AUGUST 1991

WRL
Research Report 91/10

Experience with a Software-Defined Machine Architecture

David W. Wall

Experience with a Software-Defined Machine Architecture

David W. Wall

August, 1991

Abstract

We built a system in which the compiler back end and the linker work together to present an abstract machine at a considerably higher level than the actual machine. The intermediate language translated by the back end is the target language of all high-level compilers and is also the only assembly language generally available. This lets us do intermodule register allocation, which would be harder if some of the code in the program had come from a traditional assembler, out of sight of the optimizer. We do intermodule register allocation and pipeline instruction scheduling at link time, using information gathered by the compiler back end. The mechanism for analyzing and modifying the program at link time was also useful in a wide array of instrumentation tools.

1. Introduction

When our lab built its experimental RISC workstation, the Titan, we defined a high-level assembly language as the official interface to the machine. This high-level assembly language, called Mahler, represents a compromise between a traditional assembly language and a compiler intermediate language. It is the target language for all high-level language compilers, and it is also the only assembly language available for general use. For this reason we say that the Titan/Mahler combination is a machine with a *software-defined architecture:* hardware with a software face.

We had two reasons for this approach. First, programming a RISC machine in a true assembly language can be quite unpleasant. We expected this to be especially true for us, because the Titan's instruction set is reduced even further than most RISC machines that preceded or followed it [1,5,16,24,26,31,33], and moreover we expected the architecture to change somewhat from generation to generation. Second, we wanted to do very global optimization, including intermodule promotion of variables to registers. Traditional assemblers can interfere with this by allowing actions that the optimizer does not know about; this forces the optimizer to make conservative assumptions.

Mahler is not intended to be a machine-independent intermediate language, an idea going back thirty years or more [37]. Its structure is intentionally quite close to that of the Titan family, and it is more appropriate to think of it as a high-level assembly language. To put it another way, Mahler provides the same sort of clean interface for the Titan that a CISC architecture provides for its microcode engine. Within the envelope of this interface, the assembler and linker can safely perform optimizations that require knowledge of the entire program.

This paper describes our experience with Mahler. Section 2 briefly contrasts the Titan hardware with the Mahler interface. Section 3, the bulk of the paper, describes the implementation of the Mahler assembler and linker, and describes the optimization and instrumentation capabilities of the system. Section 4 contains quantitative measurements of the effectiveness of Mahler's link-time optimizations.

2. Titan and Mahler

The Titan [29] is an experimental, high-performance, 32-bit scientific workstation developed at Digital Equipment's Western Research Lab (WRL). It is the first of a family of machines that we have designed. There have already been two different versions of Titan itself and a substantially different CMOS machine we nevertheless call the MicroTitan [23]. These machines are generally similar in that each is a pipelined load-store architecture with a large register set and a floating-point coprocessor. They are substantially different in the size of the register set, the details of the pipeline, and the interface and capabilities of the coprocessor.

Both the Titan and the MicroTitan are load-store architectures. Memory is accessed only through simple load and store instructions with a base register and a 16-bit displacement.

Early versions of some of this material appeared in *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (1986) [40], *Second International Symposium on Architectural Support for Programming Languages and Operating Systems* (1987) [44], and *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (1988) [41].

Other operations like addition and subtraction are three-register instructions that cannot access memory. Branches and jumps are *delayed* [31]: the transfer of control does not occur immediately after the branch instruction, but rather one instruction later than that. The instruction executed after a branch is said to occupy the *branch slot.* Filling the branch slot with a useful instruction rather than a no-op is desirable.

The CPU pipeline can stall for one cycle at a memory reference, depending on the instruction that preceded it, or for tens of cycles at a coprocessor operation if it tries to fetch a coprocessor result that is not ready yet. These stalls are implemented by the hardware and do not require explicit no-ops. Nevertheless, we would like to avoid these wasted cycles by scheduling other instructions to fill them whenever possible.

Mahler is the intermediate language used by all of the front-end compilers implemented for the Titan. It has a uniform infix syntax for operations performed on addresses, signed integers, unsigned integers, single-precision reals, and double-precision reals. These operations are performed on constants or named variables. A variable may be either local to a procedure or global. Some variables are user variables from the source program, while others are temporary variables generated by the front-end compiler. Branches in Mahler are not delayed. There is no explicit concept of a register in Mahler; it is as if there are unboundedly many registers available, each corresponding to a named scalar variable. Procedures are explicitly declared and explicitly called, as in a high-level language.

Thus Mahler hides several machine-level aspects of the Titan: the registers, the pipeline, the delayed branches, and the asynchronous coprocessor. Nevertheless we want to treat the Mahler interface as the official machine description, and as the only available assembly language. The Mahler implementation therefore needs to use these resources well, so that performance is not degraded and Mahler's clients do not wish that they had direct control over them.

We accomplish this by taking advantage of the fact that Mahler is the official machine description. This means that it has complete control over an entire program, regardless of what high-level languages it is written in, and regardless even of whether ‘‘assembly language’’ subroutines are included. The Mahler implementation uses this complete control to perform several code improvements at link time, when the entire program is available for analysis.

3. The Mahler implementation

Figure 1 shows an overview of our language system. The Mahler implementation consists of the Mahler assembler and an extended linker. A high-level language compiler or a human writes source files in the Mahler language. The assembler reads the Mahler source files and produces object modules. These object files can be linked in the usual manner to form an executable program, or they can be improved at link time by invoking an intermodule register allocator and an instruction scheduler. They can also be instrumented at link time in various ways. The assembler extensively annotates the object modules with information that enables these link-time transformations to be done quickly and correctly.

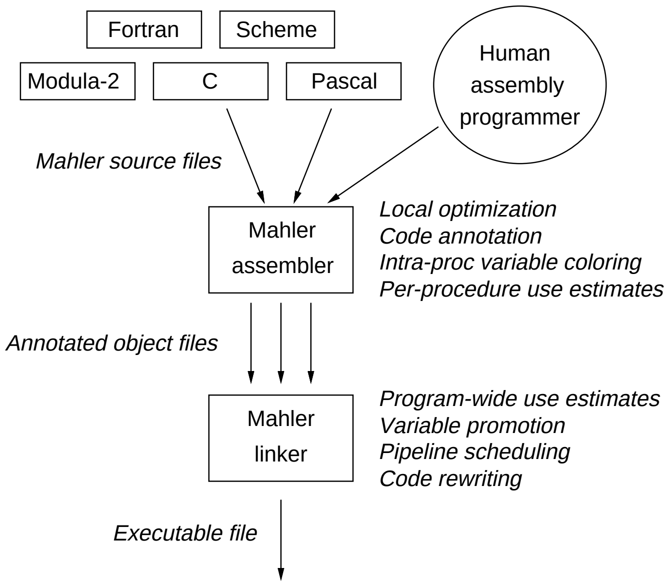

Figure 1. The Mahler implementation.

3.1. The Mahler assembler

Code selection on a RISC is easy because there are few choices. Since Mahler is at a higher level than the hardware, it is both possible and desirable to do a number of classical optimizations. Some are global optimizations based on data-flow analysis, but most are local optimizations like local strength reduction. On the Titan, for example, multiplication by an integer constant can be profitably replaced by the sum or difference of as many as four shifts.*

The Mahler assembler starts by parsing the Mahler source file and producing an internal form that represents each basic block as a directed acyclic graph (DAG). All of the local optimization is then performed on this DAG structure. Within each basic block, the DAGs for common subexpressions, including multiple uses of the same variable value, are merged when possible.

After DAG optimization, the assembler traverses the DAG, generating code for each node in turn. It allocates temporary registers from a small set of eight reserved for expression evaluation. This number of temporaries is large enough that most basic blocks do not have to spill these registers to memory, but small enough to leave as many registers as possible for global allocation.

In general, the assembler generates simple fast code whose performance will be tolerable even without link-time improvement. It makes no attempt to keep variables in registers, except locally within a basic block. For the most part, it does not try to avoid pipeline stalls, and it generates a no-op after each branch. The exceptions to this occur only inside ‘‘idioms,’’ single operations in Mahler that require sequences of several instructions at the machine level. One example of an idiom is procedure call; another is variable left shift, which is carried out as an indexed jump into a table of thirty-two constant left shifts. These idioms are somewhat like a

* Since constants between 1 and 170 can all be handled this way, this eliminates all the multiplications in many programs. Bernstein [7] explored this technique more generally.

set of standard macros, and we tailored their expansions to do useful work during pipeline delays and branch slots whenever we could.

Part of the assembler's job is to collect and record information to help the link-time register allocator and pipeline scheduler. It builds a table of the variables and procedures that are defined or referenced in the module, and includes this table in the object file. It also annotates the code heavily with a kind of generalized relocation information, which the register allocator uses to remove loads and stores of variables that it promotes to registers. Finally, it flags certain code sequences in idioms as unchangeable so the pipeline scheduler will know not to harm them.

In addition, the assembler optionally does a dataflow analysis on each procedure to determine whether two locals of the same procedure are ever simultaneously live. If they aren't, the link-time register allocator can decide to keep them in the same register.

When the assembler has finished with all of the source files, the resulting object modules can be linked in the usual manner. Alternatively, the linker can also be requested to perform register allocation and instruction scheduling based on the information the assembler has included with each object module.

3.2. The Mahler linker

The interesting part of the Mahler implementation is the linker. The Mahler linker is a standard linker, augmented with a variety of code transformation options. These transformations include optimizations like inter-module register allocation and pipeline instruction scheduling. They also include various kinds of high-level and low-level instrumentation.

The code transformation system is invoked by the linker proper, but is essentially independent of it. The linker reads into memory the modules to be linked, from object files or from libraries, and passes each in turn to the module rewriter. Depending on the requested transformation, the module rewriter changes the module in memory and returns it to the linker proper. The changed module has the same format as a module read from a file, so several independent transformations can be done in succession. After all the transformations, the linker combines the resulting object modules into a single executable image, which it writes to an executable file.

Some transformations are easy and do not depend on much context; others are harder and require knowledge of the program structure, sometimes even at the source level. This kind of information is collected by the assembler and passed in the object module to the linker, which can then combine the information from all of the modules if needed. The assembler includes this information routinely, so it will be present no matter what transformations we ask the linker to do.

Transformations require us to insert, delete, or change individual instructions. Inserting and deleting is the hard part, because this causes the addresses of code and data locations to change. To correct for this, we must do two things: compute a mapping from old addresses to new ones, and find each place in the program where an address is used and identify the address referenced there.

Computing the mapping is straightforward because we know where we are inserting or deleting instructions. We simply keep track of how much insertion and deletion has been done at each point in the program. We must be careful, however, to identify code inserted between two instructions as either *after* the first instruction or *before* the second. The destination of a branch to some instruction should be changed to point to new code inserted before that instruction, but not to new code inserted after its predecessor. The computation of the correct mapping of this instruction's address must distinguish between these two similar-looking insertions.

Finding the places where addresses are used also turns out to be relatively easy. Each object module contains a loader symbol table and relocation tables that 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 it must be resolved during normal linking, and the relocation tables tell the linker what kind of address it is. This same information also lets the module rewriter 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 branch's displacement.

Inserting and deleting instructions can have an effect on span-dependent branches. We believed that this would not be a problem, and it never was; our span-dependent branches were used only within a procedure, and no procedure was ever too big. Handling span-dependent branches is well-understood, however, and incorporating a classical algorithm [38] would not be hard.

We are translating individual object modules into new object modules, so we must also translate the loader symbol table and relocation tables. Addresses in the symbol table are tagged with types that tell us how to correct them. Addresses in the relocation tables are a bit different, since they identify specific code or data words, rather than locations; even if we insert a new instruction, the relocation entry should apply to the same instruction as before! We therefore do not translate these addresses by looking them up in our mapping; instead we build the new relocation table as we build the new segment, appending an entry whenever we append the associated instruction.

In summary, the algorithm for modifying an object module is as follows. Based on the transformation requested, the linker determines the changes to be made, possibly using information left by the assembler. In one pass over the module it makes the individual changes. As it does this, it builds a mapping from old addresses to new, in each case relative to the beginning of the module under consideration. On a second pass it corrects the addresses that appear in the changed code. During this second pass it also produces a new version of the code's relocation table. 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. This new object module can then be linked with others just as if it had been read from an object file, or can be passed in turn to another transformation for further modification.

The Mahler linker can perform many different transformations in this way. Perhaps the most interesting is the intermodule register allocation, to which we turn next.

3.3. Intermodule register allocation

A good way to use a lot of registers is to *promote* variables: allocate registers for them to live in for long periods of time. Chaitin et al. [9,10] and Chow [11] pioneered a technique that combines this with ordinary local register allocation. A key idea is that the same register can be used for many different variables if this causes no conflicts. The compiler starts by doing a liveness dataflow analysis. It then builds a conflict graph in which an edge appears between two variables if they are ever live at the same time. Two variables that are never live at the same time can be promoted to the same register. It follows that a *coloring* of this graph is equivalent to a register allocation for the variables. (A coloring of a graph is an assignment of ‘‘colors,’’ one per vertex, subject to the constraint that adjacent vertices must be different colors. A *minimal coloring* is one with the fewest possible colors. Good linear-time coloring heuristics exist, even though the problem of finding a minimal coloring is NP-complete.)

Applying this technique across procedure boundaries requires some form of inter-procedural dataflow analysis. This in turn requires a more complete picture of the program than is usually available to a compiler in an environment of separately compiled modules. We might approach this problem by compiling all the modules in one monolithic compilation. More recently, some compilers have maintained persistent program databases [34], triggering recompilation of modules when new information shows this to be necessary. In either case, it is awkward to include library routines in the process, since users don’t want to worry about what library modules are included.

Our solution is to wait until link time to select the variables to promote. We can pick the globals and locals to promote using program-wide safety and payoff information. We can keep an important global in the same register across all procedures. And when one procedure calls another, we can keep the locals of the two procedures in different registers. Spills and reloads around calls are not necessary.

Simply postponing the coloring technique until link time is impractical, however. If we wait until link time and then do a dataflow analysis, conflict graph construction, graph coloring, and final code generation, we will be doing the bulk of the back end’s work at link time. Rebuilding a program after a change will take almost as long as if we used monolithic compilation. Moreover, algorithms for dataflow analysis and conflict graph construction require slightly more than linear time. The cost may be acceptable if we apply the algorithms repeatedly to a lot of smaller modules or procedures, but we have found it too expensive to apply them to an entire program.

The call-return structure of most programs gives us an easy way to conservatively approximate the interprocedural liveness information. Our link-time register allocator assumes that a local variable is live for exactly as long as its procedure is active.* The assembler generates

* We can optionally refine this assumption with an intra-procedural coloring done by the Mahler assembler. See section 3.3.7.

code assuming that variables live in memory, loading them into temporary registers as needed within a basic block. Along with the object code, it produces extra information about the uses of variables and procedures throughout the code. If invoked, the intermodule register allocator picks the variables to promote and uses this extra information to transform each module accordingly.

Generating code that is correct as it stands but that can be easily modified later is attractive for three reasons. First, it means that the promotion of variables is entirely optional. If the expense of global register allocation is unwarranted, we can omit it. Second, it means we are not forced somehow to allocate registers for all the variables in the program. If we manage to keep the most frequently used variables in registers, we can keep the rest of them in memory and the results are still correct. Finally, it keeps us honest as designers of the system. Once we postpone anything until link time, the temptation is great to postpone everything, so that we can know what the other modules look like. The requirement that we generate working code at compile time assures that we will not succumb to that temptation.

To make the global promotion of variables optional, the assembler sets aside some registers as expression temporaries. These temporaries will not be available for link-time allocation to promoted variables. This is contrary to the philosophy of Chaitin et al., who included expression temporaries in the conflict graph, and introduced spills and reloads in order to reduce the complexity of the conflict graph to the point where it was colorable using the number of registers available. This meant that global register allocation was an integral part of code generation and could never be omitted.

3.3.1. Annotated object code

The assembler's main contribution to link-time register allocation is to explain the code it produces, by annotating it with *register allocation actions.*

action	*explanation*
REMOVE.*v*	delete the instruction
OP1.*v*	replace operand 1 by the register allocated to *v*
OP2.*v*	replace operand 2 by the register allocated to *v*
RESULT.*v*	replace the result register by the register allocated to *v*
LOAD.*v*	replace ‘‘temp := load *v*’’ by ‘‘temp := reg *v*’’
STORE.*v*	replace ‘‘store *v* := temp’’ by ‘‘reg *v* := temp’’
KEEP.*v*	do not delete the instruction, overriding any REMOVE

Figure 2. Register allocation actions invoked if *v* is promoted to a register.

These actions tell the linker how to change the code if it decides to promote some of the variables to registers. Typically the linker removes loads and stores of a promoted variable, and modifies each instruction that uses the loaded or stored value so that it references the newly allocated register instead of the original temporary. Occasionally the structure of a basic block forces variations of this. The annotations discussed in this paper appear in Figure 2. Each action is qualified by its variable name, and will be ignored if that variable is not promoted.

The simplest case is when any of the loads or stores can be removed. For example, the assignment ‘‘*x* := *y* + *z*’’ would lead to annotated code like this:

instruction	*actions*
r1 := load *y*	REMOVE.*y*
r2 := load *z*	REMOVE.*z*
r3 := r1 + r2	OP1.*y* OP2.*z* RESULT.*x*
store *x* := r3	REMOVE.*x*

(In this example, registers r1, r2, and r3 are temporaries managed by the assembler.) For instance, if we decide to keep *y* in a register, we will delete the load of *y* and change the addition instruction so that its first operand is the register allocated for *y* instead of the temporary register. The three examples below show the rewritten code that results from various link-time selections of register variables.

register y	*register x,y*	*register x,y,z*
r2 := load *z*	r2 := load *z*	*x* := *y* + *z*
r3 := *y* + r2	*x* := *y* + r2	
store *x* := r3		

If a value will not be available in the variable for as long as it is needed, the linker must replace the load by a register move rather than deleting it altogether. Figure 3 shows the two possible cases. In the first case, the variable *x* is not changed until after the last use of its original value; in the second case, *x* is reassigned before we are through. This could happen because of local optimizations like copy propagation, and means we must keep the value in the temporary and accordingly must replace the load by a register move.

instruction	*actions*	*instruction*	*actions*
r1 := load *x*	REMOVE.*x*	r1 := load *x*	LOAD.*x*
... := r1 + ...	OP1.*x*	... := r1 + ...	
... := r1 - ...	OP1.*x*	store *x* := ...	
store *x* := ...		... := r1 - ...	

Figure 3. Register actions if value is always available in *x* (left) and if *x* gets a new value before the old value is dead (right).

A similar analysis applies in the case of a store. When a value is computed and immediately stored to a promoted variable, we can find subsequent uses of that value in that variable *if* we make no new assignment to the variable before the last use of the value.

A simple assignment like ‘‘*x* := *y*’’ is a special case, because the analysis sketched above would cause both the load and the store to be marked for removal if both *x* and *y* are promoted to registers, which is clearly wrong. Instead, we use the KEEP action:

instruction	*actions*
r1 := load *y*	REMOVE.*y* RESULT.*x*
store *x* := r1	STORE.*x* REMOVE.*x* KEEP.*y* OP1.*y*

The KEEP.*y* action overrides the REMOVE.*x* action if both *x* and *y* are promoted. We end up with a single instruction if we promote either or both.

This three-part analysis of loads, stores, and simple assignments lets us correctly annotate the code we generate. We begin with a forward and backward pass over the basic block. This step marks each operation in the basic block with the variable or variables, if any, in which its result will always be available. This is followed by the generation of the annotated code. Appendix 1 sketches these algorithms. Some generated instructions may be given redundant annotations, such as two different OP1 actions. This just means that the value is available in two different variables, and either can be used. We don't know in advance which variables will be promoted, so this redundancy can be helpful.

By analyzing the basic block this carefully, the assembler is essentially planning the code needed for any possible combination of promoted variables. The rest of the work is done by the linker, which chooses which variables to promote, and uses the assembler's annotations to rewrite the code based on that choice.

3.3.2. Register allocation and module rewriting

The first thing the register allocator must do is decide which variables to promote. To help it, the assembler collects use information about each module it translates, and records this information in the object file. This information includes a list of the procedures appearing in the module and a list of the procedures each calls. It also includes an estimate of how many times each procedure uses each variable. The assembler computes this estimate for each procedure by counting the static number of times the reference is made, with references inside loops, no matter how deeply nested, counting as 10. This is a coarse estimate, but it seemed to work better than, for instance, multiplying by 10 for each surrounding loop.

The register allocator in the linker collects the use information for all the modules being linked, and then builds a call graph for the program. If there are no recursive or indirect calls, then this call graph is a directed acyclic graph (DAG). Recursive and indirect calls require special handling, which we will describe shortly.

The allocator then estimates the dynamic number of references of each variable, by multiplying each of the assembler's per-procedure estimates by the static number of places the procedure is called. (In effect, we are estimating that each procedure is called exactly once from each call site. We tried getting a better estimate by traversing the call graph and multiplying estimated counts, so that if P calls Q ten times and Q calls R ten times, we estimate 100 calls for R. Since the assembler's estimates are only rough guesses, this tended to give wildly wrong counts, some even overflowing a 32-bit integer. The current more conservative scheme works better.)

The idea behind our allocation of registers to local variables is simple. If two variables are local to different procedures, and these procedures are never simultaneously active, then the two variables are never simultaneously live. These two locals can therefore be grouped together and kept in the same register.* We do this grouping by traversing the call DAG in

* Michael L. Powell is responsible for this insight, for which I am most grateful.

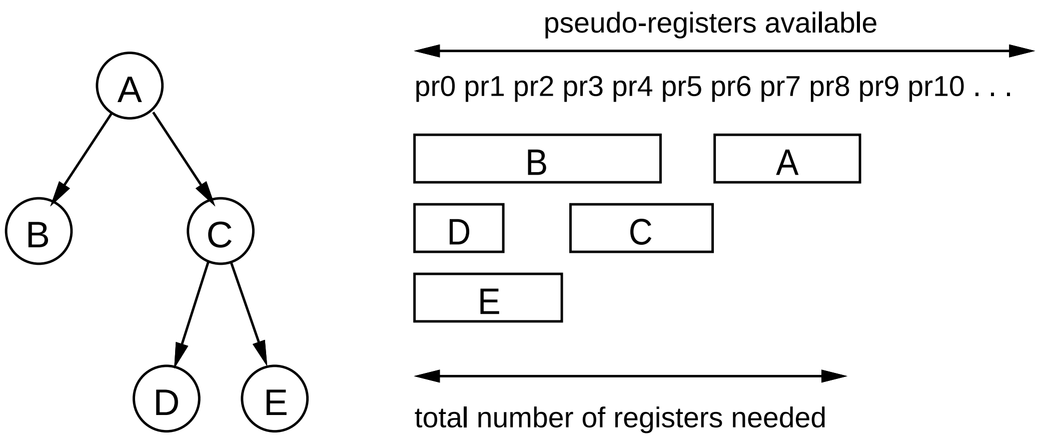
pseudo-registers available
pr0 pr1 pr2 pr3 pr4 pr5 pr6 pr7 pr8 pr9 pr10 . . .
A
B
C
D
E
B
A
D
C
E
total number of registers needed

Figure 4. A sample call DAG, with arrows showing calls in the program, and the allocation of pseudo-registers to locals of the procedures.

reverse depth-first search order [4]. We start by allocating *pseudo-registers* to the locals of each leaf procedure, beginning with pseudo-register 0. We then allocate pseudo-registers to the locals of other procedures, beginning immediately after those used by its children. Figure 4 shows an example call DAG and the associated allocation of pseudo-registers to the procedures' locals. The algorithm[*] is given in Figure 5.

```
for each proc p in reverse depth-first search order do:
    – –  First find the largest pseudo-register allocated in a
    – –         previous iteration to the children of this procedure.
    – –  Leaves have no children.
    childPseudos := 0
    for each child q of p in the call DAG do:
        childPseudos := max (childPseudos, pseudos[q])
    – –  Now allocate pseudo-registers to p's locals, starting
    – –         where its children left off
    pseudos[p] := childPseudos
    for each local v of p do:
        allocate pseudo-register number "pseudos[p]" to v
        pseudos[p] := pseudos[p] + 1
```

Figure 5. Assigning locals to pseudo-registers.

[*] My thanks to Richard Beigel for this elegant algorithm.

The register allocator then creates a pseudo-register for each individual global scalar. It gives each pseudo-register a reference frequency that is the sum of the frequencies of its variables, and sorts the pseudo-registers by frequency. Finally, it allocates an actual register to each of the most frequently used pseudo-registers.

A variable whose address is taken anywhere in the program is not eligible for promotion. This is an extremely conservative decision and might be relaxed by using dataflow analysis and type information to determine the places where the address might be used.

When we have decided which variables to keep in registers, we know which register actions to apply to the code of each module. Applying these actions is easy since each one is independent of context; the assembler did the hard work when it generated the actions in the first place.

3.3.3. Initialization

A global variable may be declared with an initial value. This value is preset in memory before the program begins to execute, so no instruction is responsible for assigning the value to the variable. If we promote such a variable, we must somehow get that initial value from the original memory location into the register. The assembler identifies these globals in the use information it puts in the object file. The startup routine that sets up the program environment before invoking the user main program includes a special register action, called INIT, that is unconditionally performed as part of register allocation. This action inserts instructions to load the initial value of each initialized global that gets promoted to register.

A local variable declared with an initial value does not present this problem. The front-end compiler generates ordinary Mahler statements to assign the variable's initial value at the beginning of the procedure. The Mahler assembler then annotates this code just like code for explicit user statements.

A parameter is more like a global than like a local. The assembler assumes that parameters are passed on the stack, so the argument value is already in the parameter location when we enter the procedure. If a parameter is promoted, the module rewriter must insert code at the beginning of the procedure to load that value into the register.*

3.3.4. Indirect and recursive calls

Our algorithm for assigning pseudo-registers to variables depends on building a directed acyclic graph to describe the static call structure of the entire program. Indirect calls don't appear on the DAG at all, and recursive calls make the graph cyclic. Something must be added to handle these kinds of calls.

One solution to the problem of indirect calls is to represent an indirect call by direct calls to each possible destination procedure. We could do this naively, by listing every procedure that is ever assigned to a procedure variable. We could also do it more carefully, by using type

* In fact, we can do better, by having the caller put the argument value directly into the parameter register. We discuss this in section 3.3.5.

information and flow analysis to narrow down the possible candidates. The naive approach seems a bad idea, but the careful approach might well be appropriate for an object-oriented program with a preponderance of indirect calls.

Our solution is different. We don't record indirect calls in the call graph, and we insert a non-trivial spill and reload around an indirect call. In this approach an indirect call archives *all* of the apparently active locals. When we arrive at the called procedure we can then behave as if we had performed a normal call visible in the call DAG. The set of active locals consists of each local whose procedure is on any path in the call DAG from the root to the calling procedure.

We chose our solution because it seemed likely that few of our applications would make heavy use of indirect calls; it was also easier. This seems not to have hurt us: a program whose author [30] describes it as ''object-oriented'' seems to benefit as much as the others.

We should note that our solution works only because we allow indirect calls only to top-level procedures. Suppose that P and Q are both nested in R and both refer to R's locals, and suppose that P calls Q indirectly. Then archiving R's locals around the call is wrong because Q might legally change one of them.

For many years we thought recursion could be handled analogously to indirect calls. In an early report [40], we described a system that did so. We removed recursion from the call graph by building a depth-first spanning tree [4] and discarding the back edges. This gave us an acyclic graph on which we could perform our usual algorithm. At the call corresponding to the discarded back edge, we added a spill and reload for a possibly large set of variables. To determine the set of variables spilled, we examined the call DAG to find the procedures appearing on each path from the called procedure to the calling procedure. The locals of these procedures were spilled. On the assumption that the back-edge call really is recursive, these are the only locals that might be recursively re-instantiated.

Unfortunately, this isn't always correct. Figure 6 shows one of two possible depth-first-search DAGs for a counter-example program with five procedures. The solid lines are calls in the DAG; the dashed line from Q to P is a call not in the DAG. Procedure R has locals in r1 and r2, and procedure Q has a local in r1. This means that procedure P can keep a local in r3 (but not r2, because P's child R uses it). Procedure T, on the other hand, *is* allowed to keep a local in r2, because T's only child is Q. The recursive call from Q to P will archive all locals of both P and Q, namely r1 and r3 but not r2. Unfortunately, the series of calls shown as a dotted line leads to an error. Both T and R feel entitled to use r2, but r2 was not archived by the backward call to P. Thus the value in T's local can be damaged by R.

It was not hard to fix this problem. The better way to remove cycles from a directed graph is to partition the graph into its strongly-connected components* and then replace each component by a single node. Then we apply the bottom-up algorithm as before, with one

* Nodes *a* and *b* are in the same strongly-connected component of a directed graph if there is a (possibly empty) path from *a* to *b* and a (possibly empty) path from *b* to *a*. Thus a strongly-connected component of a call graph is either a maximal group of procedures that can form a recursive cycle, or a single procedure that can never be in a recursive cycle [3].

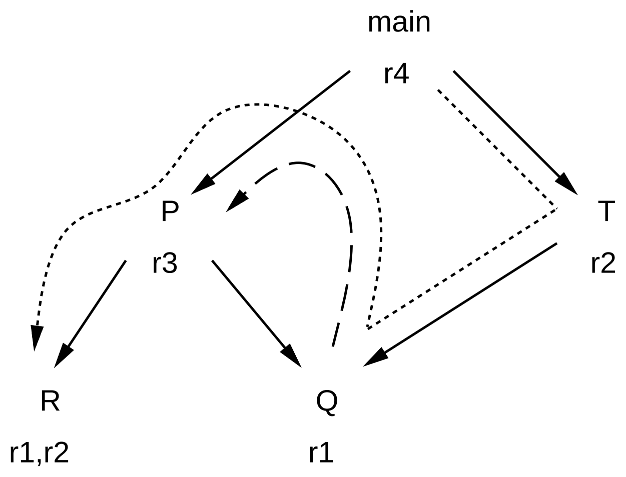

Figure 6. A depth-first-search call-DAG (solid) with a recursive call (dashed). The call chain shown (dotted) reveals a bug in our original algorithm for saving registers.

difference. When we reach a node representing an entire recursive cycle, we must pick pseudo-registers for all locals of all procedures in the cycle. We can use the same pseudo-register for locals of two different procedures in the cycle, but we must insert a spill and reload on each path between them. Our system does this simply by treating each procedure in the recursive component independently; it inserts spills and restores around any call between strongly-connected procedures. If recursion were more common, it might be worth concentrating the spills and reloads at the less frequent calls.

3.3.5. Fast argument passing

The scheme we have described so far deals with argument passing in a simple-minded way. We evaluate each argument and place it on the stack, in the memory location associated with the parameter that we are about to create. If the called procedure has a promoted parameter, it loads the argument from the stack into that register. It would be better if the calling procedure could put the argument directly into the parameter register to begin with.

We use the same idea here that we used in expression evaluation. The assembler generates code to evaluate the argument and store it on the stack. It flags this code with the usual RESULT, STORE, and REMOVE actions. These actions are not qualified by the name of the parameter, however, but rather by its position in the parameter list of the called procedure. (It cannot refer to the parameter by name because it generates the action at assembly time, when the called procedure may not be visible.) At link time, the module rewriter checks whether that parameter was promoted. If it was, the positional qualification is converted into a normal variable qualification.

We discussed the promotion of parameters to registers in section 3.3.3. In that scheme, the caller put each argument value on the stack. If we promote the parameter, we insert into the procedure entry code a load of the argument value into the parameter register. Now we must not do these loads, since the caller has already put the arguments directly into the parameter registers, and the value on the stack is garbage. Unfortunately, a caller still cannot do this if the call is indirect, because the identity of the called procedure is unknown. So we leave the inserted loads in place, and instead modify direct calls so that each enters the procedure after the point where we have inserted the loads.

3.3.6. Profiling

We do not keep all of the variables in registers, but only those that seem to be referenced frequently. As a result, the quality of the frequency estimates is important, especially in large programs. Instead of relying on estimates, we can use previously gathered execution-time profiles. Two kinds of profile are useful — a variable-use profile and a procedure-call profile.

A *variable-use profile* tells how many times each variable is loaded or stored. We can use these counts directly in place of the variable-use estimates that drive the selection of variables to promote. We built the variable-use profiler using the code modification facility in the Mahler linker. We will describe this in more detail in section 3.7.2.

The variable-use profile has the disadvantage that we may need to recompute it often for it to be effective. If the programmer adds a new variable to a procedure that is executed very frequently, that variable will not appear in the previously computed profile. It is therefore unlikely to get promoted. An alternative is to use *gprof* [17] to get a *procedure-call profile* telling us how many times each procedure was called. We can then combine these dynamic counts with the assembler's per-procedure estimates of variable use, which are computed every time the module is assembled. The resulting estimates will be less precise than a true variable-use profile, but also somewhat less sensitive to small changes in the program, and should be considerably more precise than estimates alone.

3.3.7. Intra-procedure coloring

In grouping together local variables into pseudo-registers, we assume very conservatively that a local variable is live whenever its procedure is active. This is a cheap way of estimating when locals of different procedures will conflict, and it is reasonable because the lifetimes of most procedures are short anyway. This simplification does have one important limitation: it does not allow us to combine two non-conflicting locals of the same procedure.

Recognizing two such locals does not require interprocedure dataflow analysis, only intra-procedure analysis and coloring. The assembler does a liveness analysis on each procedure it translates, grouping together locals that do not conflict. It does not allocate registers to these groups, but simply reports these groups to the link-time allocator.* The only tricky part is that the liveness analysis may allow a parameter to be combined with another local, or even with another parameter that for some reason is initially dead. If so, the register allocator must make sure that the live argument value gets properly loaded into the register that the two variables share.

* This analysis can be suppressed by the user, resulting in a faster compilation but producing code that is a few percent slower [40].

3.3.8. Alternative allocation strategies

Register allocation works by first deciding which variables and constants we should keep in registers and then rewriting the code to reflect this decision. These two phases are essentially independent. The code rewriting works in the same manner no matter how we make the selection of variables to promote.

This allows us, for example, to drive the selection by either frequency estimates or by a dynamic variable-use profile. It also allows us explicitly to exclude certain classes of variables from consideration, if we wish. In a multiprocessor system, for instance, we might wish to force global variables to reside in memory, because they are shared between processors with different register banks.

Another interesting notion is the distinction between cooperative allocation and selfish allocation. *Cooperative allocation* is the form described earlier. In cooperative allocation, a given procedure may not get all of its locals into registers. On the other hand, the allocation is based on the call graph, so spills and reloads around calls are not usually necessary. In *selfish allocation,* by contrast, each procedure is treated independently. It can keep all of its locals in registers if it wishes, but it must pay for this by saving and restoring these registers at entry and exit. In practice this means that it keeps a local in a register only if the local is used more than twice. The allocator uses frequency estimates or a dynamic profile to make that judgement.

3.3.9. Comparison with Steenkiste's allocation method

Steenkiste [35,36] independently developed an approach similar to ours. He does not do allocation in the linker and therefore has no need for annotations or module rewriting, but his algorithm for allocating registers to variables is much the same. The major difference is that Steenkiste does not use frequency information in allocating registers. Both schemes traverse the call graph starting with the leaves, in order to combine local variables into groups. Our scheme allocates registers to the most important of these groups. Steenkiste's scheme allocates registers to the groups sufficiently close to the leaves. Whenever it runs out of registers, Steenkiste's scheme reverts to doing saves and restores. Thus procedures near the leaves use cooperative allocation, and procedures far from the leaves use selfish allocation. Globals do not fit well into Steenkiste's scheme, which can hamper performance.

3.3.10. Intermodule register allocation: Summary

The Mahler assembler prepares for intermodule register allocation by annotating the code with register actions and providing intra-procedure estimates of scalar use frequency. The Mahler linker combines these estimates to obtain estimates for the entire run, groups together non-conflicting locals based on the program call graph, and allocates registers for the most important groups. It then rewrites the code to reflect this choice, removing loads and stores and modifying other instructions. As we will see in section 4.1, between 60% and 80% of the references to scalars are removed in this manner.

Once register allocation is finished, we can schedule the resulting instructions for better pipeline performance, the subject of the next section.

3.4. Pipeline instruction scheduling

Our second link-time optimizing transformation is the pipeline instruction scheduler, originally written by Michael L. Powell [44]. The instruction scheduler finds the places where cycles will be wasted. Wasted cycles can happen because of a pipeline stall in accessing memory, a stall waiting for the coprocessor to finish, or a no-op after a branch. It then tries to move instructions around so that these wasted cycles are filled with productive instructions. Because these productive instructions would otherwise require cycles of their own, the program gets faster.

The instruction scheduler works in two phases. First, it reorders each basic block independently, trying to fill stalls and branch slots from within the same basic block. Then it makes a final pass looking for branch slots that have not been filled. It tries to fill each unfilled slot with a useful or potentially useful instruction from the next block to be executed.

3.4.1. Scheduling a basic block

We say that two instructions *conflict* if they cannot be exchanged even when they are adjacent. This is true only if one of them modifies a register or memory location that the other instruction also uses or modifies. Usually it is impossible to tell if two memory references are to the same location, so we must assume the worst and suppose that they are. The one exception is when the two references use the same base register and different displacements. These two references do not conflict even though an intervening instruction might change the value of the base register, because the intervening instruction will conflict with both memory references and the order of the three will be preserved.

The scheduler starts by computing the earliest time at which each instruction can be executed relative to the beginning of its basic block. This earliest time is determined by looking for conflicts with all preceding instructions. In the same way, the scheduler computes the latest time at which each instruction can be executed, relative to the end of the block, by looking for conflicts with instructions that follow it. The instructions in the block (except for the final branch, if present, and its branch slot) are then sorted according to latest start time. Ties are broken according to the earliest start time.

Next, another pass looks for instructions that stall because of an interaction with the previous instruction. The scheduler delays such an instruction by pushing it later in the instruction sequence. This instruction can be pushed past any instruction whose earliest allowed time precedes this instruction's.

At this point, the scheduler has filled as many of the memory stalls and coprocessor stalls in the basic block as it can.[*] If the block ends with an empty branch slot, the scheduler looks at the instruction that has been sorted to the position before the branch. If this instruction can be performed later than the branch itself, then it can be moved into the branch slot to replace the no-op there.

[*] The problem of finding an optimal schedule is unfortunately NP-complete, but more aggressive heuristics than ours have been explored [20].

3.4.2. Filling slots with future instructions

When the scheduler reorganizes a basic block, it may fill the branch slot with an instruction from earlier in the basic block. When it has reorganized all of the blocks, however, some branch slots may still be unfilled. The possibility remains of looking in the block that will be executed *after* a branch, and moving an instruction from there into the branch slot. A final pass over the object code tries to do this, by looking at the destination of each such branch.

If a branch jumps to an unconditional branch, we can sometimes change the first branch so it bypasses the second, by jumping directly to the same place. The only problem is that the second branch's delay slot may contain an instruction that must be executed. If the first branch is unconditional, we can guarantee that this instruction gets executed anyway by copying it into the slot of the first branch. We cannot bypass the second branch if both have non-empty branch slots, or if both are conditional.

If an unconditional branch jumps to a nonbranch, we copy the nonbranch target into the branch slot. Then we change the branch so that it jumps to the instruction after the nonbranch. This also works if the destination is a conditional branch, because of the way delayed branches work.

The most interesting case is when a conditional branch jumps to a nonbranch. We may be able to fill the slot *speculatively,* either with an instruction from the block at the branch destination or with one from the block that we get to by falling through the branch. This is safe only if the instruction is harmless even when we end up in the wrong block; it is profitable whenever we end up in the right one.

Looking at one of the two blocks, we can pick a candidate instruction from among those instructions whose earliest time is 0; these can be executed first and so could just as well be executed in the preceding branch slot. A candidate instruction from one block is safe if the register it modifies is dead at the beginning of the other block. Dataflow analysis would give us precise information about this, but we can do pretty well just by looking to see if the other block sets that register without using it first.

If we pick the instruction from the fall-through block, we must delete it from that block. Otherwise it will be executed twice: once in the branch slot and once in the block that follows. We therefore consider using the fall-through block only if it cannot be branched to from elsewhere in the program.

If we pick the instruction from the destination block, we move it to the beginning of that block, copy it into the branch slot, and increment the destination of the branch so that it skips over the original. This leaves the destination block intact in case there is another branch to it. We are not allowed to do this a second time to fill some other branch slot, however, so we mark the instruction in the destination block as immovable.

In Figure 7(a), the only safe candidates are the loads into r1. It is worth noting that they make each other safe: moving either one into the branch slot is safe because the other one shows us that r1 is dead at the start of its block. The loads into r2 or r3 cannot be guaranteed safe, and the arithmetic operations and stores cannot even be candidates because their start times are greater than zero. Figures 7(b) and 7(c) show what happens if we select each of the two safe candidates.

```
       (a) original code        (b) fill from               (c) fill from
                                   destination                 fall-through
       if r4 = 0 goto L1        if r4 = 0 goto L1+1         if r4 = 0 goto L1
       no-op                    r1 := load v                r1 := load x

       r1 := load x             r1 := load x
       r2 := load y             r2 := load y                r2 := load y
       r1 := r1 + r2            r1 := r1 + r2               r1 := r1 + r2
       goto L2                  goto L2                     goto L2
       store z := r1            store z := r1               store z := r1

   L1: r3 := load u         L1: r1 := load v            L1: r3 := load u
       r1 := load v             r3 := load u                r1 := load v
       r3 := r3 - r1            r3 := r3 - r1               r3 := r3 - r1
       store z := r3            store z := r3               store z := r3
   L2:                      L2:                         L2:
```

Figure 7. Filling a conditional branch slot with instructions from each possible successor block.

If the branch is a backward branch, it is probably a loop and will therefore most likely be taken. We therefore prefer to use the destination block rather than the fall-through block. But a speculatively executed instruction in a branch slot is no worse than a no-op; looking in both blocks seems sensible.

3.5. Cooperation between assembler and linker

We have already seen that the assembler provides information that the register allocator uses to select register variables and rewrite the code. We have also seen that the assembler marks tricky idioms so that the scheduler will not hurt them. There are other forms of cooperation as well.

One important example is our decision to do scheduling after global register allocation. The instruction scheduler never crosses module boundaries and might therefore be part of the assembler rather than the linker. Doing scheduling before register allocation doesn't make much sense, however, because register allocation causes profound changes in the structure of a basic block. To begin with, register allocation removes many loads and stores, which are common sources of stalls. There would be little point in reorganizing the code to avoid a load stall if the load itself later disappears. Furthermore, by using more of the registers, register allocation can remove irrelevant conflicts between instructions. If the sequence

```
r1 := load x
r1 := r1 + 1
store x := r1
```

appears in the middle of a block, other unrelated uses of r1 cannot be moved past it in either direction. If *x* is promoted, however, the increment of *x* no longer uses r1, and other instructions become more mobile.

The assembler helps the scheduler by allocating the dedicated temporary registers in round-robin order. When we are finished using r1, we will usually not allocate it for a new purpose until several instructions later. Thus even a block without many promoted variables will tend to have more mobile instructions than it would if it used the same one or two temporaries over and over.

This round-robin allocation of temporary registers might hamper the effectiveness of the inter-block phase of scheduling, because it makes it harder to be sure that a particular register is dead at the beginning of the block. The assembler alleviates this somewhat by starting the cycle over again each time a new basic block begins. Thus the first temporary register to be assigned by a block is more likely to be r1 than any other, so it is more likely that the two blocks will both have such assignments. In that case either one could be moved into the branch slot.

3.6. Loopholes

Sometimes you really do need precise control over the machine. You might need to write absolutely the best possible version of a library routine that will be executed millions of times, like input/output conversion. Or you might need to write a routine that is responsible for directly maintaining part of the machine state, like an interrupt handler. Mahler has two mechanisms for getting this kind of fine control over the machine: builtin procedures, and startup procedures.

The Mahler assembler recognizes the names of a small set of builtin procedures. The name of the builtin procedure is a Mahler keyword and cannot be redefined by the user. When the assembly language contains a call to a builtin procedure, the Mahler assembler translates it to inline code instead of calling code. Builtin procedures are used to implement operations that cannot be expressed (or expressed efficiently enough) in normal Mahler. A call to a builtin procedure looks just like a normal call, so it can be made from a high-level language as well as from a Mahler routine.

We have tried to design Mahler builtins with semantics that are as clean and high-level as practical. They should satisfy the user, but they should also let Mahler know what is happening so that its analysis will not be impaired. Thus we have shied away, for example, from builtins that simply let the user examine or change a specific register. Instead, we have tried to find out what problem the user was really trying to solve, so we could design a builtin to suit. Dangerous lower-level builtins are occasionally useful to let the user decide what is really needed. Nonetheless, they are used rarely; we have always tried to replace them with more meaningful builtins when we understand the problem better.

For example, Mahler has builtin procedures to spill and reload big pieces of the register state for use in interrupt handling. It has builtin procedures to access certain extended-precision arithmetic instructions, for use by the input/output conversion routines. These routines would likely have to be changed for a new generation of Titan, and perhaps it would be better to make the entire conversion routine be builtin. Even the C library routines of *setjmp, longjmp,* and *alloca* are Mahler builtins, because Mahler must know when these things happen if it is to optimize data references safely.

Adding a new builtin procedure to Mahler is straightforward, but it must be done by the Mahler implementor: the user cannot specify arbitrary machine-level code. We have found this mechanism adequate. Machine-level control of the registers and pipeline is handled adequately by the Mahler system itself, and machine-level instruction selection is for the most part expressible in the Mahler language — where it is necessary at all. The Titan is a simple machine, and most of its user operations are directly accessible from a high-level language.

The other Mahler loophole mechanism is the startup procedure. A startup procedure has no name and is never called by another routine. To give this routine control some piece of magic is needed, either by hardware intervention (for operating system bootstrap code) or by operating system activity (for user code that begins a program execution). A startup procedure has no local variables and must ensure that a valid data environment exists before it can call a normal procedure or reference a global variable; the startup procedure can invoke builtins to do these setup tasks inline.

These two extensions were enough to avoid the use of any lower-level assembly language, even in the operating system. A UNIX operating system was experimentally ported to the Titan using only high-level languages and Mahler. Even the less pure port now in use includes only 1000 instructions of ‘‘true’’ assembly code, for bootstrapping and interrupt handling. This code is not reachable by normal means, and the normal procedures that it calls are declared outside as procedure variables, so no change to Mahler's analysis is necessary.

3.7. Instrumentation

Mahler produces all of the code in any program. It also makes extensive changes to the code at link time. This makes it easy for the linker to insert instrumentation code of various kinds. We have taken advantage of this for a variety of different applications, both high-level and low-level, and both special-purpose and general-purpose [42].

A similar facility is provided by the *pixie* tool developed by Earl Killian at MIPS Computer Systems [28]. Pixie was developed independently from our system but the possible modifications overlap with ours; its approach is in some ways similar to ours. The pixie system works by transforming a fully-linked executable instead of by transforming object files being linked. Thus it transforms the code *after* linking instead of before. This has two consequences. On one hand, their system is easier to use than ours. One must know only the name of the executable, rather than how to link its components. On the other hand, our approach is easier to implement. An isolated object file, prior to linking, contains the relocation table and loader symbol table, which make the transformation easier. Lacking this information, pixie must postpone much address translation until the modified program is executed, because it cannot reliably distinguish text addresses from data values at transformation time [25]. This introduces runtime overhead that makes pixie an unsuitable vehicle for optimization, though it has been used for several of the instrumentation purposes that Mahler has.

3.7.1. Instruction-level instrumentation

Our first instrumentation tool was an instruction-level profiler. We create a set of global counters in the instrumented program, one for each kind of event we want to count. We then insert code to increment the associated counter wherever that event appears. If we insert an increment wherever a load appears, for example, running the program will count the loads performed. This lets us count occurrences of different kinds of instructions or recognizable multi-instruction idioms like byte load or procedure call.

We can also count pipeline stalls that occur completely within a single basic block, by examining each block and inserting an increment for the right amount. This requires only the same analysis that the scheduler itself must do. Because stalls sometimes occur across a block boundary, we also create some global state variables. When we see a block with an operation whose latency extends past the end of the block, we insert code to set the state variable, and when we see a block with an instruction that could be delayed because of the state, we insert code to check the state variable and conditionally increment the stall counter.

Instruction-level instrumentation provides us with statistics that one often acquires only 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. 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.

3.7.2. Profiling

The *gprof* [17] profiler is a useful tool for performance debugging of applications. The normal way to use it 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. Running the resulting instrumented program produces a file of profile data.

The only effect of compiling with *–pg* is to insert a lot of calls to a special routine named *mcount,* which is responsible for keeping track of who calls whom. These calls are inserted at the beginning of each procedure, and it's easy to make the Mahler linker do that itself.

The advantages are considerable. Relinking is one or two orders of magnitude faster than recompiling all modules, and we no longer need to maintain the instrumented versions of libraries. Profiling is now much more attractive.

Just as we can insert profile code at the beginning of each procedure, we can insert code to count the executions of each individual basic block. The linker allocates a long vector of counts, one for each basic block in the program. The inserted code for a block increments the count associated with that block.

Our main use of the basic block execution counts is the construction of a *variable-use profile.* In section 3.7.2 we observed that the register allocation can be improved by using actual variable reference counts rather than compile-time estimates. To build the variable-use profile, we combine the basic block counts with static information 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 to determine: each variable reference counted is a load or store marked for removal if the variable is promoted. Combining these per-block statistics with the dynamic block counts gives us a dynamic profile of variable use.

3.7.3. Register management strategies

In 1988, we reported the results of a study comparing register allocation techniques and hardware register windows [41]. Some machines [23,24,29,33] treat the register set as a flat space and rely on the language system to promote the variables. Other machines [1,5,16,31] include hardware to divide the register set into a 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.

Without promotions, we would need to perform some number M_0 of loads and stores to access scalar variables and constants. Promoting some of these to registers reduces this number to M_1. (In either case, of course, we would need loads and stores to access array or pointer structures. These are not included in M_0 or M_1.) Unfortunately, any scheme for promotion also requires that we add some number S of new loads and stores to spill and reload promoted variables. With register allocation, the new loads and stores are needed when one makes procedure calls that would otherwise harm registers in use. This might be all procedure calls or, in our scheme, only recursive and indirect calls. With register windows, the new loads and stores are needed when one runs out of buffers after a long series of calls or returns. When this happens, an exception occurs that is handled by spilling or reloading windows. Although these loads and stores do not appear in the user program, it seems only fair to charge for them.

We can take both effects into account by computing a *miss ratio* $(M_1+S)/M_0$. The miss ratio is a measure of how thoroughly we were able to remove loads and stores associated with scalars. If the miss ratio is zero, we have managed to keep all variables and constants in registers, and did not need to perform any spills and reloads. If the miss ratio is more than unity, then the spills and reloads slowed the program down more than keeping things in registers speeded it up. The miss ratio is a property of the dynamic behavior of a program. A particular load or store instruction may be counted many times, or no times, depending on how often control passes through it.

We used our automatic instrumentation to compute the miss ratio for a variety of hardware and software promotion schemes. This usually required more than simply instrumenting the executed code. For register allocation, we inserted code to count not the loads and stores that were actually executed, but those that would have been executed with a given allocation scheme and a given number of registers. We similarly counted the times we passed a place where a load or store would have been *removed.* For hardware windows, the saves and restores depend on the history of recent calls and returns and on how deep the call chain is. We inserted instrumentation code to keep track of this and to simulate the state of the circular window buffer. This let us count hypothetical saves and restores at the times when a machine with windows would actually overflow or underflow. Some of the results will be sketched in section 4.1.

3.7.4. Address traces

An address trace is the sequence of instruction 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 is to physically monitor the address bus, using either a hardware monitor [13] or extra microcode [2], and log the addresses seen. This approach can slow execution by an order of magnitude, and in any case is not suitable for modern RISC machines, which are integrated on a single chip and have no microcode. 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 is to use link-time code modification to instrument the code. Wherever a data memory reference appears, the linker inserts a very short, stylized subroutine 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. Borg [8] provided this facility with operating system kernel support that lets the trace buffer be shared by several multiprogrammed processes and even the kernel itself. Each time the buffer fills up, a special untraced process is resumed to consume it, either dumping it to an output device or, more usefully, incrementally continuing a parameterized cache simulation.

4. Performance of register and pipeline management

How well does Mahler do at managing the registers and pipeline? By instrumenting the Mahler implementation and the code it produces, we were able to produce some answers to this question.

We performed the measurements described in this section using five artificial benchmarks and ten real programs in use at WRL. These fifteen programs are summarized in Figure 8. *Sed, egrep, troff, yacc, rsim,* and *mx* are written in C, and the Boyer benchmark is written in Scheme, a dialect of Lisp; the rest are written in Modula-2.

	lines	*vars*	*procs*	*remarks*
Livermore	268	347	20	Livermore loops
Whetstones	462	413	37	Floating-point
Linpack	814	397	31	Linear algebra [14]
Stanford	1019	565	65	Hennessy's suite [18]
Boyer	600	1702	98	Lisp theorem-proving [15]
sed	1751	259	31	Stream editor
egrep	844	241	28	File search
troff	7577	947	175	Text formatter
yacc	1856	689	73	Compiler-compiler
metronome	4287	1240	157	Timing verifier
rsim	3003	911	133	Logic simulator [39]
grr	5883	1833	243	Printed circuit board router
eco	2721	1008	150	Recursive tree comparison
ccom	10142	2193	333	C compiler front end
mx	26449	2869	354	Mouse-based editor [30]

Figure 8. The fifteen test programs.

4.1. Performance of intermodule register allocation

The aim of intermodule register allocation is to promote important variables and constants to registers, so as to avoid memory references. The miss ratio, defined in section 3.7.3, is a measure of how effective we are at this. We used automatic instrumentation to compute the miss ratio of our register allocation technique assuming a range of possible register set sizes.

Figure 9 shows the miss ratios for each of the fifteen test programs, plotted as a function of the number of registers available for global register allocation. The left-hand graph shows the effects of register allocation based only on the compile-time estimates of variable use frequency. The right-hand graph shows the effects based on an actual variable-frequency profile for the same run. In this graph, six of the test programs had miss ratios very close to zero; these are labelled with numbers rather than names. These graphs extend beyond the actual limits of the Titan, in which 52 registers are available for global register allocation. The dotted curve is the arithmetic mean of the miss ratios for all fifteen programs.

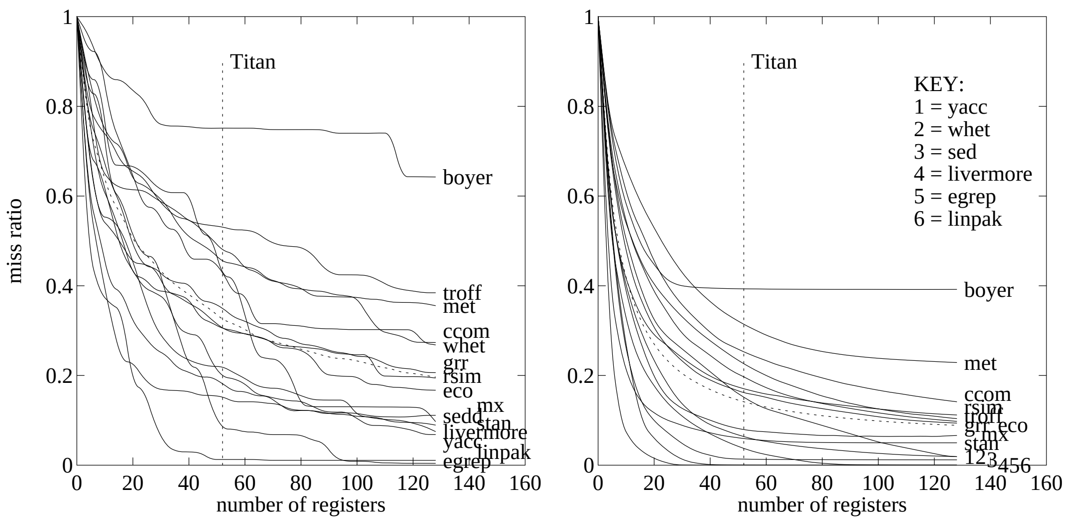

Figure 9. Miss ratio as a function of registers allocated, for allocation using estimates (left) or dynamic profile (right).

There is considerable variation across the test suite, but it is not obviously correlated with the size of the program. The mx program is the largest, and moreover is written [30] in ‘‘object-oriented’’ style with frequent use of procedure variables, but it nevertheless does considerably better than average. Even with relatively few registers, most programs can make half or more of their scalar references from registers. On the Titan the average is around two-thirds.

Use of a profile not only improves the curves, it also smooths them: each time a new register is available, the allocator makes the best choice of what to do with it. Each profile used here was for identical inputs to the program in question, so the curves on the right represent the best possible behavior.* Unsurprisingly, the profile makes the biggest difference when few registers are available.

As discussed in section 3.7.3, we computed the miss ratio curves for several register management techniques. Figure 10 shows the average miss ratio for each scheme, as a function of number of registers used. The left-hand graph shows the average for the five artificial benchmarks, and the right-hand graph shows it for the ten real programs. We examined six different schemes. Each scheme was tested twice, basing its decisions first on compile-time variable-frequency estimates, and then on an actual variable-frequency profile. The labels on Figure 10’s curves are defined below; a ‘‘P’’ indicates that the scheme used a dynamic variable-use profile rather than estimates.

Link-time allocation (L, LP) is our technique, described in section 3.3.† It starts by keeping all scalars in memory except within a basic block, but improves this at link time based on inter-module analysis. It builds a complete call graph and an estimate of how often each scalar is

* This is optimistic, but perhaps not grossly so [43].

† The miss ratios for each benchmark under schemes L and LP were shown in Figure 9.

used. Locals are combined into non-conflicting groups based on the call graph, and the most frequently used globals and groups of locals are promoted to registers. Little-used scalars continue to live in memory, and are brought into temporary registers whenever they are needed.

Compile-time allocation (C, CP) keeps each procedure's locals in registers but saves and restores those registers on entry and exit so that this procedure does not interfere with others. Globals are not kept in registers, as this requires interprocedure analysis to be safe and effective.

Steenkiste allocation (S, SP) is an improvement of compile-time allocation. Steenkiste allocation starts with a compile-time allocation, in which each procedure uses essentially the same set of registers. Then it builds a complete call graph, and locals of procedures near the leaves are renamed to different registers so that saves and restores are not needed. The decision of which locals to rename is determined by the structure of the call graph and not by use frequency. Globals are not kept in registers.

Hybrid allocation (H, HP) is a combination of our link-time allocation and Steenkiste allocation. It starts with a compile-time allocation with saves and restores in each procedure. A second phase then tries to use the remaining registers to rename locals so as to remove saves and restores, or to hold globals. This second phase is based on the estimated savings as well as on the structure of the call graph. As with both our link-time allocation and Steenkiste allocation, using the call graph allows the same register to be used for several non-conflicting locals.

Fixed-sized windows (W, WP) assumes that the register set is divided into overlapping hardware windows in a circular buffer. The buffer shifts at each call, allowing eight new registers to be used. When a long chain of calls or returns causes the buffer to overflow or underflow, exactly eight registers are saved or restored to make room for the required window. Although these saves and restores are done by trap code, we assume that the trap has absolutely no overhead, and charge only for the loads and stores that occur. This is optimistic; real window machines have a significant trap overhead, and try to minimize it by saving and restoring several windows at a time. In that case, more may be saved or restored than is necessary.

Variable-sized windows (V, VP) assumes that the register set is used as a circular buffer of variable-sized windows. At each call, the buffer shifts for the number of registers the procedure needs for its locals. When the buffer overflows or underflows, exactly enough registers are saved or restored to make room for the required window. Again, we assume that the window traps have no overhead, and we therefore charge only for the loads and stores.

In Figure 10, if we look only at the five artificial benchmarks, we might conclude that the link-time schemes L and H were clear winners over the others. The results of the ten real programs suggest that the advantage is not so dramatic. Compile-time allocation, Steenkiste allocation, and windows have smaller average miss ratios for the real programs than they do for the artificial benchmarks. This merely shows that artificial benchmarks are unrealistic. Modern allocation techniques aim to exploit the locality of procedures. Unfortunately, artificial benchmarks often have unrealistically small procedures and an unrealistically high ratio of global references to local references.

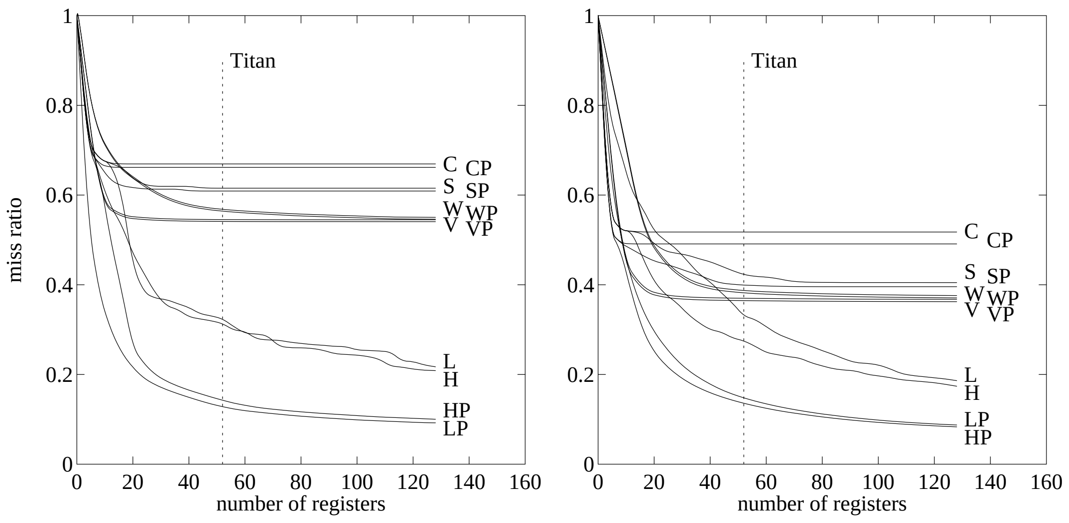

Figure 10. Miss ratio as a function of registers allocated, for each register management scheme, averaged over the five artificial benchmarks (left) or the ten real programs (right).

For most of our range of study, variable-sized windows don't do noticeably better than fixed-sized windows. On the Titan, with 52 registers available, windows and link-time allocation do about equally well. The variable-window curve flattens out almost immediately, suggesting that windows would have the biggest advantage over allocation when only a few registers are available. We should remember, though, that the miss ratio for windows does not include the overhead of the trap for window overflow or underflow. Its apparent advantage for small register sets may be spurious.

A profile had little effect on schemes not designed to exploit a profile, like Steenkiste's scheme. Nevertheless, a profile was occasionally of use even to these schemes. It revealed variables that were never used at all and that therefore did not need a register.

Having implemented both Steenkiste's scheme and our own, the hybrid scheme was an easy generalization. It turns out to be reasonably successful. It combines the best of Steenkiste's scheme (complete coverage of locals, at the smaller expense of saves and restores) with the best of our link-time scheme (inclusion of globals and use of frequency information). Although it has little to recommend it over our normal scheme when applied to the artificial benchmarks, its performance on the real programs is good. It stays consistently ahead of our scheme, and takes an early lead that makes it suitable even when relatively few registers are available, giving it a clear advantage over both its parents.

When we first made the comparison, the hybrid scheme did several percent worse than our original scheme. The reason was interesting. In our standard libraries, several routines make calls through procedure variables, even when it is not necessary. The *exit* routine is one such, as are all routines that set up arguments for system calls. In either Steenkiste allocation or the hybrid scheme, these routines, along with any routines above them in the call graph, are rendered ineligible for renaming of local registers. This is overkill, however. We never return from the *exit* routine, and so it would be safe to rename locals above it on the call graph. Similarly, the routine that performs a system-call trap is short and stylized, and poses no threat to renamed registers above it. The designers of a system using this sort of register allocation would probably tune their libraries so that they did not unnecessarily handicap the allocator. Alternatively, they might build in knowledge of these exceptions so that the allocator could work around them. The latter would be more dangerous but might be easier. We assume that the problem would be addressed in some manner, and wanted only to get realistic measurements of the Steenkiste and hybrid schemes. We therefore adopted the latter approach.

The link-time allocation schemes (L and H) are clear winners asymptotically, though it must be admitted that without a profile other schemes work better with small register sets. As future machines exhibit higher levels of integration and more instruction-level parallelism, we are likely to see an increase in the size of register sets. This will make the advantages of link-time allocation schemes even more important.

The better performance of the link-time schemes comes from two advantages over the other schemes. First, the link-time analysis makes it safe to include globals without risk of aliasing errors. Second, the use of variable-frequency estimates (or profiles) lets them use their registers preferentially for important variables. By including this kind of interprocedural and intermodule analysis, we could improve the window schemes enough to give them comparable miss ratios [41]. Still, if we are willing to do the intermodule analysis, it is unclear why we should want the windows.

4.2. Performance of pipeline instruction scheduling

Our other link-time optimization is pipeline scheduling. The Titan CPU can waste cycles because of a pipeline stall or an unfilled branch slot. The scheduler hides these CPU-wasted cycles by moving other instructions into them, in an intra-block phase followed by a modest inter-block phase, as described in section 3.4. Because the inter-block phase fills branch slots speculatively from the likely successor block, we must be careful how we count CPU-wasted cycles. A branch slot filled with an instruction from the destination block, for example, is useful whenever the branch is actually taken, and is wasted only when the branch is *not* taken; the reverse is true if we fill the slot from the fall-through block.

We used Mahler's instruction-level instrumentation to count useful and wasted cycles in unscheduled code, in code with only intra-block scheduling, and in code with full scheduling.[*] The results are shown in Figure 11. CPU-wasted cycles in unscheduled code made up between 20 and 60 percent of all cycles. We normalized the CPU-wasted cycles for each benchmark so that the wasted cycles for unscheduled code is unity; thus the fraction removed is zero. In the left-hand graph, global register allocation was done; in the right-hand graph it was not. The dotted curves are the arithmetic means over all 15 programs.

In most cases the second phase of scheduling was less effective than the first, because the second phase is trying only to fill branch slots. Two of the programs, however, improved proportionally more in the second phase than in the first. The spread of improvements was about

[*] This does not count wasted cycles due to effects outside the processor, such as cache misses or page faults.

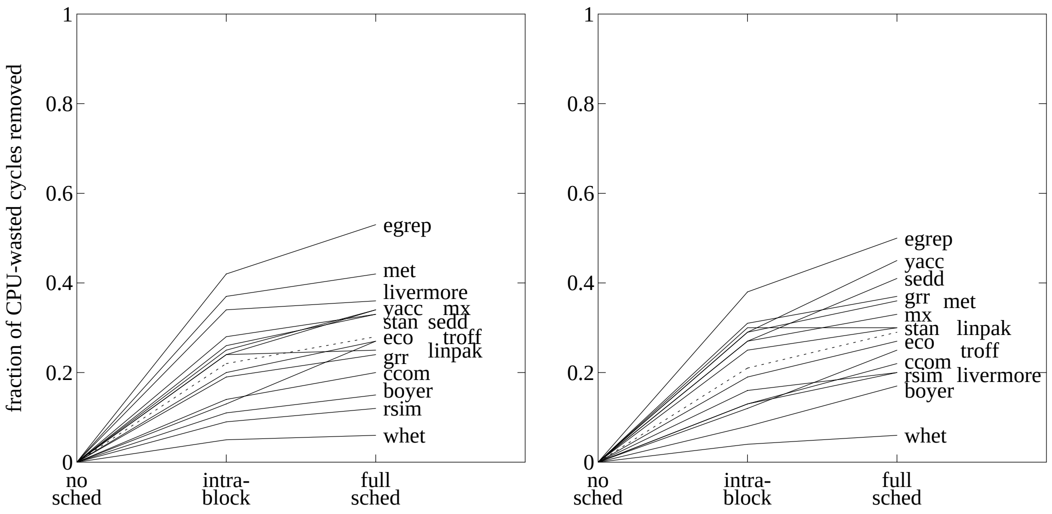

Figure 11. Fraction of all CPU-wasted cycles that are removed by different scheduling intensities, with global register allocation (left) and without (right).

the same with and without register allocation. On the average, scheduling removed about one fourth of the CPU-wasted cycles.

Also of interest is how well the scheduler can fill branch slots with useful instructions. As before, we must consider how often a branch slot is usefully executed rather than how often it contains an instruction other than a no-op. The results are summarized in Figure 12, which shows how often various kinds of branch slots contained useful instructions when executed, averaged over all the test programs.*

The certainty of an unconditional jump allows us to fill the slot almost all the time. In fact, the only exceptions are certain branches whose destinations are unknown, as when we are jumping into a table.

Because a backward conditional branch is likely to be a loop, we try to fill its slot from the destination block in preference to the fall-through block. The result is that the slot after a backward conditional branch is likely to be usefully filled around two-thirds of the time.

Figure 12 shows that global register allocation has little effect on either intra-block scheduling or inter-block filling of unconditional branch slots. Unfortunately, it does significantly reduce our ability to speculatively fill the slot of a conditional branch in the inter-block phase.

This is because register allocation broadens the set of registers referenced. In code without register allocation, the first instruction of a block is likely to be a load of some variable into the first temporary register. It is likely that any pair of possible successor blocks will

* The fraction of useful branch slots is not zero even without scheduling, because the Mahler assembler fills in a few branch slots itself, within idioms (see section 3.1).

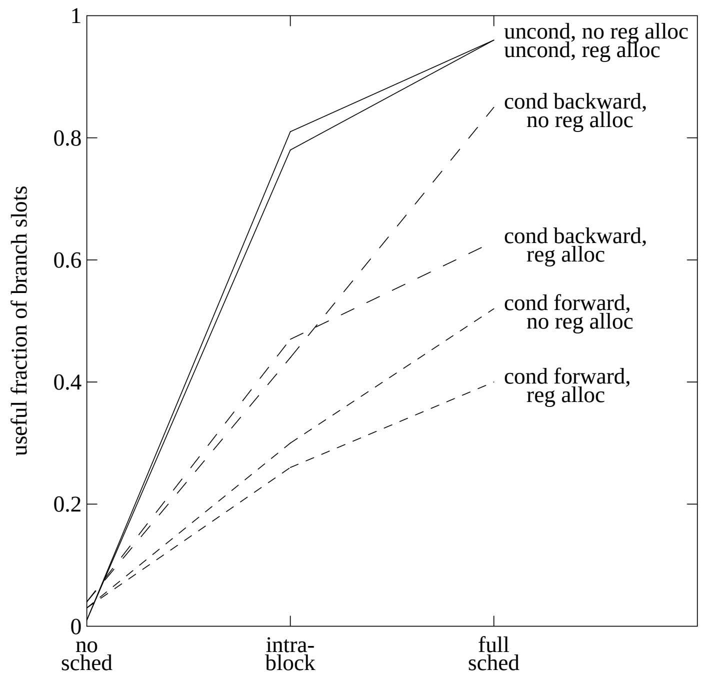

Figure 12. Fraction of branch slots that are useful for three kinds of branches, with and without register allocation.

both start with such a load instruction. In that case, the scheduler will know that either one can safely be moved into the preceding branch slot. After register allocation, however, these loads are likely to be absent; the first instruction is probably an operation using the register allocated for the variable that was loaded in the original code. The variables used by the two successor blocks will not necessarily be the same. The scheduler will therefore have more trouble guaranteeing that a candidate instruction will be harmless if control goes the other way.

Although register allocation degrades the filling of forward branches and backward branches by comparable proportions on the average, there is an interesting asymmetry between the two. Figure 13 shows the effect of register allocation on the fraction of usefully filled branch slots after scheduling. The left-hand graph shows forward conditional branches, and the right-hand graph shows backward conditional branches. Each graph shows the fraction of slots that are useful in each of the fifteen test programs, first when register allocation is not done and then when it is done. The range of fractions without register allocation is about the same in either case. The effect of register allocation on the two cases is quite different, however. Register allocation degraded the filling of forward branch slots by degrading most of the programs about the same amount. In contrast, it degraded the filling of backward branch slots by spreading out the distribution. Some programs had many backward branch slots filled, with or without register allocation. Others did much worse under register allocation. The variation does not seem to be related to differences in the relative abundance of forward and backward branches. Why this asymmetry exists is still unclear.

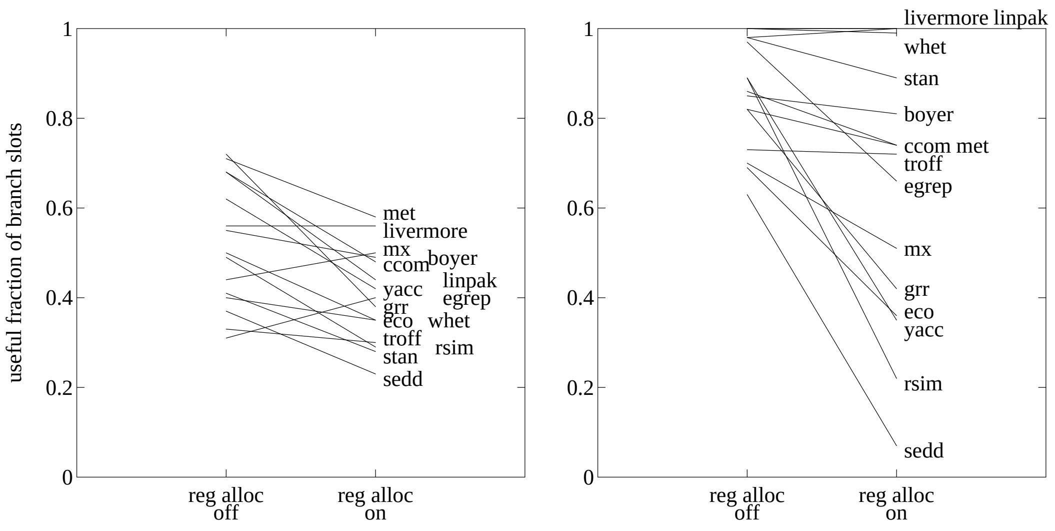

Figure 13. Effect of register allocation on the dynamic fraction of useful branch slots after scheduling, for forward conditional branches (left) and backward conditional branches (right).

5. A retrospective assessment

Performing optimizations at link time is not free. The register actions increase the size of an object file, typically by a factor of 2. Register allocation by itself slows down the link step by a factor of 1.5 to 3. Asking for both register allocation and scheduling slows it down by a factor of 2 to 4. We had hoped it would be less than this; slowdowns of 2 to 4 are enough to keep some people from using link-time optimizations routinely. Because our register allocation exploits intermodule information, an alternative would be to require complete recompilation, so that the intermodule information can be collected. A complete recompilation takes anywhere from 5 to 50 times as long as a non-optimizing link, not including recompiling the relevant library modules. An optimizing link is clearly preferable.

Whether people at WRL are willing to accept the slowdown of optimizing at link time depends on how fast our programs run after optimization. Figure 14 shows the runtimes of the fifteen test programs for various combinations of register allocation and scheduling, relative to the runtimes when neither was done. Even without profiling, the average payoff in using both is a 25% improvement in speed.

What would we do differently? Our biggest mistake, despite our attempts to avoid it, was that we pushed too much into the linker. The Mahler assembler should have done intra-procedure promotion of variables, inserting saves and restores that the link-time allocator would remove. This would have led directly to the hybrid scheme described in section 4.1, making the link-time part of the algorithm both more effective and faster. With fewer memory references making it past the assembler, we might even have been able to do pipeline scheduling in the assembler rather than the linker.

We were also disappointed in the low appeal of the profile-driven allocation. Even though using profile-driven allocation made programs 10% to 15% faster than using normal

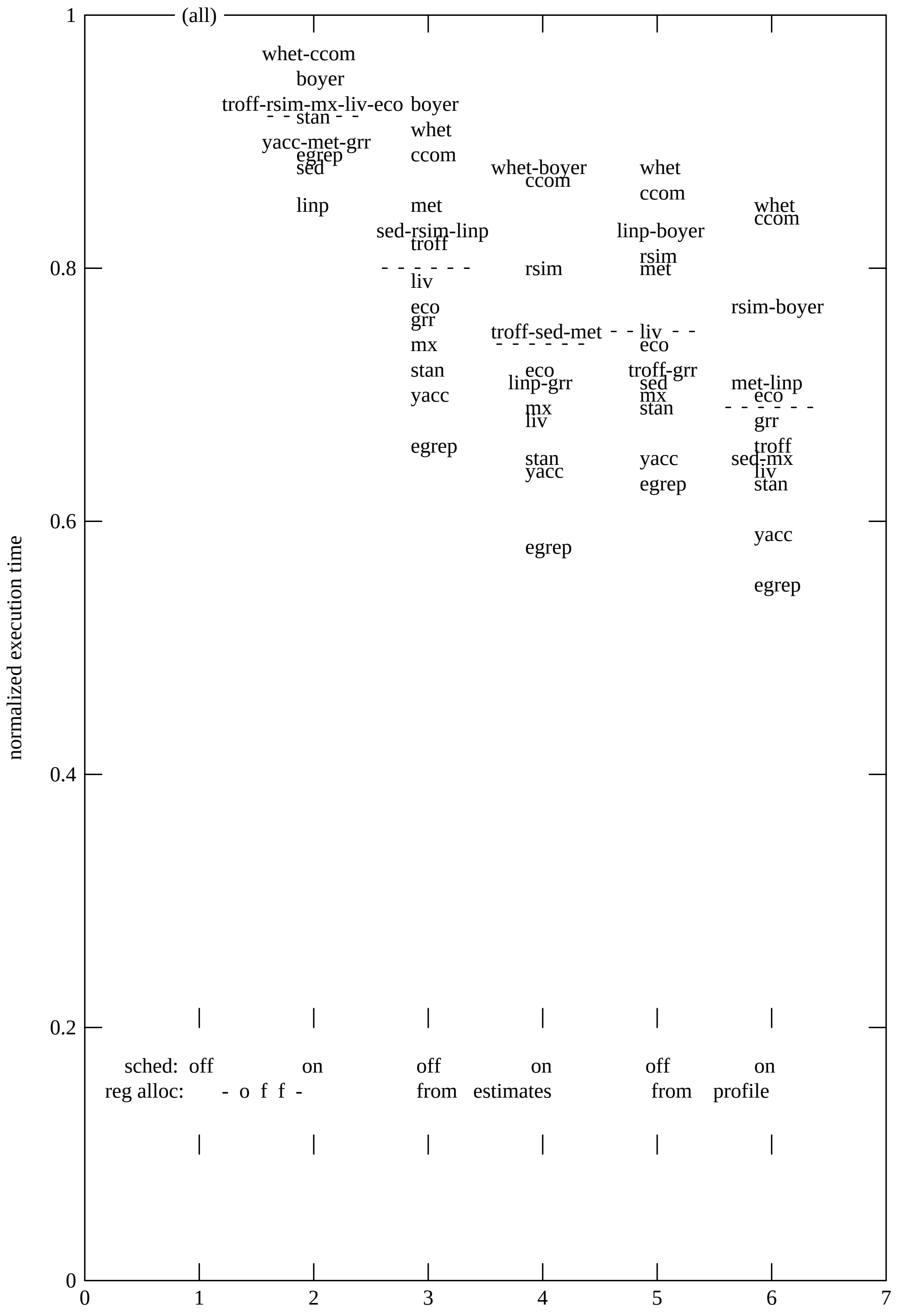

Figure 14. Execution times for various optimization combinations, normalized to the case of no link-time optimization.

link-time allocation, few people went to the trouble. In retrospect, this is not terribly surprising. The programs that one wants to squeeze the last bit of performance from are usually the programs that run a long time, which means they are not programs one wants to run as part of the build step. Perhaps if a profile had offered 50% or 90% improvement this would be different, but our experience may serve as a warning to those exploring profile-driven optimization.

We did not attempt to make the debugger understand link-time optimized programs. This was due mainly to a lack of people. Register allocation should not have been a significant problem since a variable is either promoted for its entire lifetime or it is not promoted at all. Pipeline scheduling can rearrange instructions enough to make statement-oriented breakpoints difficult, but there is now a fairly well-understood tradeoff between code motion and debugging [19]. It is interesting to note that we had merely to relink if we wanted to convert between an optimized program and a debuggable one. This is considerably more convenient than if we had had to recompile; if the inconvenience had been greater than it was, we might have taken the trouble to make the debugger smarter.

Our greatest success was the technique of link-time code modification. By itself, register allocation at link time may be overkill; very global optimization by monolithic compilation of source files or of intermediate-code files [12,21,36,45] or perhaps by reference to persistent program data bases [34] could still turn out to be a better tradeoff. But our machinery for code modification led us to develop a wide variety of tools for performance analysis at the source and machine levels. Most of these tools actually require very little of the machinery we implemented, but their advantage is profound. It has transformed *gprof* from something we use a few times a year into something we use a few times a month or week.

Late modification of the compiled program, though not a new idea, is getting an increasing amount of attention. It is a handy stage to reorder procedures or basic blocks to improve cache performance [27,32], because only when the code is in our hands do we know how big basic blocks and procedures really are. Benitez and Davidson [6] describe an optimizing linker that does fairly general peephole optimizations. Johnson [22] discusses an elegant approach called *postloading:* code modification occurs after linking and is completely separate from the linker, but the linker always retains in the executable file enough information for the postloader to understand the uses of addresses and the like.

Finally, we are also pleased with our use of Mahler as the ‘‘software-defined architecture’’ for the Titan. Three related but different machines have been designed under the Mahler umbrella, with few changes needed in the front-end compilers. The implementation of Mahler was good enough that users interested in performance were not heard to wish that they could get at the underlying machine. The main benefit of the ‘‘RISC philosophy’’ is the ability to make sensible tradeoffs between hardware and software. A software-defined architecture like Mahler allows us to make these tradeoffs flexibly and without causing turmoil in the higher levels of the environment.

History and acknowledgements

In 1982, Forest Baskett founded Digital Equipment's Western Research Lab. Its initial project was the development of the Titan, a high-performance scientific workstation intended to exploit the advantages of simplified architecture and hardware-software tradeoffs. Baskett and Michael L. Powell were interested in doing very global allocation before I joined the lab, and provided many prods and insights. Powell implemented our first pipeline scheduler, and the associated instruction-level instrumentation. David Goldberg invented the first version of the Mahler language and a global register allocator that worked at compile time, as well as the name ‘‘Mahler.’’ Loretta Guarino Reid provided a linker that worked on Titans, and contributed to the interface between the linker proper and the code modification facilities. Richard Beigel did a prototype study of register allocation during his summer with us, suggesting that allocation based on the call graph would be effective. Many thanks must also go to Jud Leonard, Gene McDaniel, Jeff Prisner, and Neil Wilhelm, for first advocating and then tolerating Mahler's extreme approach toward hiding the underlying machine.

In more recent years, newcomers have contributed to the applications in performance measurement. Anita Borg took the hack that enabled address tracing and incorporated it into a system that allows traces of multiprogram sets and even of kernel activity. Scott Nettles and Jeff Mogul spurred the implementation of link-time *gprof*.

Joel Bartlett, Patrick Boyle, Mary Jo Doherty, Alan Eustace, Norm Jouppi, Jeff Mogul, and John Ousterhout were unyieldingly critical of drafts of this paper, forcing me to strengthen it considerably. My thanks to them.

Finally, I must thank all of WRL's varying population over the years, for helping to build a stimulating environment in which strange ideas and real systems turn out to be the same thing.

Appendix 1. Generation of annotated code.

We assume that a basic block is represented as a sequence of commands of three kinds. A *leaf* command says to obtain the value of a particular variable. An *assign* command says to assign the value of a particular previous command to a particular variable. An *operate* command says to perform a particular operation on the results of two previous commands and make the new result available to later commands.

There are links from an *operate* or *assign* command to the commands it uses, and a link from a *leaf* or *assign* command to the symbol table entry for the variable it uses. Commands also have the following fields, which we will fill as we perform the algorithm:

```
index : cardinal;                 – – index in basic block
firstUse : link to Command;       – – link to first command using c
lastUse : link to Command;        – – link to last command using c
assigned : link to Variable;      – – where c gets assigned, if available
inLeaf : boolean;                 – – c is leaf and will be avail in leaf var
```

We also need Variables to have a field

```
nextAssn : cardinal;        – – index of command that next assigns to v
```

that we use for bookkeeping while we decorate the commands.

The first step is to find out if each value will be available, either in a leaf variable or as the result of an assignment, for as long as we need it. This is done with a forward pass over the basic block, followed by a backward pass. The forward pass is as follows:

```
index := 0
for each command c do
    c.index := index
    index := index + 1
    c.assigned := nil
    c.inLeaf := false
    c.firstUse := nobody
    for each command c2 that is an operand of c do
        if c2.firstUse = nobody then
            c2.firstUse := c
        c2.lastUse := c
    if c is ‘‘leaf v’’ or ‘‘v := rhs’’ then
        v.nextAssn := MAXINT
```

and the backward pass is as follows:

```
for each command c, in reverse order do
    if c is ‘‘v := rhs’’ then
        if (rhs.firstUse = c)
        and (v.nextAssn >= rhs.lastUse.index) then
            rhs.assigned := v
        v.nextAssn := c.index
    elseif c is ‘‘leaf v’’ then
        if v.nextAssn >= c.lastUse.index then
            c.inLeaf := true
```

Then we can generate annotated code. The main algorithm for this is on the next page.

```
procedure GenCode ( )
    for each command c do
        if c = ‘‘leaf v’’ then
            Generate an instruction to load v into a temporary register
            FlagLeaf (c, v)
        elseif c = ‘‘v := rhs’’ then
            Generate an instruction to store rhs from its temp reg into v
            FlagAssn (c, v, rhs)
            FlagOp (1, rhs, expr)
        else        – – c is an operation
            Generate one or more instructions to perform the operation.
            Call FlagOp(n,opnd,c) for any insts that use the value of opnd
                as the n-th operand, where opnd is some operand of c
            Call FlagResult(c) for any instructions that produce the result
                of this operation
```

This algorithm uses the following four subsidiary algorithms to annotate individual instructions:

```
procedure FlagLeaf (c : Expr; v : Symbol)
    if c.inLeaf and c.assigned=x then                 REMOVE.v  RESULT.x
    elseif c.inLeaf then                              REMOVE.v
    else                                              LOAD.v

procedure FlagAssn (c : Expr; v : Symbol; rhs : Expr)
    if rhs.firstUse ≠ c then                          STORE.v
    elseif rhs.assigned = nobody then                 STORE.v
    elseif rhs=‘‘leaf x’’ and rhs.inLeaf then         STORE.v  REMOVE.v  KEEP.x
    elseif rhs=‘‘leaf x’’ then                        STORE.v
    else                                              REMOVE.v

procedure FlagResult (c : Expr)
    if c.assigned = x then                            RESULT.x

procedure FlagOp (n : cardinal; opnd, c : Expr)
    if n=1 then op := OP1 else op := OP2
    if opnd=‘‘leaf v’’ and opnd.inLeaf then                 op.v
    if opnd.assigned=x and opnd.firstUse≠c then             op.x
```

References

[1] Advanced Micro Devices. *Am29000 Streamlined Instruction Processor User's Manual* (1987). Advanced Micro Devices, Inc., 901 Thompson Place, P. O. Box 3453, Sunnyvale, CA 94088.

[2] 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.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms,* pp. 189-195. Addison-Wesley, 1974.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools,* pp. 660-664. Addison-Wesley, 1986.

[5] 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.

[6] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pp. 329-338. Published as *SIGPLAN Notices 23* (7), July 1988.

[7] Robert Bernstein. Multiplication by integer constants. *Software — Practice and Experience 16* (7), pp. 641-652, July 1986.

[8] 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.

[9] G. J. Chaitin. Register allocation & spilling via graph coloring. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* pp. 98-105. Published as *SIGPLAN Notices 17* (6), June 1982.

[10] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages 6,* pp. 47-57, 1981.

[11] Frederick C. Chow. *A Portable Machine-Independent Global Optimizer — Design and Measurements.* PhD dissertation, Stanford University. Available as Computer Systems Laboratory Technical Note 83-254. Stanford University, December 1983.

[12] Fred C. Chow. Minimizing register usage penalty at procedure calls. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pp. 85-94. Published as *SIGPLAN Notices 23* (7), July 1988.

[13] Douglas W. Clark. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems 1* (1), pp. 24-37, February 1983.

[14] Jack J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. *Computer Architecture News 11* (5), pp. 22-27, December 1983.

[15] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems,* pp. 116-135. The MIT Press, 1985.

[16] Robert B. Garner, et al. The Scalable Processor Architecture (SPARC). *Digest of Papers: Compcon 88,* pp. 278-283, March 1988.

[17] 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.

[18] John Hennessy. Stanford benchmark suite. Personal communication.

[19] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems 4* (3), pp. 323-344, July 1982.

[20] 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.

[21] Mark I. Himelstein, Fred C. Chow, and Kevin Enderby. Cross-module optimizations: Its implementation and benefits. *Proceedings of the Summer 1987 USENIX Conference,* pp. 347-356, June 1987.

[22] S. C. Johnson. Postloading for fun and profit. *Proceedings of the Winter '90 USENIX Conference,* pp. 325-330, January 1990.

[23] Norman P. Jouppi and Jeffrey Y.-F. Tang. A 20 MIPS sustained 32 bit CMOS microprocessor with high ratio of sustained to peak performance. *IEEE Journal of Solid-State Circuits 24* (5), pp. 1348-1359, October 1989.

[24] Gerry Kane. *MIPS R2000 Risc Architecture.* Prentice Hall, 1987.

[25] Earl A. Killian. Personal communication.

[26] Ruby B. Lee. Precision Architecture. *IEEE Computer 22* (1), pp. 78-89, January 1989.

[27] 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).

[28] MIPS Computer Systems. *RISCompiler and C Programmer's Guide.* MIPS Computer Systems, Inc., 930 Arques Ave., Sunnyvale, California 94086. 1989.

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

[30] John Ousterhout. Personal communication.

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

[32] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation,* pp. 16-27. Published as *SIGPLAN Notices 25* (6), June 1990.

[33] 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.

[34] 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.

[35] Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization.* PhD dissertation, Stanford University. Available as Stanford Computer Systems Laboratory Technical Report CSL-TR-87-324. March 1987.

[36] 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.

[37] J. Strong, et al. The problem of programming communication with changing machines: A proposed solution. *Communications of the ACM 1* (8), pp. 12-18, August 1958, and *1* (9), pp. 9-15, September 1958.

[38] Thomas G. Szymanski. Assembling code for machines with span-dependent instructions. *Communications of the ACM 21* (4), pp. 300-308, April 1978.

[39] Christopher J. Terman. *User's Guide to NET, PRESIM, and RNL/NL.* M.I.T. Laboratory for Computer Science, 545 Technology Square, Room 418, Cambridge, Massachusetts.

[40] 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.

[41] 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.

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

[43] David W. Wall. Predicting program behavior using real or estimated profiles. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* pp. 59-70. Published as *SIGPLAN Notices 26* (6), June 1991. Also available as WRL Technical Note TN-18.

[44] 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. Published

as *Computer Architecture News 15* (5), *Operating Systems Review 21* (4), *SIGPLAN Notices 22* (10), October 1987. A more detailed version is available as WRL Research Report 87/1.

[45] William Wulf. Personal communication.

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.

‘‘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.