JUNE 1991

WRL
Technical Note TN-19

Systems for Late Code Modification

David W. Wall

digital

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Systems for Late Code Modification

WORKING PAPER

David W. Wall

June, 1991

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Abstract

Modifying code after the compiler has generated it can be useful for both optimization and instrumentation. This paper compares the code modification systems of Mahler and pixie, and describes two new systems we have built that are hybrids of the two.

This paper covers material presented at the
CODE ’91 International Workshop on Code Generation,
Schloss Dagstuhl, Germany, May 20-24, 1991.

1. Introduction

Late code modification is the process of modifying the output of a compiler after the compiler has generated it. The reasons one might want to do this fall into two categories, optimization and instrumentation.

Some forms of optimization must be performed on assembly-level or machine-level code. The oldest is peephole optimization [11], which acts to tidy up code that a compiler has generated; it has since been generalized to include transformations on more machine-independent code [2,3]. Reordering of code to avoid pipeline stalls [4,7,18] is most often done after the code is generated because the pipeline stalls are easier to see.

Other forms of optimization depend on having the entire program at hand all at once. In an environment with separately-compiled modules, this may mean we must apply the optimization to machine-level code. Global reorganization of code to reduce instruction cache misses [10] is one example. Intermodule allocation of registers to variables is another; the Mahler system [16] chose the register variables during linking and modified the object modules being linked to reflect this choice. Register allocation is a fairly high-level optimization, however, and other approaches have been taken, such as monolithic compilation of source modules or intermediate-language modules [15] or compilation with reference to program summary databases [14].

Optimization removes unnecessary operations; instrumentation adds them. A common form of machine-level instrumentation is basic block counting. We transform a program into an equivalent program that also counts each basic block as it is executed. Running the instrumented program gives us an execution count for each basic block in the program. We can combine these counts with static information from the uninstrumented program to get profile information either at the source level, such as procedure invocation counts, or at the instruction level, such as load and store counts [9,12].

Some events that we might want to count require inter-block state. Counting pipeline or coprocessor stalls, for example, can be done with basic-block counting only if any stall is guaranteed to finish before we leave the basic block. Counting branches taken or fallen through, or the distribution of destinations of indirect jumps, are other examples. Instrumentation to count these events is no harder than counting basic blocks, but requires different instrumentation [17,18].

Still other kinds of instrumentation are not hard to do by code modification. Address tracing for purposes of cache modeling [1,12] can be done by instrumenting the places where loads and stores occur and also the places where basic blocks begin. A 1988 study [19] compared software register allocation with hardware register windows, by instrumenting the code to keep track of procedure call depth and counting, for example, the times when a machine with register windows would overflow its buffer of windows.

Naturally, certain kinds of transformation are best done before we reach machine-level code. Transformations like macro expansion are usually defined as source-to-source transformations, and are nearly always implemented as such. Global movement of operations out of loops could in principle be done at the machine level, but it makes more sense to do it earlier, when more of the semantics of the program are easily available. Inline procedure expansion is often most useful if it can be followed by normal global optimizations, because the latter act to specialize the body of such an expansion, so inline expansion might not be done at the machine-level.

Nevertheless, it is clear that a wide variety of transformations for purposes of optimization and instrumentation can be done on machine-level code. This paper compares two existing systems for late code modification, finds advantages and disadvantages to both, and then describes two new systems we have built that represent compromises between these two, combining the advantages (and a few of the disadvantages) of both.

2. Overview of late code modification.

There are two different questions to address. How do we decide what modifications to make? How do we make those modifications and still get a working program?

The former depends on the transformation. In basic block counting, we simply insert a load, add, and store at the beginning of each block, to increment the counter associated with that block. In interprocedure register allocation, we decide which variables to keep in registers -- the hard part -- and then we delete loads and stores of those variables and modify instructions that use the values loaded and stored. In any case this paper will not discuss how to decide what changes to make, except in passing.

There are two main problems associated with making the modifications correctly. The first is that adding and deleting instructions causes the addresses of things to change. We must be able to correct for these changes somehow, or references to these addresses will not work right. The second is that the modifications may themselves need resources that must somehow be acquired. Most commonly these resources are registers: for example, in block counting we need at least one register in which to do the increment, and another to hold the address of the count vector.

A code modification system that knows enough about the program can address both these problems without introducing much or any overhead in the transformed program. This makes the system a suitable medium for optimization as well as for instrumentation. If we are forced to introduce overhead, to deal with the problems dynamically rather than statically, optimization is unlikely to be feasible, and a large overhead will reduce the usefulness of the system even for instrumentation.

Next we will look at two systems for late code modification, and compare their approaches.

3. Mahler

The Mahler system [16,17,18] is the back-end code generator and linker for the Titan [8,13], an experimental workstation built at DECWRL. The Mahler system does code modification in the linker. A wide variety of transformations are possible:

intermodule register allocation, instruction pipeline scheduling, source-level instrumentation compatible with gprof [6], and instruction-level instrumentation like basic block counting and address tracing [1]. The linker decides in the usual manner which modules to link, including modules extracted from libraries. Each is passed to the module rewriter, which modifies it according to the transformations requested. The transformed modules are then passed back to the linker proper, which links them just as if they had originally come from object files.

Correcting for changed addresses is easy in this context. An object module contains a loader symbol table and relocation dictionary, which mark the places where unresolved addresses are used. An unresolved address may be one that depends on an imported symbol whose value is not yet known, or may be one that is known relative to the current module but that will change when the module is linked with other modules. In either case the linker is responsible for resolving it, and the relocation dictionary tells the linker what kind of address it is. This same information lets us correct the value, if only by leaving the relocation entry in place so that the linker will correct it.

Other addresses are not marked for relocation, because they are position-relative addresses and will not change when the module is linked with others. These are all manifest in the instruction format itself, as in the case of a pc-relative branch instruction. If instructions are inserted between a branch and its destination, we increase the magnitude of the displacement in the instruction word.*

If the module rewriter is applying a transformation that requires some new registers, such as block counting, they are easy to obtain, because global register allocation is also done in the linker. Thus we can choose to allocate fewer registers than the maximum, reserving a few for the use of the instrumentation code. This leads to some overhead, because the variables that would have used those registers must now live in memory and must be loaded and stored as needed. However, the Titan has 64 registers, and losing the last few never made much difference to performance.

Mahler code modification is made still easier by the combination of two circumstances. First, the presence of the loader symbol table means that the compiler can pass hints through it. Second, the Mahler compiler is the back end of all the high-level language compilers, and is also the only assembler available for the Titan, which means that *any* code the module rewriter sees is guaranteed to have been produced by the Mahler compiler. For example, the Titan has no variable-shift instruction; this operation is implemented as an indexed jump into a series of 32 constant-shift instructions. The code modifier must be careful not to damage this idiom, and would wreak havoc if for example it blithely changed the order of these shifts. The Mahler compiler flags this idiom with a symbol table entry, which tells the module rewriter what it is.

* Although this might seem to handle all the cases, in theory there is another that would be troublesome. If we did an unrelocated computation of some kind, added the result to the pc, and jumped to that address, it would be very difficult to correct this address statically. Such a computation is one possible implementation of a case-statement, but the more common one is to use the computation to select the destination from a table of complete addresses, each marked for relocation.

Mahler's approach of code modification in the linker arose from the desire to do intermodule register allocation without giving up separate compilation. With that machinery in place, it was natural to do instrumentation there as well. This unfortunately means that to request instrumentation a user must relink, and so must have access to the object modules and libraries that the program is built from. On the other hand, this approach means the user need not recompile, and means we need not maintain instrumented versions of the libraries, as has been the usual practice, for example, in systems that support gprof [6]. The need to relink is inconvenient, however, and it is interesting that another approach is possible.

4. Pixie

Pixie [9,12], developed by Earl Killian of MIPS, does block counting and address tracing by modifying a fully-linked executable. The executable file may even have had its symbol table removed, though the block counts are normally analyzed by tools that need to see the symbol table. This approach is much more convenient for the user, who does not have to be able to rebuild the executable. It is less convenient for pixie, however, which must work harder to preserve program correctness, and introduces run-time overhead in doing so.

As with Mahler, some address correction is easy. Pc-relative branches and jumps are easy to recognize, and pixie can change their displacements just as Mahler can. Indirect jumps pose more of a problem, because their destinations are computed at other points in the program, and we no longer have the relocation dictionary to let us recognize those points. Pixie constructs a big translation table that maps code addresses in the original program into code addresses in the modified program. This table is built into the modified executable, and every indirect jump is preceded by code that translates it at run time by lookup into this table. Even when pixie knows it is computing a code address, it computes the old version of that address, so that the runtime translation lookup will always work correctly regardless of where the address came from. This leads to a rather odd expansion of a jump-and-link instruction, which expands to an ordinary jump together with a pair of instructions that load what would have been the return address in the unmodified program.

Fortunately for pixie, data addresses do not change even though the code segment gets larger. By convention, data segments on MIPS systems occupy a place in the address space that is far removed from the code segment; there is ample room for a code segment to grow manyfold before it affects data addresses. Without this convention, data addresses might change (as in fact they do under Mahler on the Titan); pixie would have to do loads and stores with an extra level of indirection, just as it does indirect jumps.

The instrumentation added by pixie needs three registers for its own use, which it must steal from the registers already used by the program. Pixie accomplishes this by maintaining three memory locations that have the ''user'' values of these three registers, and replacing uses of these registers by uses of the memory locations. Since the MIPS processor has a load-store architecture, this adds some overhead, but it is typically small compared to the overhead of the instrumentation itself. These three registers can then be used for pixie's purposes, and also as temporaries in the expansions of user instructions that use them. To pick the registers to steal, pixie looks for the registers that have the fewest number of static references.

Pixie uses no hints from the compiler, and in fact does not even use knowledge about the patterns of code the compiler generates. This means it can tolerate assembly code and even code from ‘‘foreign’’ compilers, neither of which is sure to adhere to the MIPS compiler conventions.

5. What more do we need?

Pixie is convenient to use, but incurs a fair bit of runtime overhead. By far the majority of indirect jumps are subroutine returns, and most of these are from ordinary direct calls. In these cases the overhead of going through the address table is logically unnecessary. If all we want to do is instrumentation, the overhead is probably acceptable, though there are certainly times when we might want to do lightweight or time-critical instrumentation. In any case, the overhead makes it an unsuitable vehicle for optimization.

Mahler's use of the loader information lets it transform a program with much smaller runtime overhead. It is built into the Titan linker, however, and depends to an unknown extent on the integration of the back-end compiler and the linker. Moreover, it is currently targeted only at the Titan family and is unlikely to be retargeted to the MIPS.

It seems that a compromise between Mahler and pixie might be in order. Much of pixie's overhead is not logically necessary: for instance, if a procedure is certain to be called only directly, those calls and returns can safely be translated as calls and returns, without going through the address table. Moreover, all of the MIPS compilers have the same backend, so we can be almost as sure of understanding coding conventions as we are with Mahler on the Titan. A variant of pixie that paid attention to compiler conventions and other information might be able to reduce the overhead significantly, perhaps enough to let us use it for optimization as well as instrumentation. The rest of this paper describes prototypes of two such compromises that we have built at DECWRL.

6. Nixie

The first compromise we built was a system called *nixie.* Nixie works very much like pixie, but makes certain assumptions about the code it modifies. The framework of this approach is threefold. First, the MIPS architecture has a strong bias in favor of using r31 as the return address register. Second, the direct jump-and-link (JAL) and jump-register (JR) instructions used for normal procedure call and return in the original code will work just fine in the modified code, provided that we change the literal destination field from an old address to a new one. Third, all of the MIPS compilers use the same back end code generator, whose coding conventions are therefore uniform over all compiled code.

Nixie therefore assumes that any JR via r31 is in fact a return, and (sensibly) that any JAL is a call, and leaves them both unchanged. A jump-and-link-register (JALR) instruction is assumed to be an indirect call to a procedure whose address is in the instruction's operand register; this instruction is preceded by address translation code to look up the procedure address and convert it from the old space to the new.

We would not be far wrong if we further assumed that any other JR is the indexed jump from a case-statement. These we can handle by noting that the MIPS code generator produces only a few distinct code templates for these indexed jumps. By examining the instructions before the JR, we can confirm that this JR is indeed the indexed jump from a case-statement, and can also determine the address and size of the associated jump table in the read-only data segment. This allows us to translate the entries in this table statically, so that we need not insert translation code.

The only hole in this argument is that a program can contain code that was originally written in assembly language. Such code was *not* produced by the code generator, and might not follow its conventions. Users tend not to write such code, but it is not uncommon in the standard libraries. Fortunately, most assembly routines in the libraries follow the conventions well enough for our purposes: they return through r31, and they seldom have case-statements at all.

In the standard libraries, nixie knows of two dozen assembly routines that have unusual jumps. It finds these in an executable by looking for the file name in the loader symbol table, and then pattern-matching the code at that address to confirm that the routine is really the one from the library. If the symbol table is absent, nixie searches the entire executable for the patterns. When one of these nonstandard routines is found, nixie marks the violations so that it will know what to make of them later.

Any jump that we have been unable to explain with this whole analysis triggers an error message. In programs without procedure variables, we are normally able to translate all jumps cleanly, without using the runtime address table, and if so we do not include the address table in the transformed executable. In accomplishing this, we have conveniently also assigned a meaning to all jumps, distinguishing between procedure calls, procedure returns, and case-statement indexed jumps. This lets us consider doing global analysis using the control structure of the program. The reduction, often to zero, of the runtime overhead of doing program transformation, together with the ability to do global analysis, means that we should be able to use this technique to do low-level optimization as well as instrumentation.

On the other hand, certain assumptions are necessary. We assumed that a JAL or JALR is used only for procedure call; if this is false our understanding of the global structure may suffer. More seriously, we assumed that a JR via r31 was a return; if this is false we will not translate the destination of the jump correctly, and the transformed program won't even run correctly.

Moreover, we can't always remove the runtime address table. Mahler never needed it, and it would be nice to dispense with it in general. This is the motivation for the second compromise system.

7. Epoxie

Mahler can do all of its address translation statically, because it operates on object files, which still have loader symbol tables and relocation dictionaries. We wanted a pixie-like tool that would work the same way, but didn't want to change the standard linker. One option would be to make the tool transform an individual object file, and then apply it to each of the object files in turn, but this would make it inconvenient to include library files. Our solution was to assume that we can begin by completely linking the program but retaining all of the loader information. In practice this is easy to do: most standard linkers have an option for *incremental linking,* which lets us link together several object files into what is essentially one big object file that can then be linked with others to form a complete executable. In our case we will link together everything there is, but we will pretend there might be more so that the linker will keep the relocation information we want.

This done, we can modify the result in the same manner as Mahler. Where the code or data contains an address, either the use is marked for relocation or else the use is self-evident, as with a pc-relative branch. Like Mahler, this system translates not only code addresses that appear in the code, but also code addresses that appear in the data, the relocation dictionaries, and the symbol table. Unlike Mahler, this system need not translate data addresses, because the addresses of the data segments do not change.

The resulting system is nearly identical in structure to nixie, and is called *epoxie,* because all the glue holding the executable together is still visible. With epoxie, we never need runtime address translation. For many applications, we need not do as thorough an analysis of the program's jumps: for basic-block counting, for example, we do not care whether a JR is a case-statement or a subroutine return, and epoxie does not need to know this for a safe translation.

8. Pitfalls

There are several tricky bits worth explaining.

There is occasionally a reason to have code in the data segment. For example, an instruction might be constructed on the fly for emulation purposes. None of the code modification systems discussed here understand this possibility, and therefore will not instrument such code. Fortunately, such code arises seldom. Not counting it as a basic block will probably not skew results much. On the other hand, branching to code in the data segment and then back to the code segment could easily confuse us if we are doing a transformation that requires understanding the control flow.

There is also occasionally reason to have data in the code segment. The MIPS Fortran compiler puts constant arguments of subroutines in the text segment.* The documentation for pixie suggests that an early version of the MIPS Fortran compiler

---

* Presumably this is so it can pass these arguments by reference, secure in the knowledge that a protection violation will occur if the subroutine tries to modify them. I don't know why it doesn't put them in the read-only data segment instead.

used code-segment data as part of the implementation of FORMAT statements. There are two problems with data in the code segment. First, we access this data via an address into the code segment, and code modification will probably change the address. Second, if we mistake the data for code, we may instrument or modify it, causing the computations that depend on it to get the wrong values.

Pixie and nixie deal with both problems by including a copy of the original code at the beginning of the modified code segment. Since loads and stores do not have their addresses translated as jumps do, following the untranslated address will lead to the old, unmodified code. This means we will both look in the right place and get an unaltered value.

Because epoxie corrects all addresses statically according to the relocation dictionary, the address will lead us to the right place in the modified code. It won't help, then, to include a copy of the original code. Instead, we must distinguish between instructions and data in the code segment, and be sure not to alter the latter. Fortunately, it turns out that the MIPS loader symbol table contains information that makes this possible. This information also lets nixie determine (if the symbol table is present) whether including the old code is necessary.

Nixie uses the runtime address translation table only if there are indirect procedure calls in the program. Unfortunately, our standard C library has one routine, called *fwalk,* that takes a procedure address as an argument and makes indirect calls to that procedure. *Fwalk* is called in all programs, as part of the normal exit sequence. Rather than surrender and use the translation table all the time, nixie recognizes the *fwalk* routine, and also recognizes direct calls to it. It inspects the instructions around these calls to find the instructions that compute the procedure address passed. If these instructions compute a literal address, as is usually the case, nixie simply modifies them to compute the corrected address. This is admittedly unsafe*, because *fwalk* might itself be called indirectly, in which case we can't recognize the call and therefore won't do the translation of its argument. Pixie and epoxie do not have this problem, because all code addresses are translated at the same time: at run time for pixie and at modification time for epoxie.

The MIPS and Titan architectures both have delayed branches, so when a branch or jump is taken, the instruction after the branch (sometimes called the ''branch slot'') is executed before control is transferred. Code modification may replace the branch slot instruction by a series of several instructions. In that case we cannot just leave these instructions in place, because only the first will be executed if we take the branch. Fortunately, a branch slot is often filled with an instruction from before the branch, in which case the expansion of this instruction can be safely moved back before the branch.

* The technical term for this is ''sleazy.''

It may be incorrect, however, simply to move the slot back before the branch, if the slot changes a register that the branch uses.* In this case there are two possible approaches. One is to duplicate the expansion, so that we replace

```
conditional-branch to L
slot instruction
```

by

```
      reverse-conditional-branch L1
      nop
      slot expansion
      unconditional-branch to L
      nop
L1:   slot expansion
```

(If we are careful we can do without the two nops.) This is roughly what pixie does. Another choice is to expand to

```
tempreg := evaluate condition
slot expansion
if tempreg branch to L
nop
```

Epoxie and nixie do this, for a bad reason: I didn't understand a tricky property of the relocation dictionaries, which I thought made it impossible to correctly duplicate the code if it was marked for a particular kind of relocation.† Pixie's approach is longer but simpler, and works better in the case where there is a branch from somewhere else to a branch slot.‡ Pixie can simply consider everything up to L1 as part of the expansion of the branch, and everything at L1 and beyond as the expansion of the slot. In contrast, epoxie must do an ugly thing: it changes

---

* This can happen, for example, if the assembler filled the branch slot by moving an instruction from the destination block or the fall-through block. It does this only if it is safe; i.e. if the register changed by this instruction is set before it is used in the alternative block.

† Specifically, the MIPS loader format has a pair of related operations, R_REFHI and R_REFLO, which it uses to relocate a pair of instructions that compute a 32-bit address by combining two 16-bit parts. These relocation entries must be consecutive in the dictionary, because we must inspect both instructions to know how to relocate the first. The problem arises if the slot instruction is relocated R_REFLO, where the associated R_REFHI relocation appears before the branch. If we duplicate the expansion of the slot, we will have two R_REFLO entries and only one R_REFHI entry, which I thought at first was illegal. In practice, though, a R_REFHI can be followed by any number of R_REFLOs, as long as they all make the same assumptions.

‡ This can happen explicitly in assembly code, of course, but also occurs as the result of pipeline scheduling, if a branch slot is filled from the fall-through block, but the fall-through block is itself the destination of another branch.

```
        conditional-branch to L
LL:     slot instruction
```

to

```
        tempreg := evaluate condition
        branch to L2
        nop
LL:     tempreg := false
L2:     slot expansion
        if tempreg branch to L
        nop
```

so that branches to LL will follow the right path. This works, but it is distinctly inelegant.

System calls present a few problems to these systems. System calls are allowed to destroy the first 16 registers, which makes them unsuitable choices as bookkeeping registers. A sigreturn trap restores the values of the other 16 from a data structure, so pixie, nixie, and epoxie precede this trap with code that copies the saved values of the bookkeeping registers into its own memory representation, and stores the bookkeeping values of these registers into the sigreturn data structure. In addition, either a sigreturn or a sigvec call involves a text address that will eventually be transferred to by the operating system kernel. The code of the kernel is not modified by our tools, so pixie and nixie precede these traps with code to translate the address. Epoxie need not, because the original address computation, wherever it is, has relocation entries that cause epoxie to translate it. If we want to do anything just before termination, like write a file of basic block counts, we can insert code to do so before each _exit system trap: pixie, nixie, and epoxie all work this way.

Probably the biggest pitfall is the use of pattern-matching to recognize certain library routines. This is relatively fail-soft, because a routine that is listed as present in the symbol table but that does not match the pattern triggers a warning, and a jump that cannot be classified triggers an error. Nonetheless, the patterns are very specific, and a new version of the library routine will probably fail to match. The right solution is to raise the level of the assembly language slightly, so that assembly code adheres to the same conventions as compiled code. One simple way to do this is by requiring the assembly programmer to explain the violations of the conventions, and encoding these explanations in the symbol table. If we do not wish to change the assembler, a less attractive solution is to develop a tool that looks for violations in the libraries themselves, and asks the nixie/epoxie maintainer to categorize them. These explanations would in turn generate the patterns used by nixie and epoxie.

	mahler	*epoxie*	*nixie*	*pixie*
ccom	1.4	2.1	2.1	3.0+
doduc	1.3	1.6	1.6+*	2.0+*
eco	1.5	2.0	2.0	2.9+
egrep	1.6	2.1	2.1	2.9+
eqntott	1.5	2.1	2.2+	3.0+
espresso	1.5	2.1	2.1+	2.7+
fpppp	1.2	1.6	1.6+*	2.0+*
gcc1	1.6	2.4	2.4+	2.7+
grr	1.5	2.0	2.0	2.7+
li	1.6	2.1	2.1+	3.1+
linpack	1.4	1.9	1.9	2.7+
livermore	1.6	2.0	2.0	2.7+
matrix300	1.6	2.1	2.2+*	3.0+*
met	1.5	2.0	2.0	2.8+
nasa7	1.4	1.9	1.9+*	2.6+*
sed	1.5	2.0	2.0	2.7+
spice	1.3	1.7	1.8+*	2.1+*
stanford	1.4	2.0	2.0	2.8+
tomcatv	1.5	2.1	2.1+	2.9+
whetstones	1.5	2.0	2.0	2.8+
yacc	1.5	2.1	2.1	2.8+

+ means runtime address table required; add 1.0

* means original code required; add 1.0

Figure 1. Ratio of code size to original.

9. A few numbers

We have a prototype version of nixie and epoxie. To compare its code modification to that of Mahler and pixie, we implemented basic block counting. Figure 1 shows the expansion in code size for a variety of programs.

The ratio of code size reflects only the difference in size between the executable instructions in the original and instrumented versions. In addition, pixie and occasionally nixie need the address translation table that converts old addresses to new, which is as long as the original code segment. Moreover, pixie and nixie sometimes need the original code to be included before the instrumented code, because data items exist in the code segment.*

* In practice, pixie and nixie align these pieces on particular boundaries for convenience, resulting in even larger instrumented versions; I believe this is not logically necessary, and so the empty space between pieces is not counted.

	mahler	*epoxie*	*nixie*	*pixie*
ccom	1.6	2.1	2.1	3.0
doduc	-	1.2	1.2	1.3
eco	1.7	1.9	1.9	2.5
egrep	1.6	2.0	2.0	2.4
eqntott	1.5	2.1	2.2	2.7
espresso	1.6	1.9	1.9	2.2
fpppp	1.0	1.1	1.1	1.2
gcc1	1.7	2.2	2.2	2.7
grr	1.6	1.4	1.4	1.7
li	1.7	2.0	2.0	2.7
linpack	1.1	1.1	1.1	1.1
livermore	1.1	1.3	1.3	1.4
matrix300	-	1.1	1.1	1.1
met	1.5	1.7	1.7	2.3
nasa7	1.0	1.1	1.1	1.1
sed	1.6	=	=	=
spice	1.2	1.3	1.3	1.4
stanford	1.5	1.7	1.7	2.1
tomcatv	1.1	1.1	1.1	1.1
whetstones	1.2	1.3	1.3	1.6
yacc	1.6	1.9	1.9	2.4

- means runtime error in unmodified program

= means run too short for resolution of system clock

Figure 2. Ratio of runtime to original.

Figure 2 shows the ratio of execution times. It is interesting that the increase in execution time is typically rather less than the increase in executable code size. Perhaps this is because both versions are slowed by cache misses, which are likely to be comparable in the two versions. Or perhaps long blocks, whose expansion is proportionately smaller, are executed more frequently than short ones.

We can tell Mahler, epoxie, and nixie to perform all the transformations for basic-block counting, but to leave out the actual counting code. This tells us how much of the expansion in code space and time is the overhead of stealing the bookkeeping registers and possibly doing some runtime address translation, as opposed to the instrumentation itself. Figure 3 shows the result. We can see that the expense of the bookkeeping overhead is very small: it is amusing that adding the overhead of nixie or epoxie made grr 8% *faster!* From this we see that the smaller time and space expansion of nixie in comparison to pixie must come from nixie's ability to do most address translation statically. We can also see that using nixie or epoxie to do optimization seems quite feasible, even if we are forced to introduce a little translation overhead. Mahler's time and space expansion due to overhead alone is nearly negligible: the largest

	code size			run time		
	mahler	*epoxie*	*nixie*	*mahler*	*epoxie*	*nixie*
ccom	1.00	1.02	1.02	1.01	1.11	1.11
doduc	1.00	1.02	1.03+*	-	1.04	1.03
eco	1.00	1.04	1.04	1.01	1.17	1.17
egrep	1.00	1.06	1.06	1.02	1.00	1.00
eqntott	1.00	1.06	1.07+	0.99	1.01	1.01
espresso	1.00	1.07	1.07+	1.00	1.00	1.00
fpppp	1.00	1.01	1.03+*	1.00	1.01	0.99
gcc1	1.00	1.03	1.03+	1.01	1.08	1.08
grr	1.01	1.05	1.05	1.01	0.92	0.91
li	1.00	1.02	1.02+	0.99	1.00	1.02
linpack	1.00	1.05	1.05	1.00	1.01	1.01
livermore	1.00	1.05	1.05	1.00	1.10	1.10
matrix300	1.00	1.03	1.05+*	-	1.00	1.00
met	1.01	1.05	1.05	1.01	1.00	1.00
nasa7	1.00	1.06	1.07+*	1.00	1.02	1.02
sed	1.00	1.05	1.05	1.00	=	=
spice	1.00	1.05	1.05+*	0.99	1.02	1.02
stanford	1.00	1.04	1.04	1.00	1.00	1.00
tomcatv	1.01	1.02	1.04+	1.00	1.02	1.02
whetstones	1.00	1.04	1.04	1.00	1.00	1.00
yacc	1.01	1.05	1.05	1.01	1.00	1.00

+ means runtime address table required; add 1.0

* means original code required; add 1.0

- means runtime error in unmodified program

= means run too short for resolution of system clock

Figure 3. Ratio of time and space with overhead code but not instrumentation code

increase in runtime is 2%, and the runtime even decreased slightly in several cases. The overhead is so small partly because the register allocation is integrated more closely with the instrumenter, but mostly because the Titan has 64 registers, so that taking a few away is less important than it is on the 32-register MIPS architecture.

10. Conclusions

A late code modification system that compromises between the integrated approach of Mahler and the conservative stand-alone approach of pixie is possible. We have prototypes of two such systems, nixie and epoxie, that require different amounts of compile-time information to be preserved. The overhead of these systems is usually quite small. Because nixie and epoxie correct the symbol table information -- something pixie could do but does not -- the resulting modified file can be run under the debugger to the same extent as the original. This may not matter for an instrumenting transformation (except to make it easier to debug the instrumentation process), but it is likely to be important if we use nixie or epoxie for optimizing transformations.

Nixie and epoxie work by understanding enough about the code in a program that they can assign meanings to all the jumps in a program, allowing the entire control structure to be determined. This should mean that we can do global analysis of that structure. Understanding the global control structure may let us look at using these tools for low-level global optimizations, such as interprocedure register allocation and pipeline scheduling. In any case we should be able to do instrumentation less intrusively.

Acknowledgements

Aaron Goldberg experimented with a less general form of nixie's approach as part of his summer project [5] at DECWRL, and I am indebted to him for several stimulating discussions.

Most of what I know about pixie I have learned by disassembling the code it produces, but along the way Earl Killian has given me a hint or three, for which I am most grateful. In particular Earl tipped me off about the run-time address translation, which is, as far as I know, entirely his invention.

My thanks for helpful criticisms of this paper to Anita Borg, Bill Hamburgen, and Scott McFarling.

References

[1] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long address traces from RISC machines: Generation and Analysis. *Seventeenth Annual International Symposium on Computer Architecture*, pp. 270-279, May 1990. A more detailed version is available as WRL Research Report 89/14, September 1989.

[2] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *Transactions on Programming Languages and Systems 6* (4), pp. 505-526, October 1984.

[3] Jack W. Davidson and Christopher W. Fraser. Register allocation and exhaustive peephole optimization. *Software--Practice and Experience 14* (9), pp. 857-865, September 1984.

[4] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 11-16. Published as *SIGPLAN Notices 21* (7), July 1986.

[5] Aaron Goldberg and John Hennessy. MTOOL: A method for detecting memory bottlenecks. WRL Technical Note TN-17, December 1990.

[6] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120-126. Published as *SIGPLAN Notices 17* (6), June 1982.

[7] John Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems 5* (3), pp. 422-448, July 1983.

[8] Norman P. Jouppi, Jeremy Dion, David Boggs, and Michael J. K. Nielsen. MultiTitan: Four architecture papers. WRL Research Report 87/8, April 1988.

[9] Earl A. Killian. Personal communication.

[10] Scott McFarling. Program optimization for instruction caches. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 183-191, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue).

[11] W. M. McKeeman. Peephole optimization. *Communications of the ACM 8* (7), pp. 443-444, July 1965.

[12] MIPS Computer Systems. *RISCompiler and C Programmer's Guide.* MIPS Computer Systems, Inc., 930 Arques Ave., Sunnyvale, California 94086. 1986.

[13] Michael J. K. Nielsen. Titan system manual. WRL Research Report 86/1, September 1986.

[14] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 28-39. Published as *SIGPLAN Notices 25* (6), June 1990.

[15] Peter A. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for LISP. *ACM Transactions on Programming Languages and Systems 11* (1), pp. 1-32, January 1989.

[16] David W. Wall. Global register allocation at link-time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 264-275. Published as *SIGPLAN Notices 21* (7), July 1986. Also available as WRL Research Report 86/3.

[17] David W. Wall. Link-time code modification. WRL Research Report 89/17, September 1989.

[18] David W. Wall and Michael L. Powell. The Mahler experience: Using an intermediate language as the machine description. *Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 100-104, October 1987. A more detailed version is available as WRL Research Report 87/1.

[19] David W. Wall. Register windows vs. register allocation. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 67-78. Published as *SIGPLAN Notices 23* (7), July 1988. Also available as WRL Research Report 87/5.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Technical Note TN-19, June 1991.