for a trusted compiler. 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, SW92, Wal86, Wal92].

At present, binary transformation applications face a difficult trade-off. Previous approaches to implementing robust transformations incur significant disk space and run-time overhead. To improve efficiency, some current systems sacrifice robustness, relying on heuristic assumptions about the program's control flow and register usage and recognition of complex, compiler-dependent code generation idioms. This reliance on heuristic information limits the scope and effectiveness of a binary transformation application.

In this paper we present *adaptable binaries (AB)*, a technique for implementing robust, efficient, and compiler-independent binary transformations. Adaptable binaries support three classes of binary transformation operations. First, *control operations* allow transformation tools to distinguish code from data, identify basic blocks, and identify targets of indirect control transfer instructions. Second, transformation tools can use *edit operations* to insert, delete, and reorder machine instructions. Finally, *register operations* make registers available for use by inserted 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 three 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 all unstripped executable files is 81%. Debugging information increases 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. For example, one of the main sources of inefficiency in DEC's VAX-to-Alpha binary translation is the use of machine emulation in cases where control is transferred to an unknown target instruction. Adaptable binaries eliminate the need for such emulation.

Finally, we measured the run-time transformation overhead of two applications, `pixie` and `MemSpy` [Digc, MGA92]. `Pixie` is a simple but widely-used performance analysis tool that counts basic blocks. For our benchmark programs, relative to the original execution time of the program, we were able to eliminate `pixie's` 110% average transformation overhead. `MemSpy` is a sophisticated analysis tool for precisely identifying memory hierarchy bottlenecks. Taking advantage of the information available in adaptable binaries reduced its average runtime transformation overhead from 1296% to 33%.

The rest of the paper is organized as follows. Section 2 defines the operations supported by adaptable binaries and discusses why they cannot be implemented efficiently under existing systems. Section 3 details the information and program analysis required to build adaptable binaries and discusses our prototype implementation. Sections 4 quantitatively evaluates the space requirements and run-time performance of different transformation strategies. Section 5 surveys the related work in this area.

2 Binary Transformation Operations

In this section we detail the fundamental implementation issues that complicate the support of control, edit, and register operations on binary programs. We define each set of operations and outline existing strategies for their support. We demonstrate that, because existing binaries lack crucial control, relocation and register information, these strategies must rely on conservative assumptions about the program and must resort to emulation when static analysis fails. 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 support of binary transformation.

2.1 Control Operations

Control operations provide program control flow information to binary transformation tools. For example, `pixie` counts basic blocks to obtain profiling information. To identify basic block boundaries, `pixie` necessarily relies on heuristics, rendering it inaccurate in some cases. Similarly, data breakpoint systems can use data flow analysis to reduce the number of memory update instructions that require monitoring. To make such analysis effective, the breakpoint system requires an accurate control flow graph [WLG93]. Finally, site-specific optimization techniques, such as instruction scheduling based on profiling information [SW92], also require control flow information.

The fundamental step in implementing control operations is the resolution of *indirect control transfers*. 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 run-time. Common programming language abstractions such as function pointers, case statements, and continuations are typically implemented using indirect control transfer instructions. Without adaptable binaries, the presence of indirect control transfers creates two significant problems. First, the utility of control flow graphs is extremely restricted. Because there are no constraints on the targets of indirect control transfers, any instruction in the program must be assumed to be a possible control transfer target. Applications that use control flow graph analysis to reduce instrumentation overhead must rely on compiler-dependent heuristics to guess basic block boundaries, accepting reduced accuracy and portability in exchange for greater efficiency.

Second, without the control information present in adaptable binaries, binary transformation applications cannot reliably distinguish 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, 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 read-only *data segments*. In the presence of indirect control transfers, there is no robust method for distinguishing this data from code. If data, mistaken for code, is instrumented, the transformed program will be incorrect. 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.

One solution to the problem of distinguishing code from data, 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. All load and store addresses are unaltered; hence, the data in the original code segment is used. The disadvantage of this approach is that it doubles the size of the transformed program.

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 a number of applications [Digc, LB92, Wal91], is to *dynamically relocate* affected references. Consider a transformation application that only changes the location of instructions. Control transfer instructions whose targets are statically resolvable can be updated during transformation. Indirect control transfers must be dynamically relocated using a translation table, built at transformation time, that maps old addresses to new addresses. Like code duplication, this technique doubles the size of the binary program, because the translation table must be the same size as the code segment. Binary transformation systems can also apply dynamic relocation to loads and stores if the locations of data objects are changed.

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.

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 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. Unfortunately, this solution incurs significant run-time overhead. 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; as stated above, current systems cannot reliably build a control flow graph. Further, without interprocedural 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.

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 control, 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 a control flow graph for each procedure, supporting control operations such as the identification of basic block boundaries. *Relocation information* supports editing operations by allowing all references to instruction and data objects to be statically updated. Finally, *register usage information* supports live register analysis, significantly improving the performance of register operations.

This information can not be reliably derived from current binary programs. Fortunately, the necessary information is readily available in any compiler 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 uses program analysis to reconstruct complete control, relocation, and register usage information.

3.2 Control Information

The information needed for control operations can be synthesized by constructing a control flow graph for each procedure. Three components of the control flow graph can not be reliably derived 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.
2. Remove an address `address` from the work list and create a corresponding basic 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 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 and extend it with *intra-file* relocation information. In rare cases, inter-file relocation does not support changing the relative order of instructions within a file. 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 intra-file offset `-16`. If instrumentation code is inserted between `codeLabel` and `codeLabel-16`, the constant offset must be adjusted through intra-file relocation information.

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:

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

- *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.

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

function f()
    Memory ← `reg`   *;save callee-saved register before using it*
    ... function body ...
    `reg` ← Memory   *;restore callee-saved register before returning*

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. 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 (ABS). The system supports both ecoff and a.out binary formats and can read OSF/1, SunOS, and Ultrix binaries. Most of its code is independent of the target architecture and binary format. 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.

In contrast, the ABS required several months of implementation and is over 10000 lines of C code. Most of the complexity of this implementation is in synthesizing full control, relocation, and register usage information from the AB information and in providing machine-independent abstractions for binary transformation operations. This machine-independent layer supports the construction of portable binary transformation applications.

In addition, we are currently modifying the Orbit T compiler to output AB information. This compiler makes considerable use of indirect control transfer instructions in implementing continuations [KKR+86]. At present, the compiler outputs the majority of the necessary AB information.

4 Evaluation

To quantify the impact of adaptable binaries on binary transformation applications, we performed two types of experiments. First, we measured the disk space overhead incurred by adaptable binaries. Second, we measured the run-time *transformation overhead* of supporting two transformation operations: instruction insertion and obtaining temporary registers. 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. We compare the overhead of programs transformed using previously proposed techniques to the overhead of programs transformed using our AB system. In all cases, we report transformation overheads relative to the original program's execution time.

In addition to measuring these basic transformation operations in isolation, we analyzed the operations required by two binary transformation applications: `pixie` and `MemSpy`. We measured the transformation overhead of these operations as implemented by the original programs. We then re-implemented these operations to take advantage of AB information and measured their transformation overhead. Our experiments used the C SPEC92 benchmarks, and were performed on a DecStation 5000/240 with 32 Megabytes of memory.

4.1 Space Overhead

Table 1 presents the disk space overhead of the control, relocation, and register usage information required by adaptable binaries. Our ABS stores adaptable binary information using the conventional binary symbol table and compresses this data using a variation of Ziv-Lempel compression [ZL77]. 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.

4.2 Individual Operations

Control operations do not directly introduce run-time overhead although the lack of control operations can preclude optimization of instrumented code. For example, `QPT`'s use of the control flow graph reduces the number of basic block counters by a factor of two and the number of counter increments by up to a factor of four [BL92]. However, we can directly measure the overhead of strategies for supporting edit and register operations.

4.2.1 Edit Operations

In order to successfully insert instructions into a binary program, a binary transformation tool must correctly translate indirect control transfer instructions. We measured three strategies for doing this. First, the transformation tool can use dynamic relocation, prefacing each indirect control transfer site with a table lookup that translates the old jump target address to the new jump target address. Second, using out-of-line insertion, the transformation tool can avoid this problem by never moving existing instructions. Third, the transformation tool can use the relocation information in adaptable binaries to safely update references to moved instructions. We designate these alternatives DYN-RELOC, OUT-OF-LINE, and AB respectively.

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 every indirect control transfer in the original program. Table 2 shows that the average performance overhead of this approach was 34.3% on our example programs.

Table 2 also gives the transformation overhead for OUT-OF-LINE for inserting instrumentation code before every load and store instruction. On average, the OUT-OF-LINE incurs 112.5% execution time overhead. The AB approach incurs no execution time overhead for supporting instruction insertion, since adaptable binaries contain the necessary information to statically update references to moved instructions.

```
Memory ← temp-reg
     Save temporary register temp-reg.
temp-reg ← jump-target-address - code-start-address
     Calculate offset of current jump target address.
temp-reg ← temp-reg + table base address
     Add jump target address offset to base of dynamic relocation table.
temp-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.

4.2.2 Register Operations

We measured the cost of obtaining 2, 4, and 8 temporary registers before every load and store instruction in our benchmark programs. Because they do not have accurate control flow or register usage information, transformation tools not using adaptable binaries must save and restore the required registers. With AB information, a transformation tool can accurately compute live register information and save only the live registers. For example, Table 2 shows that the execution time overhead of obtaining 4 registers with AB information is only 30.6%, compared to 157.3% without AB information.

4.3 Applications

The measurements of individual operations presented above suggest that adaptable binaries can increase the efficiency of transformed binary programs. In this section, we present case studies of two binary transformation applications: instruction profiling and memory system simulation. For instruction profiling, we used the commercial program `pixie`. For memory system simulation, we used the research system `MemSpy`.

4.3.1 `Pixie`

`Pixie` transforms a binary program to count basic blocks. It then uses the count of basic blocks to compute instruction frequencies, wasted cycles due to floating point pipeline interlocks, and other instruction profiling information. The transformed program simply counts basic blocks and outputs the counts to a file for post-processing.

We measured the transformation overhead for basic-block counting of the original `pixie` and a version that takes advantage of AB information. The AB version is both faster and more accurate than the original. It is more accurate because it can identify all possible targets of indirect control transfers and therefore all basic block boundaries. `Pixie` must rely on heuristics to guess basic block boundaries.

In addition to using a form of dynamic relocation to support editing operations, the original `pixie` implementation obtains two scratch registers and one reserved register. Table 4 gives the transformation overhead for the original and AB versions.

For our benchmark programs, `pixie` approximately doubles the end-to-end running time of the program. Of this overhead, 110.6% is due to the Pixie's transformation overhead, and 40.4% to maintaining counters for each basic block. Thus, adaptable binaries reduce `pixie's` end-to-end overhead by approximately a factor of four. For applications that use control flow analysis to optimize the placement of counters, such as `QPT`, adaptable binaries will provide even larger relative savings.

4.3.2 MemSpy

MemSpy measures memory system performance. It inserts instrumentation code before every load and store instruction in the original program, as well as all procedure entry and exit points. Currently, MemSpy uses the Tango simulation system to modify the assembly language version of a program. In doing so, it avoids the instruction insertion overhead incurred by pixie. However, Section 5 describes some important disadvantages of this choice.

Because the simulation code is sufficiently complex, the MemSpy designers chose to insert call instructions before each load and store, rather than to duplicate the simulation code inline. In addition, the simulation code is written in C, and thus assumes that all caller-save registers are available for use.

Since MemSpy does not currently have accurate register liveness information, it must save and restore all the caller-save registers. Using the register information available in adaptable binaries, we instrumented our benchmark programs to save only those caller-saved registers that were live at each call to the MemSpy simulation routines. Table 4 shows that this use of AB information reduces the transformation overhead due to register operations of MemSpy by more than a factor of 39. MemSpy's authors report that register operations account for approximately 60% of MemSpy's total simulation overhead [MGA92], and hence the use of adaptable binaries can eliminate the majority of MemSpy's end-to-end overhead.

5 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. In contrast, adaptable binaries place no constraints on compilers. Further, the Stardent system did not consider control, intra-file relocation, 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 or intra-file relocation 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 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, SW92]. 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.

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 significant 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. As mentioned above, the Titan system linker performs inter-procedural optimizations [Wal92].

6 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.

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.*

[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).

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986.

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

[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.

[SW92] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. Technical Report 92/6, Digital Western Research Laboratory, December 1992.

[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*, 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*, 1993.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A Figures

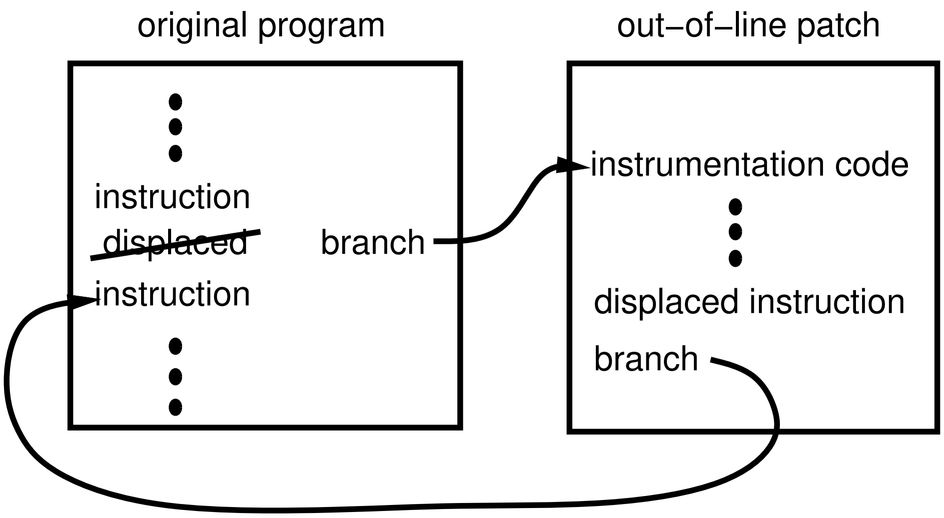

Figure 2: Simple example of out-of-line insertion of instrumentation code.

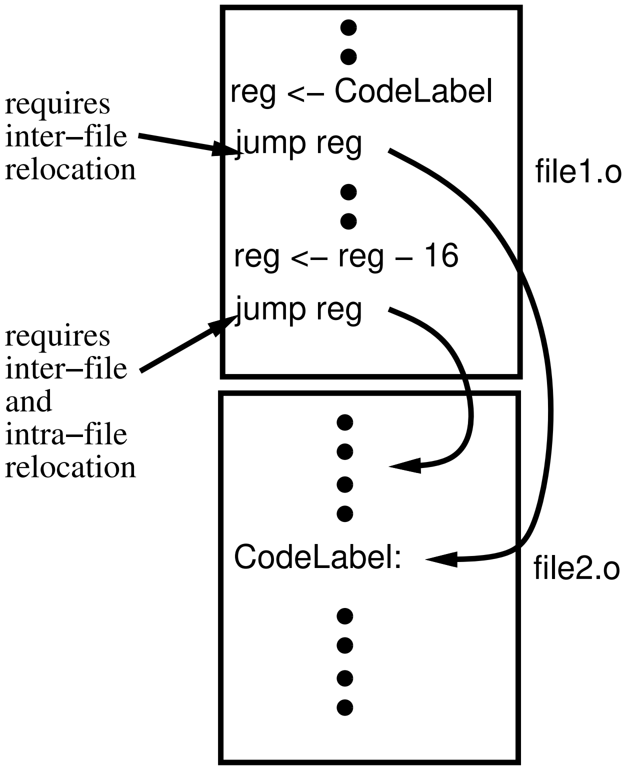

Figure 3: Two indirect control transfer instructions. The first `jump` requires only inter-file relocation information. Because the second `jump`'s target address is computed using an intra-file offset (`-16`) relative to a relocated label (`codeLabel`), both inter-file and intra-file relocation information is required.

B Performance Data

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 1: 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.

Benchmark		DYN-RELOC	OUT-OF-LINE
`052.alvinn`	FP	55.8%	112.2%
`026.compress`	INT	18.3%	104.8%
`056.ear`	FP	28.4%	98.8%
`023.eqntott`	INT	20.8%	86.1%
`008.espresso`	INT	10.7%	102.2%
`001.gcc1.35`	INT	58.0%	154.1%
`022.li`	INT	48.5%	170.1%
`072.sc`	INT	34.2%	71.3%
Average		**34.3%**	**112.5%**

Table 2: 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.

Benchmark		Current Systems			Adaptable Binaries		
		Two Registers	Four Registers	Eight Registers	Two Registers	Four Registers	Eight Registers
`052.alvinn`	FP	43.5%	91.3%	199.4%	32.4%	32.8%	33.1%
`026.compress`	INT	43.2%	83.1%	311.0%	23.1%	27.1%	30.0%
`056.ear`	FP	62.8%	133.4%	288.7%	26.6%	28.0%	28.3%
`023.eqntott`	INT	69.1%	152.8%	337.9%	13.1%	43.3%	47.8%
`008.espresso`	INT	54.6%	281.7%	668.8%	18.1%	22.3%	23.2%
`001.gcc1.35`	INT	51.1%	91.2%	195.8%	22.5%	27.9%	28.3%
`022.li`	INT	68.7%	310.6%	892.4%	37.2%	41.2%	45.1%
`072.sc`	INT	45.3%	114.7%	289.9%	19.3%	22.4%	22.6%
Average		**54.8%**	**157.4%**	**398.0%**	**24.0%**	**30.6%**	**32.3%**

Table 3: Transformation overhead, relative to native execution time, of obtaining the specified number of scratch registers at each load and store instruction.

Benchmark		Current Systems		Adaptable Binaries	
		`pixie`	`MemSpy`	`pixie`	`MemSpy`
`052.alvinn`	FP	42.4%	618.8%	42.6%	33.5%
`026.compress`	INT	69.3%	983.3%	45.0%	31.1%
`056.ear`	FP	99.8%	901.7%	2.0%	28.7%
`023.eqntott`	INT	223.7%	1182.6%	48.2%	49.6%
`008.espresso`	INT	68.4%	2178.0%	56.0%	23.0%
`001.gcc1.35`	INT	111.1%	619.7%	71.0%	29.0%
`022.li`	INT	122.6%	3010.2%	39.1%	45.3%
`072.sc`	INT	147.1%	876.2%	19.2%	24.0%
Average		**110.6%**	**1296.3%**	**40.4%**	**33.0%**

Table 4: Transformation overhead, relative to native execution time, for the applications `pixie` and `MemSpy`.