SEPTEMBER 1989

WRL Research Report 89/17

Link-Time 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 two other research laboratories located in Palo Alto, the Network Systems Laboratory (NSL) and 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 research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

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

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University 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`
Internet:	`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.

Link-Time Code Modification

David W. Wall

September, 1989

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

Abstract

Many existing or potential programming tools require the program to be completely recompiled with a special compiler option. This is usually inconvenient for the program developer, and may reduce the usefulness of the tool or the frequency with which the tool is employed. It may also require the maintenance of different versions of standard libraries, each compiled with the appropriate options for a different tool.

The difference between modules compiled with and without the special option is often simple and regular. If so, we can effect this difference by modifying the normally-compiled object code at link time, instead of recompiling. This reduces the overhead of using the tool by an order of magnitude, making it much more convenient.

1. Introduction

Recompiling an entire multi-module program from scratch is usually so expensive that one does it only reluctantly. In spite of this, many useful tools for program optimization or performance analysis require the recompilation of the entire program. The recompilation is done with a compiler option specifying that the resulting program will be used in connection with that tool.

A common example of this is the **gprof** [7] profiler. A **gprof** profile tells, among other things, how many times and by whom each procedure was called. The normal way to use **gprof** is to recompile all of your source modules with the compiler option **–pg** and then link the new object modules with standard libraries that have themselves previously been compiled with the same option. This results in an executable program augmented with instrumentation code to keep track of the calls that occur. Running this instrumented program produces a file of profile data, which the **gprof** program uses to produce a formatted, readable profile.

The requirement that you recompile your source modules is a stringent one. Global recompilation may take one or two orders of magnitude longer than the usual development cycle, and when you have obtained your profile you must then recompile everything once more in order to remove the instrumentation. On top of that, the library of instrumented library routines must be maintained as well. This makes system administration more complicated, and requires disk space for an instrumented version of every standard library. More subtly, the need for an instrumented library makes it more awkward to allow several fundamentally different *kinds* of instrumentation.

It turns out that the effect of compiling with **–pg** is quite small, hardly worth the expense of recompilation. An alternative is to modify the object code itself. If we build a step into the linker to convert uninstrumented object modules into instrumented ones, we can avoid the expense of recompiling altogether. Since the linker extracts the necessary library modules from the library, we can do the same transformation on these, eliminating the need for a separate instrumented library.

We first developed this capability to enable certain very global optimization. As part of the Mahler code generator for DECWRL's Titan, we built an interprocedural register allocator and pipeline instruction scheduler, which have been described elsewhere [20,21]. These optimizations required us to develop a technology of link-time code modification. Although we did not realize it immediately, it was an easy step to use the same machinery, usually in a simpler form, for performance instrumentation as well.

This paper is divided into two major sections. The first describes the technique of link-time code modification, in more detail than previous reports have done. The second describes a variety of different applications we have explored using this technique.

2. Related work

The notion of modifying object code is not fundamentally new. *Peephole optimization* is the process of improving generated code by making local transformations of it, and has been around for decades [15]. More recently, attempts have been made to characterize some or all of the normal optimization process as ''peephole'' optimization, by generating naive code and then transforming it into better code [5,6]. These techniques differ from ours in two ways. First, they have been used only for the purpose of optimization and not for performance analysis. More important, these techniques have been applied as the last part of compilation rather than as the first part of linking. This means that the compiler was free to keep the code in an internal or incomplete form, more convenient for transformation, instead of producing standard object modules that could be linked normally.

At the other extreme is the **pixie** tool developed at MIPS Computer Systems [16]. This tool transforms a fully-linked executable program into an equivalent instrumented executable. Thus it transforms the code *after* linking instead of before. **Pixie** was developed independently from our system but the possible modifications overlap with ours; the motivation for their system was quite similar to ours.

The **pixie** system works by transforming a finished executable instead of by transforming object files being linked. This has two consequences. First, their system is easier to use than ours. In our system the user must know how to perform the link step, so that the appropriate code modification option can be requested. On the other hand, our approach is easier to implement. An isolated object file, prior to linking, contains the relocation dictionary and loader symbol table, which make the transformation easier. For example, **pixie** must postpone some address translation until the modified program is executed, because not enough is known about the meanings of data items when **pixie** is performing the transformation [14]. In contrast, our system does all address translation as part of code modification. Because our approach is easier to implement, we have (I believe) used it in a wider variety of ways. No doubt any of our applications could be done within the framework of **pixie,** but the fact is that many of them have not.

3. Modifying code

Our system works as follows. The compiler always compiles code the same way (neglecting options that are irrelevant to this paper). The linker reads the object modules into memory one at a time, from object files or from libraries. If the user asks the linker for a transformation, then each module is modified in memory; no modified object file is written. The module rewriter is essentially independent of the linker proper. It accepts a module that the linker proper has read from a file, and it

returns a changed module that could just as well have been read from a file. The linker combines the object modules, modified or not, into a single executable image, which it writes to an executable file.

The linker has two obstacles it must overcome for our system to work. First, it must determine what transformations must be applied. These transformations might include changing, adding, or deleting instructions at various points in the module. Second, it must actually make the transformations, and convert the object module into an equivalent module that is internally consistent, so that it can then be linked correctly.

The problem of determining the changes to make can be easy or hard, depending on the transformation that is desired. For some applications, the compiler must mark certain points in the code so that the linker will know what to do there. Such a mark can be made by including a loader symbol whose value is the address marked. The compiler routinely includes all these marks, for all different kinds of instrumentation. A mark is irrelevant unless the user asks the linker for a transformation with which the mark is associated. Other important points can be determined from information that is in the normal object format. For example, the linker can determine the boundaries between basic blocks by looking for all text addresses in the loader symbol table and by looking at the destinations of all pc-relative branches.

This leaves the question of making the modifications correctly.

Changing an instruction is easy, and in fact all linkers do that routinely. Relocation is the process of filling in part of an instruction with address or offset information that is not known until link time. Normally this is done with the aid of a *relocation dictionary* that contains the addresses of instructions and data items that must be relocated, together with a reference to the loader symbol whose value determines the relocation to be done. The reason it is easy is that it is essentially context-independent. Once we know what change to make to the instruction, it can be made without affecting any other instructions.

Adding or deleting instructions is somewhat harder. When instructions are added, others must slide down to make room for them. When instructions are deleted, others must slide up to fill in the gap. Sliding an instruction up or down changes its address, and we must correct for this change if the module is to remain self-consistent. To do this, we must know all the places in code or data where the address appears, explicitly or implicitly, and we must modify these places so that the correct new address appears instead.

This turns out to be easy to do, because most such places are marked for relocation. For example, if the address of a procedure appears literally in some instruction or data word, it must be marked for relocation, because the address of the procedure may change as a result of the normal linking process. To put it another way, a normal linker *already* moves code up and down, because every module except the first must be positioned after all previous modules. Addresses that were correct relative to the beginning of the separately compiled module must be relocated when that module is

combined with others.

Addresses can also appear implicitly, in pc-relative operations. These may also be marked for relocation. But they might not; a pc-relative jump to a destination in the same module can normally be completely resolved at compile time, so link-time relocation would be unnecessary. Fortunately, pc-relative references can be recognized from their own appearance. Either the instruction is one that always makes an implicit pc-relative reference, like some branch instructions, or it explicitly uses the pc as an operand. In either case we can compute the effective address by combining the address of the instruction with the offset in the instruction.

Once we have found the places where addresses appear, we must know how to convert them into the correct new addresses. This requires us to make two passes over the module; one to plan the changes, so that we know how much correction is needed at each point, and one to commit the changes and correct the addresses that appear. (It is actually a little cleaner to go ahead and commit the changes in the first pass, but we must be careful not to lose the information that tells us where the addresses that need to be corrected appear.) After the first pass we can build a mapping from old addresses into new addresses, and use it to translate addresses in the second pass.

In constructing this mapping we must remember that when we insert code between two instructions we will think of it as being either *after* the first instruction or *before* the second. This makes a difference. If there is a branch to the second instruction, we need to know whether that branch should still jump to that instruction, or instead should jump to the inserted code. If we want to go on jumping to the original instruction, we must translate the destination address of the old branch by adding the length of the inserted code; if we want to jump to the inserted code then we leave the destination address unchanged.

It is important to remember that the transformation we are making is from one object module into another. We are not doing any of the operations one normally associates with the linking process; that will come later. All addresses in the old module are relative to the beginning of the module, and the same will be true of the modified module that we end up with.

We want to produce an object module in the usual form, just like those the linker might read from object files. So we must make sure it has a correct relocation dictionary and a correct loader symbol table. Addresses that appear in these must be corrected just as addresses in code or data are.

The symbol table is easy. Entries in it are tagged with types that tell us whether or not they are addresses. We modify them by looking them up in the mapping, just as if they had appeared in the code.

The relocation dictionary is easy, too. Each entry applies to one instruction or data item, designated by an address relative to the beginning of the module. We must correct the address, and also add or delete relocation entries that apply to instructions that are added or deleted. Correcting the addresses in relocation entries is different from the other address correction we have done, however. An address here specifies a

particular instruction rather than a location in the code. If we insert another instruction before a relocated instruction, we do *not* want the relocation suddenly to apply to the new instruction instead! The easiest way to accomplish this is to build a new relocation dictionary as we build the new version of its associated code segment, adding new relocation entries whenever we add an old or new instruction that requires relocation. The new address is then manifest because we just finished putting the instruction there.

To summarize, then, the general algorithm for modifying an object module is this. Based on the transformation requested, the linker determines the changes to be made. Some of these changes may be triggered by marks left by the compiler. In one pass it plans (and possibly makes) the individual changes. As it does this, it builds a mapping from old addresses to new addresses, in each case relative to the beginning of the module under consideration. On a second pass it commits the planned changes and corrects the addresses that appear in the changed code. During this second pass it also produces a new version of the code's relocation dictionary. It then goes on to correct the addresses that appear in the data segment and in the loader symbol table. The result is a new version of the object module, which can then be linked with others just as if it had been read from the object file.

4. Applications

We have developed a wide range of applications for our code modification technology. Some were for very special purposes, used to investigate a particular question and then essentially discarded. Others were for production program development tools that we continue to use routinely.

4.1. Interprocedural register allocation

The first application was Mahler's interprocedural register allocator. This system has been described previously [20] so only an overview will be given here.

At link time, the register allocator builds a call graph for the entire program and uses it to combine scalar variables into ''pseudo-registers.'' Two variables can be assigned to the same pseudo-register if they are local to procedures that are never simultaneously active, or if they are local to the same procedure and the compiler has determined that the two are never simultaneously live. Global variables and constants are assigned to singleton pseudo-registers. The pseudo-registers are then sorted in order of frequency of use, computed from estimates of variable reference frequency produced by the compiler. The most frequently-used pseudo-registers become real registers. The advantages of doing this at link-time instead of compile time are that globals can be safely included and that locals in different modules can be given the same registers or different registers depending on whether they interfere with each other.

Once the register allocator has decided which variables and constants should be kept in registers instead of in memory, the linker must rewrite the code to reflect that decision. It must remove loads and stores of these variables, and it must change the operands of other instructions so that they are the registers allocated to these variables

instead of temporary registers generated by the compiler. It must also deal with calls that are recursive or indirect through procedure variables, both of which require us to insert saves and restores at certain points.

To make all these changes correctly, the linker relies on very extensive marking by the compiler. Essentially every point where the value of a variable is loaded, used, computed, or stored is marked so that the linker can recognize it and know which variable is relevant. This information requires a lot of space; about half the space in a typical object module is register allocation marks. However, the pervasiveness of these marks made it easy for us to see other applications of the technique, even when the information required by the application was much less.

4.2. Pipeline instruction scheduling

The Mahler system also includes a pipeline instruction scheduler [21]. The scheduler rearranges instructions so that there are fewer places where dependencies between instructions cause hardware pipeline stalls. In contrast to the register allocation, the pipeline scheduler needs little global information. It simply breaks the module into basic blocks and re-orders each basic block independently. (In fact the basic blocks are not quite independent, as the scheduler may try to fill a branch slot in one block with an instruction from a block executed immediately after the first block.) It must also tolerate the presence of certain tricky idioms, such as a variable left shift implemented as an indexed jump into a table of constant left shifts, which it would be wrong to re-order. These idioms are marked by the compiler so that the scheduler can easily recognize them.

Thus the scheduler makes use of no intermodule information. This means that the scheduling could be done in the compiler instead of in the linker. It is done in the linker for the simple reason that it works much better if done after register allocation, because the transformations done in register allocation make large changes in the dependency structure of the basic block.

This is unfortunate in a production sense, because the scheduling is relatively expensive; when invoked, it slows down the linker by a large fraction. However, it has also allowed us to experiment with different pipeline structures. We created an interface that described a pipeline structure, and attached the interface both to the linker and to a fast instruction-level simulator. We could thereby schedule code according to the specified pipeline, and then simulate it according to the same pipeline. This gave us a more realistic estimate of the consequences of adopting a particular pipeline structure than we would have gotten had we merely simulated code produced for a different pipeline structure, or no particular pipeline at all.

The interface assumes that instructions fall into disjoint *classes*, such as add/subtract, logical operations, singleword load, and so on. The classes are hardwired into the interface, but are intentionally quite specific. It is quite unlikely that two instructions in the same class will ever have substantially different pipeline properties. The user can combine the classes into groups called *units.* If the architecture has a functional unit that does add/subtract and also logical operations, the user would group

these into the same unit. The user can then specify five properties of the pipeline. First, each class has a *result latency,* which is the span between the time an operation begins and the time its result can be used in a later instruction, including bypass. Next, each unit has an *issue latency,* which is based on how intensively the unit is pipelined; it is the span between the time this unit issues one operation and the time when it can issue a second operation. This may be independent of the result latencies of these operations. Third, each unit has a *multiplicity.* There may be several instances of a given functional unit, so as to exploit parallelism beyond that provided by the degree of pipelining. Some pipelines impose an *address latency* because a memory reference needs its address operand earlier in the pipeline than other operations. An address latency means essentially that an operation has a slightly longer result latency if its result is being used as the address in a memory reference. Finally, an architecture may allow several consecutive instructions to be issued in the same cycle, if they are mutually independent. This multiple issue is limited by the availability of the necessary functional units, and also by the *maximum multiple issue.* This is a limit on the maximum number of instructions that can issue in one cycle, regardless of whether more instructions and functional units are available.

The interface has been extended to specify other machine properties as well, including the number of registers and the detailed structure of its caches and cache buffers. We have used it for a variety of architectural studies [9,10,11,12,13] as well as to estimate the performances of machines proposed both within DECWRL and by product groups elsewhere in DEC.

4.3. Instruction-level instrumentation

When the pipeline scheduler was finished, the question naturally arose of how well it did. Code modification provided a convenient way to answer this question, and was DECWRL's first use of this technique for performance analysis rather than performance enhancement.

The idea behind the instruction-level instrumentation is that we create a set of global counters in the instrumented program, one for each event of interest. For example, we may be interested in the number of load instructions that are executed. Then we analyze each basic block, and insert code in it to increment all the counters associated with events that occur in that block. If we execute the resulting instrumented program, and then output the counters, we get dynamic counts of all the events.

It is easy to use this technique to count any kind of event that is apparent from a static analysis of the basic block. For instance, we can count the executed loads or stores or coprocessor instructions. We can also count events like the execution of delayed branches whose branch slots were not filled. We can even count the executions of various multi-instruction idioms such as byte load or store, or procedure call or return.

To a first approximation, we can also count pipeline stalls with this method. We examine the basic block for sequences that will cause a pipeline stall, such as (on the Titan) a store immediately followed by a load. We insert code to increment the

''store-load-stall'' counter by the number that we find in this block. This analysis is the same analysis that the scheduler itself must do.

Counting stalls in this way is only approximate, however, because a stall can happen at the boundary between two basic blocks. To make the counts precise, we create some auxiliary globals in the instrumented program, to record certain state information about the currently executing block. We check each block for instructions that might cause a stall, depending on what happens in the next block executed. When we find such instructions, we insert code to record the fact. For instance, if a basic block ends with a store instruction, we insert code that sets the ''ends-with-store'' bit. If a basic block begins with a load, we insert code that looks at the ''ends-with-store'' bit and increments the ''store-load-stall'' counter if it is set (and clears the bit in any case). This lets us exactly count events that depend on inter-block information.

Instruction-level instrumentation provides us with statistics that one normally acquires by instruction-level simulation. Inserting the instrumentation at link-time is expensive, but it is an order of magnitude cheaper than inserting it at compile-time would be. Executing the instrumented program is expensive, too; the instrumentation can slow the program by an order of magnitude. But simulating the program instead would be slower by two to four orders of magnitude. For this kind of application, link-time instrumentation seems to be the best approach by far.

4.4. Procedure-level profiling

The example of **gprof** was mentioned earlier in this paper. This profiler usually requires the recompilation of all the source modules in the program, with the option **–pg.** However, the only effect of this option is to insert, at the beginning of each procedure, a call to a special routine named **mcount.** This routine is responsible for keeping track of the calls that occur. In our approach, the compiler always marks the place where the call to **mcount** would be inserted. If the user tells the linker to link for **gprof,** then the linker inserts a call to **mcount** at each such point.

The advantages are obvious. We can use **gprof** without recompiling from scratch. We do not need to maintain a separate instrumented version of each library. The world is now easier and faster for both the user and the system administrator.

Others at DECWRL are experimenting with some extensions to this approach. Normally, **gprof** obtains execution times by ''pc-sampling.'' The profiled program is run in parallel with another process that periodically looks at the pc of the profiled program, to determine which procedure is executing. This approach has a few disadvantages. First, it means that assignments of expense are only statistical; if *A* calls *P* 90 times and *B* calls *P* 10 times, then **gprof** can only assume that 90% of the time spent in *P* was for the benefit of *A* rather than *B.* This may be just plain wrong; *A* may call *P* only for easy cases. (Maybe that's why it calls so much.) Second, it may make it difficult to profile the kernel of the operating system. Some machines may not allow the kernel to be interrupted. Even if the kernel can be interrupted, the timer interrupts themselves trigger certain events in the kernel, which means that pc-sampling based on the same timer interrupts may cause a skewed picture of where the

kernel spends its time. Third, call overhead itself may account for a significant fraction of a program's execution time, and the pc-sampling technique charges the caller and callee for the amount of overhead that occurs in each. Different calling conventions might result in a much different profile.

The Titan has a time-of-day clock that is visible to the user. We are experimenting with an alternative **gprof** that depends on the linker's ability to insert code noting the current time at key points. This will allow us to assign charges based on the actual current context. It will allow us to dispense with a parallel pc-sampling process, making kernel-profiling easier. And it will allow us to determine the expense of each call, charging whichever routine we wish, or reporting the time in some new way.

4.5. Block-level profiling

Just as we can insert code to keep track of the number of times we execute each procedure, we can insert code to count the executions of each individual basic block. Basic blocks are shorter than procedures, so we want to minimize the overhead of this counting. We therefore insert the counting code in-line rather than insert a call to a counting procedure. The linker allocates a long vector of counts, one for each basic block in the program, and the inserted code for a block increments the count associated with that block.

There are many applications of this information, most of which we have not yet explored. Our main use of the basic block execution counts is the construction of a *variable-reference* profile. Our intermodule register allocator normally selects the important variables by looking at estimates of variable reference frequency that are produced by the compiler. Often these estimates are not very good, and the allocation can be improved by using actual reference counts instead of estimates. A variable-reference profile tells us how many times each variable or constant was referenced during the program execution; if we feed it to the register allocator and relink, we get a better allocation and a faster executable. To build the variable-reference profile, we combine the basic block counts with static information that can be produced by the register allocator. This static information tells, for each basic block, how many variable references it contains, and which variables they are. This is easy for the register allocator to determine, because the references it counts are marked for removal in the event that the variable loaded or stored is assigned to a register.

We hope to use procedure profiles and basic block profiles more extensively, to guide other aspects of the optimization process. If the optimizer knows where the program spends its time, it should be able to spend more effort optimizing those parts.

4.6. Register management strategies

In 1988, we reported the results of a study comparing register allocation techniques and hardware register windows [22]. A modern processor probably has a lot of registers, and exploits them by keeping variables in them. Different techniques exist for accomplishing this. Some machines [8,9,17,19] treat the register set as a flat space and rely on the language system to allocate the registers to variables. Other machines

[2,3,18] include hardware to divide the register set into circular buffer of *windows.* When a procedure is called, the tail of the buffer is advanced in order to allocate a new window of registers, which the procedure can use for its locals.

Both techniques allow us to dispense with many loads and stores that would otherwise be needed. Both techniques also require us to include new loads and stores that would *not* otherwise be needed. In the case of register windows, the new loads and stores are needed when we run out of buffers after a long series of calls or returns. In the case of register allocation, the new loads and stores are needed when we make procedure calls that would otherwise harm registers in use. In a naive allocation this might be every procedure call; our more sophisticated allocation inserts saves and restores only for recursive or indirect calls.

Thus we can imagine evaluating a hardware or software register management technique, by counting the loads and stores saved and then subtracting the loads and stores inserted. Our study did this via link-time code modification. Register allocation techniques were measured by inserting code to count the different kinds of loads and stores. Register window techniques were measured by inserting code to maintain a simple data structure describing what the current state of the window buffer would be on a window machine with specified parameters. The results showed that hardware windows did not have a significant advantage over register allocation.

4.7. Address traces

An address trace is the sequence of instruction and/or data addresses referenced by a program during the course of its execution. Such a trace is useful in simulating the performance of different cache configurations. Previous techniques for acquiring address traces fall into two general categories. One approach involves adding microcode to the machine [1]. The microcode watches the address bus and logs those it sees. This approach slows execution by an order of magnitude, and is not suitable for machines without microcode, like most modern RISC processors. The other approach is simulation. We can build an instruction-level machine simulator that also logs the memory references made by the simulated program. Unfortunately, simulation is very slow; simulating a program typically takes two to four orders of magnitude longer than executing it. With the advent of machines with very large caches, we must use very long address traces to get realistic and useful results. Simulation is too slow to do this easily.

Our approach was to use link-time code modification to instrument the code. Wherever a data memory reference appears, the linker inserts a very short, stylized call to a routine that logs the reference in a large buffer. The same thing is done at the beginning of each basic block, to record instruction references. When the buffer fills up, something is done to process it; either it is dumped to an output device, or a cache simulation is resumed to process it.

It is important that we not consider single programs in isolation, because the cache is shared by all user processes and also by the system kernel. Others at DECWRL [4] are extending this technique by making the system kernel allocate a

shared buffer that everybody can use; this allows the traces of kernel and user activity to be correctly interleaved. Some interesting work in the operating system is necessary, to synchronize things properly and to correct for the effect of the execution slowdown on the behavior of of time-triggered events in the kernel.

5. Conclusions

Link-time code modification is easy to do. The format of object modules and the structure of the linker provide a convenient interface for applying various transformations on each module being linked. Where necessary, the compiler can routinely provide hints that are ignored unless a relevant transformation is requested.

Link-time code modification is flexible. We have used it for a wide variety of one-shot and production applications. We have used it as a vehicle for optimization techniques that can make impressive differences in the speed of programs. It has enabled performance instrumentation from the level of individual instructions up to the level of user procedures; almost by accident it has changed the standard tool **gprof** from something one might use a few times a year into something one might use a few times a month or week. We expect to be able to add to its uses for some time to come.

6. Acknowledgements

Michael L. Powell implemented the first pipeline scheduler and instruction-level instrumenter. Scott Nettles worked on using the time-of-day clock to improve the applicability of **gprof.** Anita Borg is pursuing the operating system issues involved in address traces. Norm Jouppi was instrumental in helping to design the interface for machine specification described in the section on pipeline scheduling.

References

[1] Anant Agarwal, Richard L. Sites, Mark Horowitz. ATUM: A new technique for capturing address traces using microcode. *Proceedings of the 13th Annual Symposium on Computer Architecture,* pp. 119-127. Published as *Computer Architecture News 14* (2), June 1986.

[2] Advanced Micro Devices. *Am29000 Streamlined Instruction Processor User's Manual.* Advanced Micro Devices, Inc., 901 Thompson Place, P. O. Box 3453, Sunnyvale, CA 94088.

[3] Russell R. Atkinson and Edward M. McCreight. The Dragon processor. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 65-69. Published as *Computer Architecture News 15* (5), *Operating Systems Review 21* (4), *SIGPLAN Notices 22* (10), October 1987.

[4] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long address traces from RISC machines: Generation and Analysis. WRL Research Report 89/14.

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

[6] 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.

[7] 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.

[8] John L. Hennessy, Norman P. Jouppi, Steven Przybylski, Christopher Rowen, and Thomas Gross. Design of a high performance VLSI processor. In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration,* pp. 33-54. Computer Science Press, 11 Taft Court, Rockville, Maryland.

[9] Norman P. Jouppi and Jeffrey Y.-F. Tang. A 20 MIPS sustained 32 bit CMOS microprocessor with high ratio of sustained to peak performance. To appear in *IEEE Journal of Solid-State Circuits.*

[10] Norman P. Jouppi. The non-uniform distribution of instruction-level and machine parallelism and its effect on performance. Submitted to the Dec. 1989 special issue on computer performance of *IEEE Transactions on Computers.*

[11] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems,* pp. 272-282, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue). Also available as WRL Research Report 89/7.

[12] Norman P. Jouppi. Packaging and integration plateaus of processor performance. To appear in the 1989 International Conference on Computer Design.

[13] Norman P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. *Proceedings of the 16th Annual International Symposium on Computer Architecture,* pp. 281-289, June 1989.

[14] Earl Killian. Personal communication.

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

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

[17] Michael J. K. Nielsen. Titan system manual. WRL Research Report 86/1. Digital Western Research Laboratory, 100 Hamilton, Palo Alto, CA 94301.

[18] David A. Patterson. Reduced instruction set computers. *Communications of the ACM 28* (1), pp. 8-21, January 1985.

[19] George Radin. The 801 minicomputer. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47. Published as *SIGARCH Computer Architecture News 10* (2), March 1982, and as *SIGPLAN Notices 17* (4), April 1982.

[20] 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.

[21] 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. A more detailed version is available as WRL Research Report 87/1.

[22] 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.

''A Simulation Based Study of TLB Performance.''
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

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

''TurboChannel T1 Adapter.''
David Boggs.
WRL Research Report 91/4, April 1991.

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

''Don't Fidget with Widgets, Draw!.''
Joel Bartlett.
WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.''
G. May Yip.
WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.''
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.''
David W. Wall.
WRL Research Report 91/10, August 1991.

''Network Locality at the Scale of Processes.''
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

''Cache Write Policies and Performance.''
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

''Packaging a 150 W Bipolar ECL Microprocessor.''
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

''Observing TCP Dynamics in Real Networks.''
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

''Systems for Late Code Modification.''
David W. Wall.
WRL Research Report 92/3, May 1992.

''Piecewise Linear Models for Switch-Level Simulation.''
Russell Kao.
WRL Research Report 92/5, September 1992.

''A Practical System for Intermodule Code Optimization at Link-Time.''
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

''A Smart Frame Buffer.''
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

''Recovery in Spritely NFS.''
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

''Tradeoffs in Two-Level On-Chip Caching.''
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

''Unreachable Procedures in Object-oriented Programing.''
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

''Limits of Instruction-Level Parallelism.''
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

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.

‘‘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.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.