DECEMBER 1992

A Practical System for Intermodule Code Optimization at Link-Time

Amitabh Srivastava
David W. Wall

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

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

There two other research laboratories located in Palo Alto, the Network Systems Laboratory (NSL) and the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

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

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

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

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

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

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

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

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

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

A Practical System for Intermodule Code Optimization at Link-Time

Amitabh Srivastava
David W. Wall

December 1992

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

Abstract

We have developed a system called OM to explore the problem of code optimization at link-time. OM takes a collection of object modules constituting the entire program, and converts the object code into a symbolic Register Transfer Language (RTL) form that can be easily manipulated. This RTL is then transformed by intermodule optimization and finally converted back into object form. Although much high-level information about the program is gone at link-time, this approach enables us to perform optimizations that a compiler looking at a single module cannot see. Since object modules are more or less independent of the particular source language or compiler, this also gives us the chance to improve the code in ways that some compilers might simply have missed.

To test the concept, we have used OM to build an optimizer that does interprocedural code motion. It moves simple loop-invariant code out of loops, even when the loop body extends across many procedures and the loop control is in a different procedure from the invariant code. Our technique also easily handles ''loops'' induced by recursion rather than iteration. Our code motion technique makes use of an interprocedural liveness analysis to discover dead registers that it can use to hold loop-invariant results. This liveness analysis also lets us perform interprocedural dead code elimination.

We applied our code motion and dead code removal to SPEC benchmarks compiled with optimization using the standard compilers for the DECstation 5000. Our system improved the performance by 5% on average and by more than 14% in one case. More improvement should be possible soon; at present we move only simple load and load-address operations out of loops, and we scavenge registers to hold these values, rather than completely reallocating them.

This paper will appear in the March issue of *Journal of Programming Languages*. It replaces Technical Note TN-31, an earlier version of the same material.

1 Introduction

Code optimization is conventionally the domain of the compiler. This is sensible: it is the compiler that generates the code to begin with. Furthermore, the compiler begins with the original source code, and thus knows as much as possible about the intentions of the programmer, including the types of variables, the use of high-level operations, and pragmas that the programmer may have included to give the compiler even more help.

Nonetheless, a conventional compiler may be limited in two ways that degrade the code it produces.

First, in most environments the compiler looks at only one module at a time. This *separate compilation* would be hard to do without, since it greatly reduces the turnaround time for the run-edit-rebuild cycle.

Second, the compiler itself is divided into phases that must communicate. Retargetable compilers may have machine-independent phases that are insulated from the machine-dependent parts. The machine-independent optimizations may therefore not have a complete picture of the resources that are available. Similarly, in many environments the compiler produces assembly code rather than true object code. The advent of reduced instruction sets has led to environments in which the two are not isomorphic, and optimizations below the level of assembler cannot be done by the compiler.

We can compensate for these limitations by adding a link-time step that analyzes and optimizes the object code that is being linked. Though some of the high-level structure is missing from object code, link-time optimization has two important properties that compile-time optimization may not. First, at link time we can see the entire program at once, rather than just a single separately compiled module. Second, we *are* looking at object-level code, which even the compiler may not have done. These two properties lead to several advantages.

One advantage is that seeing the whole program at once allows more precise analysis. We don't have to assume the worst when we see a call to an unknown routine. Among other things this allows alias analysis of global variables, something difficult to do well at compile-time.

Seeing the whole program also gives us a better understanding of the tradeoffs of code generation. It is easier to have an overall picture of where the program probably spends its time and other resources, and thereby where optimization will pay off most.

As mentioned previously, the compilation process can be quite fragmented by the need to deal with separately compiled modules and by the division of labor between the phases in the compiler and assembler. Separate compilation forces us to compile one module without knowing much

about its companion modules, and the generation of assembly code instead of object code may mean that some resources must be dedicated to the use of the assembler and unavailable to the compile-time optimizer. This fragmentation of compilation requires the adoption of conventions about the use of registers, the stack, and even instruction selection. With the whole program at hand at the object level, however, we need not be bound by these conventions, allowing more global management of these resources.

Similarly, working with the final code means that we can do machine-level optimizations. The compiler may generate assembly-level operations that we can see to be multiple-instruction sequences in the object code, parts of which may be loop-invariant and eligible for code motion.

Finally, working like the linker at the object level gives us a certain degree of independence from the particular compilers in our environment, and even from the particular source languages. Naturally we cannot expect to analyze and improve code regardless of its origin. Nevertheless, alternative approaches like monolithic compilation at the source level or even at the intermediate code level are much less flexible than our approach.

Of course, modifying object code has disadvantages, too. Many parts of the high-level structure have been irretrievable lost. Other parts are retrievable only with some effort, and our system has to undo and redo some of the work done by the normal compiler system, such as compiling of idioms like case-statements or the filling of delayed branch slots. These difficulties are the price we chose to pay for intermodule analysis without a strong dependency on a particular compiler or language. A richer object format would alleviate some of them; in fact, some kinds of "lost" structure, such as type information, may still be available if the program has been compiled with debugging.

In this paper we describe OM, our system for link-time code optimization, and place it in the context of other work in the area. As a proof of concept, we then describe our use of OM to build a link-time optimizer that does interprocedural liveness analysis and interprocedural code motion. Finally, we evaluate the performance of our link-time optimizer, which improved compiler-optimized code by 5% on average and by more than 14% in one case.

2 The OM System

OM takes as input a collection of object files and libraries that make up a complete program. It processes the input modules in three phases: construction of the intermediate representation from input modules, application of instrumentation and optimization, and generation of object code from the intermediate representation. Figure 1 shows the organization of OM. OM currently runs

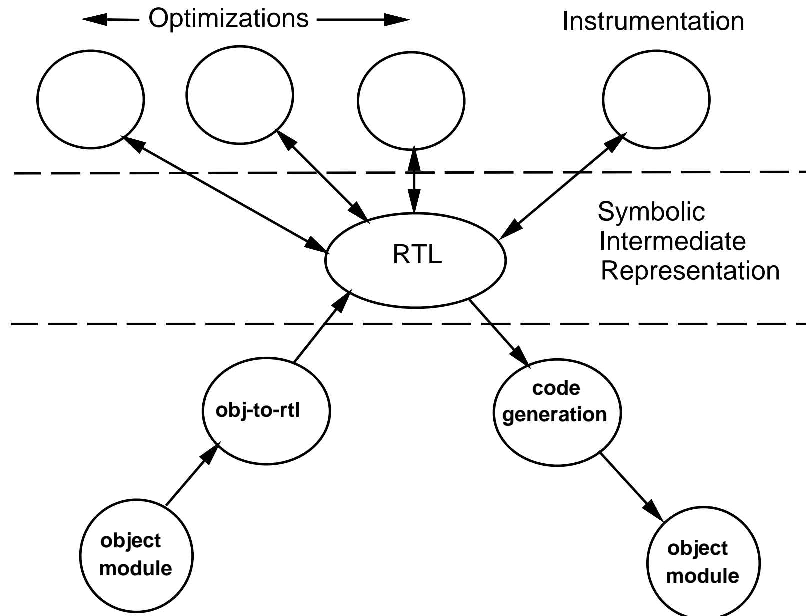

Figure 1: System organization of OM

under Ultrix on a DECstation[1] and has been tested with modules produced for MIPS processors by Fortran, C, C++, and Modula-2 compilers.

Construction of Intermediate Representation

The first phase processes the input object files in their usual form, with a loader symbol table and relocation tables, to build the intermediate form. The relocation information lets us distinguish addresses from coincidental numeric constants [Wal92]; we can therefore perform our transformations without introducing the dynamic translation overhead of schemes that transform executable files [LB92, MIP86]. We can also determine exactly which procedures and variables have had their addresses taken; because we can see the whole program at once, we need not make worst-case assumptions about code we cannot see.

The intermediate representation of OM consists of a simple Register Transfer Language (RTL) and a symbol table. This RTL is machine-independent but has been designed for RISC architectures. It has generic instructions such as **add** and **sub** that operate on registers and constants. Only **load** and **store** instructions access memory. Procedure calls are simple transfers of control, and parameter passing is explicitly exposed. RTL uses an infinite register set model

[1]Ultrix and DECstation are trademarks of Digital Equipment Corporation.

with some dedicated registers such as stack-pointer.

All address references in RTL are symbolic. PC-relative branches are converted into branches to targets that are labels in RTL. Similarly all references to data areas are also converted to symbolic references. In MIPS code, the pair of related operations, R_REFHI and R_REFLO, are used to relocate a pair of instructions that compute a 32-bit address by combining two 16-bit parts. We find the referenced symbol from the relocation entries, add it to the symbol table, and use the symbol in its RTL instruction. This conversion to purely symbolic addresses allows us to freely move and modify RTL instructions without worrying about harming unrelated instructions.

Recovering the original structure of a source-level case-statement is necessary if we are to know as much about the control structure as the compiler did. A case-statement is compiled as an indirect jump to an address loaded from some location in a branch table indexed by the case index value. The branch table for a case statement is normally laid out in the read-only data area with addresses of the different branch targets stored in consecutive locations. The code for the case-statement is easy to recognize, and the address of the branch table is obtainable by examining this code. By finding all the case-statements in a program, we partition the table space into different branch tables, which lets us know the size of each. This in turn tells us the possible destinations of each indexed jump. We can cross-check the table size against the part of the case-statement code that does a range check on the index.

Certain assembly-level operations, such as load-address and double-precision loads and stores, are revealed in the object code as sequences of simpler instructions. Although this allows us to optimize this code in ways not possible at the assembly level, it is still helpful to understand how these simpler instructions fit together. The relocation tables are helpful in reassociating the parts of a load-address operation, since the instruction fields that are combined to make up the address must be marked for relocation. However, double-precision loads and stores require some additional work. A double-precision load is really two single-word loads from an even/odd register pair accessing adjacent memory locations. Since a double-precision load occurs in an even/odd pair, we use the odd floating-point register in a load instruction as the starting point, and search for the matching instruction, which must use the conjugate register and must address the adjoining location.

There are no delayed branches in our RTL; when we read a MIPS module, we must identify instructions in delay slots and move them out for representation in RTL. A branch slot is often filled with an instruction from before the branch, in which case this instruction can be safely moved back before the branch. If the slot changes a register which the branch uses, it is incorrect simply to move the instruction back before the branch. Instead, we duplicate the branch slot and

push it ahead to both the destination address and the fall-through address. If we can see that the slot instruction is dead code in either successor location, we need not copy it there. In producing the RTL, we check for deadness simply but conservatively, by seeing whether the basic block in question kills the instruction directly; this occasionally causes unnecessary duplication, but interprocedural dead code removal and peephole optimizations performed in the optimization phase can remove any that remain.

Analysis and Optimization

The second phase of OM divides the RTL form of the program into a collection of procedures. Instructions in each procedure are divided into basic blocks. We build the usual control flow graphs for each procedure, and a complete call graph for the entire program. These data structure are used by the various optimization and analysis phases. OM used these data structures to analyze C++, C, and Fortran programs and measured the amount of unreachable code.[Sri91]. Each optimization pass may add more information to the RTL which following passes may use. Since optimizations operate on the RTL, machine-independent optimizations would be unchanged when OM is retargeted to handle other architectures.

Code Generation

The final phase of OM translates the RTL into object module for the target architecture. All addresses must be recomputed; instructions may have been moved, altered or deleted. We must introduce delay slots if required by the target architecture, and then schedule the code to fill them and to remove other pipeline hazards and stalls. We also heuristically change the order of output of procedures to help in cache behavior.

If the target architecture is same as the input architecture, code generation is more or less a matter of reversing the work done in the input phase. If OM is used to translate from one architecture to another, different code generation is required. The RTL is intended to be machine-independent, so this should not be a problem: several existing compilers already generate code from a machine-independent RTL [DF84, Sta92]. At present, however, we do no architectural translation, so both the input and the output to OM consist of MIPS modules.

3 Overview of Prototype

As a proof of concept, we implemented a simple form of invariant code motion, along with some other simpler optimizations that work with it. OM looks for loop-invariant operations to move out of loops. These loops may include several entire procedures, and can be induced by either iteration or recursion. For this prototype, we restricted ourselves to the simplest kinds of invariant code, because the hard part is defining what is meant by a multi-procedural loop. The operations it looks for at present are load-address operations (which are guaranteed to be invariant) and loop-invariant load operations. Because these operations are so simple, their results must be kept in registers for moving them to pay off, so we also do a liveness analysis to find registers that are unused over the range of the loop. This liveness analysis in turn makes it easy to do interprocedural dead code removal.

Section 4 describes our liveness analysis, which treats the entire program as a single flow graph and uses interprocedural summary information to filter the flow into or out of a procedure. Section 5 describes our current optimizations.

4 Interprocedural Live Variable Analysis

Our data flow analysis takes advantage of modern large memories by using a direct, brute-force approach. We treat the entire program as a single flow graph, with edges between procedures corresponding to calls and returns. In the manner of Myers [Mye81], we split a procedure call into a pair of conjugate blocks: the *call* block executed just before the transfer to the called procedure, and the *return* block executed just after the called procedure. A call block has a single successor, the *entry* block of the called procedure. A return block has a single predecessor, the *exit* block of the called procedure. An entry block, of course, has as many predecessors as there are places that call its procedure; likewise for successors of exit blocks. Because we deal with code at the object level, the argument passing and stack manipulation that are associated with procedure calls are represented explicitly and are not considered part of the call itself. Figure 2 shows an example program graph. Procedures *p1* and *p2* each contain a call to procedure *p*. Interprocedural edges are present from the call blocks to the entry of *p*, and from the exit of *p* to the return blocks.

As usual, for each block B in the program, we want to compute sets IN[B] and OUT[B] of variables that are live immediately before and after B. We treat the whole program as one big flow graph so that this information can cross procedure boundaries; for example, we want to consider **x** and **y**, but not **z**, as live at the entry to *p*. We must take care, however, to disallow information

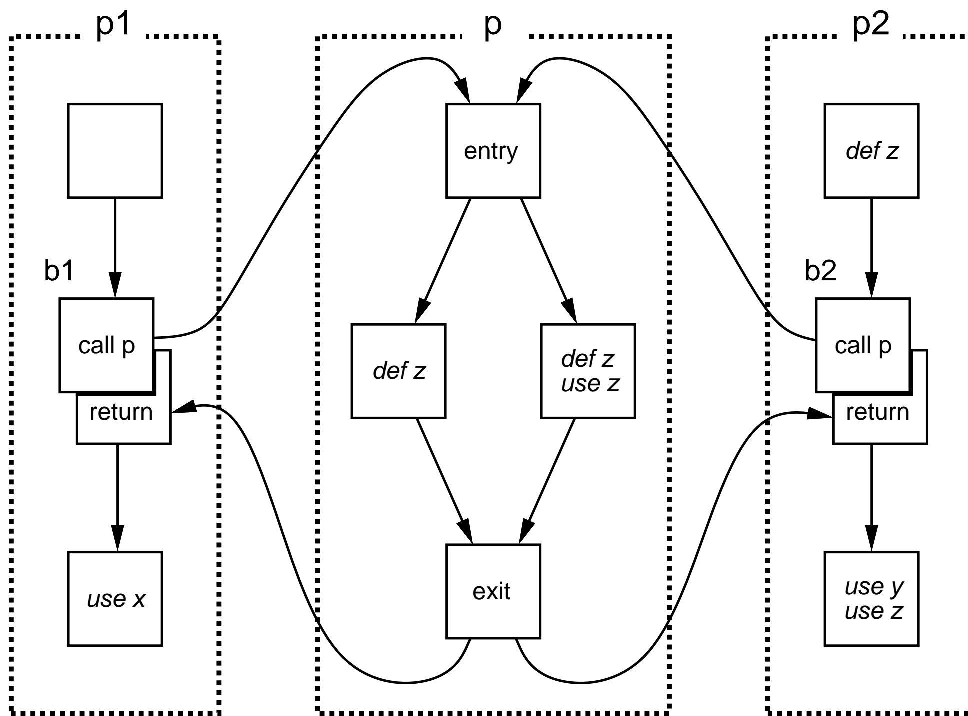

Figure 2: Single flow graph model

flow on paths that cannot be followed in a real execution. For example, there is a path from block b1 through procedure *p* (where **y** is live) to the use of **y** at the end of *p2*. But it would be imprecise to conclude from this that **y** is live at b1, because it is meaningless to talk about calling *p* from b1 and then returning to the conjugate block for some other call.

To accomplish this, we use a two-phase approach similar to the one described by Landi and Ryder [LR90] for alias analysis. In the first phase, we delete all return edges and let information flow over normal edges and call edges, and from a return block to its conjugate call block. In this phase we compute both liveness (which is "may" information) and its counterpart deadness (which is "must" information). Because information does not flow over return edges, this phase gives a precise, flow-sensitive summary of the possible effects of calling each procedure, including the effects of any calls that procedure makes. We then replace the return edges and delete the call edges, using the procedure summary information from the first pass to compute the variables that are live before a call. This is similar to Callahan's approach [Cal88], which computed the same flow-sensitive summary information but went to considerable trouble to avoid having the whole program in memory at once. Because information can flow from a caller to a callee as well as vice versa, this results in more precise information than approaches like those of Barth [Bar77] or Banning [Ban79], which use flow-insensitive summary information to capture the effects of a call within the analyzed procedure but do not tell the analyzer how control got to this procedure in the first place.

Figure 3 sketches our two-phase dataflow algorithm and the equations we use in each phase.

The equations for a normal basic block are the same as in a standard liveness analysis. The variables that are live on exit from B are simply those live on entry to any successor of B. The variables that are live on entry to B are those live on exit but not set by B, along with those used by B before being set by B. Note that in Phase 2, the successors of a procedure exit block are the corresponding return blocks, and information flows along these edges.

Since a call/return pair consists only of transfers of control, these blocks neither set nor use variables. The IN and OUT sets of such a block are therefore identical. A variable is live at a call block for either of two reasons: it may be a variable that is used during the procedure call, or it may be a variable that is used after return from the called procedure but was not killed during the procedure call. During the first phase, we model this by allowing information to flow to a call block both from the entry to the procedure and from the call block's conjugate return block. During the second phase, when information flows into the procedure from many different return blocks, we cut off information flow across call edges and use instead the entry sets we converged to in the first phase. This prevents imprecise information from flowing through a procedure

Phase 1: Delete return edges and compute LIVE (“may” information) and DEAD (“must” information), according to the following equations.

normal blocks:

DEF[B] = variables defined by B before any use in B
USE[B] = variables used in B before any definition in B
LIVEIN[B] = USE[B] $\cup$ LIVEOUT[B] $-$ DEF[B]
LIVEOUT[B] = $\cup_s$ LIVEIN[S] for all successors S of B
DEADIN[B] = DEF[B] $\cup$ DEADOUT[B] $-$ USE[B]
DEADOUT[B] = $\cap_s$ DEADIN[S] for all successors S of B

call and return blocks:

LIVEOUT[call] = LIVEIN[entry] $\cup$ LIVEOUT[return] $-$ DEADIN[entry]
LIVEOUT[return] = $\cup_s$ LIVEIN[S] for all successors S of return
DEADOUT[call] = DEADIN[entry] $\cup$ DEADOUT[return] $-$ LIVEIN[entry]
DEADOUT[return] = $\cap_s$ DEADIN[S] for all successors S of return
IN sets for a call or return block are identical to its OUT sets

When this converges, define, for each procedure P:

PUSE[P] = LIVEIN[B] where B is the entry to P
PDEF[P] = DEADIN[B] where B is the entry to P

and then throw away the LIVE and DEAD sets computed in this phase.

Phase 2: Restore return edges and delete call edges, and compute LIVE (“may” information) according to the following equations.

normal blocks:

DEF[B] = variables defined by B before any use in B
USE[B] = variables used in B before any definition in B
LIVEIN[B] = USE[B] $\cup$ LIVEOUT[B] $-$ DEF[B]
LIVEOUT[B] = $\cup_s$ LIVEIN[S] for all successors S of B

call and return blocks, for call to procedure P:

LIVEOUT[call] = PUSE[P] $\cup$ LIVEOUT[return] $-$ PDEF[P]
LIVEOUT[return] = $\cup_s$ LIVEIN[S] for all successors S of return
IN sets for a call or return block are identical to its OUT sets

Figure 3: Algorithm and data flow equations for live variable analysis

between a return block and a call block that are not conjugates.

Because we want to determine liveness of registers as well as user variables, we add one improvement to this algorithm. Procedures commonly save a register on entry and restore it on exit, and then utilize the register for some unrelated purpose in between. Neither such unrelated references, nor the saves and restores that protect them, are considered true uses or definitions in our analysis. We could extend this specialization to user variables as well, but such protected unrelated uses of a user variable are rare.

Indirect procedure calls via procedure variables pose a problem for any interprocedural analysis. We model such a call as a call to an abstract procedure that in turn calls each actual procedure that might be the value of the procedure variable. Determination of this set of actual procedures can be crude or sophisticated. Ours is crude: we have exactly one abstract procedure, and it calls each procedure whose address is ever taken or specified as an initial value. We handle global variables similarly: a read or write via an unknown pointer is assumed to read or write any variable whose address is ever taken.

We use a variation of the standard iterative algorithm [ASU88] to solve the data flow equations: we repeatedly examine the basic blocks, computing the IN and OUT sets from the equations, until we make one complete pass without changes. The standard iterative algorithm would visit the basic blocks in reverse-depth-first search order, but this is complicated by the fact that this order changes from phase 1 to phase 2. Instead, we compute the depth-first order for the blocks in each procedure independently, and also the depth-first order of the procedures in the call graph. In either phase, we visit all the nodes of a procedure consecutively, in their own reverse-depth-first order. In phase 1, we visit the procedures themselves in reverse-depth-first order, so that called procedures tend to be visited before their callers. In phase 2 we reverse this, so that callers tend to be visited before the procedures they call. In practice this required no more iterations than the normal "flat" depth-first ordering that ignores procedures, and saved us both the time and the memory needed to compute the "flat" ordering.

5 Interprocedural Optimizations

The OM system performs several optimizations, with more planned. In this section we discuss two: interprocedural dead code elimination and interprocedural loop-invariant code motion.

Both require the interprocedural live variable analysis described previously. Interprocedural data flow information improves the effectiveness of normal intraprocedural optimizations. For example, movement of code out of loops may be limited by the availability of registers to hold

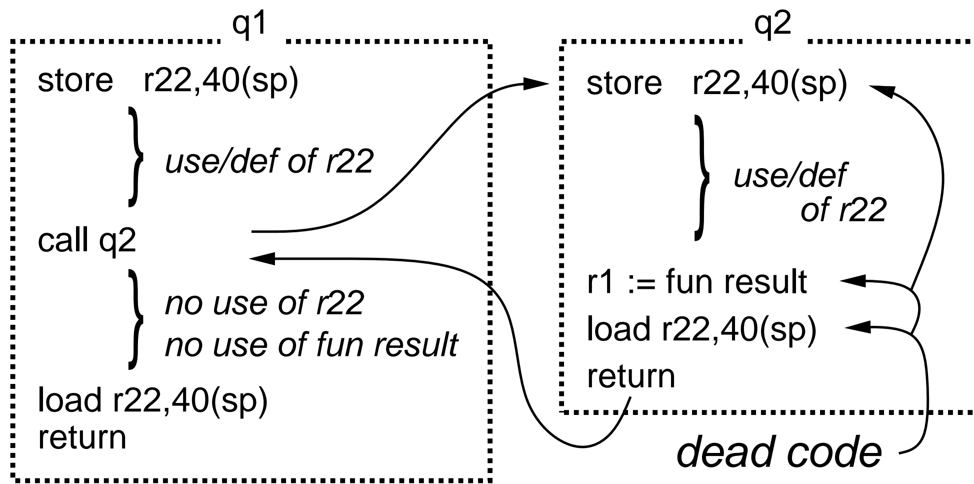

Figure 4: Interprocedural dead code elimination

the invariant results; interprocedural information can help us find more free registers. Code that a given procedure considers part of its job can be seen as useless in the wider context.

Because our analysis is more precise than the traditional summary approach, opportunities like these arise more often. Moreover, optimizing the whole program at once lets us perform optimizations such as code motion out of loops even across procedure boundaries.

5.1 Interprocedural Dead Code Elimination

Interprocedural liveness information allows us to remove dead code that would not be recognized as dead by a compiler. Some examples are shown in Figure 4. The compiler has treated **r22** as a callee-save register, but we can see that it is dead at the only call to *q2*. The save and restore in *q2* are therefore useless and can be removed. Similarly, *q2* returns a function result that is not used by its caller, a common occurrence in languages such as C; our analysis allows us to remove the computation of the result.

To discover that these pieces of code are dead, we need information about *q1* to be available when we examine *q2*. The traditional summary approach would have found neither of these, because summary information flows only up the call graph from *q2* to *q1*.

5.2 Interprocedural Loop-Invariant Code Motion

Code inside a loop whose effect does not change from one iteration to the next can be moved outside the loop. Mechanisms for detecting loops and loop-invariant code are well-understood

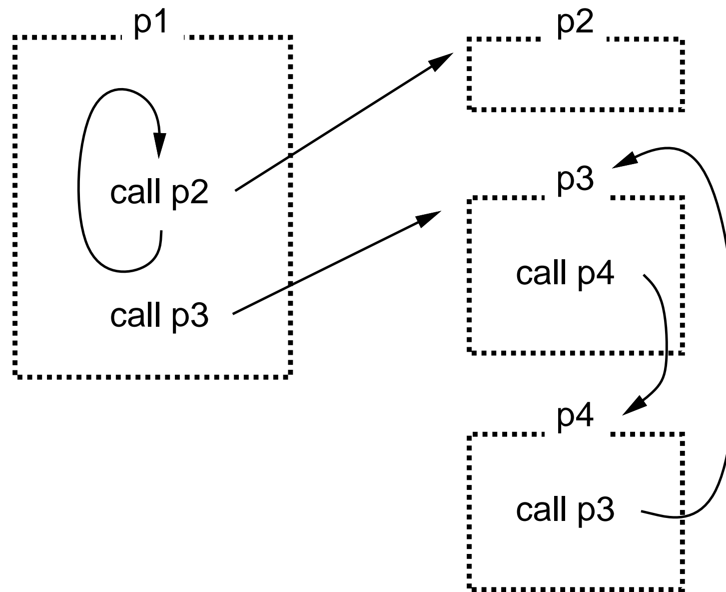

Figure 5: Interprocedural loops

for single procedures [ASU88], but certain loops are not completely visible to techniques that process one procedure at a time. Figure 5 shows two examples. Procedure *p2* is called from a loop in procedure *p1*. Looking at *p1* or *p2* in isolation will not tell us if *p2* contains loop-invariant code. Procedures *p3* and *p4* call each other in a recursive cycle. Code that is invariant across both could be moved out into *p1*.

Richardson [Ric91] suggested that this kind of optimization could be accomplished by procedure inlining followed by normal intraprocedural analysis and optimization. That approach has several drawbacks. Inlining already presupposes some kind of interprocedural analysis to find the most useful inlining opportunities. Loops that arise through recursion cannot be made intraprocedural by inlining. And inlining can have unpredictable and sometimes detrimental effects on cache performance unless cache behavior is used to guide inlining decisions [McF91]. Good interprocedural analysis together with judicious use of procedure cloning [CHK92] would avoid these problems.[2]

Though we are looking for loop-invariant code across procedure boundaries, our ambitions in this prototype are modest. Our liveness analysis can tell us when a register is dead or unused

[2]Richardson also found that inlining overwhelmed the existing intraprocedure optimizer, sometimes causing it to crash and sometimes simply confusing it so badly that optimization was degraded. This probably reflects only how the intraprocedural optimizer was written and tuned rather than any fundamental principle, but it shows that inlining does not simply reduce interprocedural optimization to the "previous problem" of intraprocedural optimization.

over a particular part of the program, and we look for simple, obviously invariant values to keep in these registers. At present, the only operations we move across procedure boundaries are load-address operations, which produce constant values by definition.

There are three parts to the problem: determining the extent of the loop, determining the set of available registers, and selecting the operations to move.

Determining the extent of the loop is the most interesting part. The classical way of finding a loop is to look for a strongly-connected component of the flow graph that is dominated by some point just outside it. This approach does not work well here, for two reasons. First, the call-edges and return-edges result in many "false loops" that are not computationally valid: control cannot ever follow these paths in a real computation, which is why we needed to use summary information to filter the data flow over these edges. Even if we somehow took this into account, however, this approach would fail to find loops that call utility routines reachable from elsewhere. We want a loop-finding algorithm that will let us correctly find loops that range over several procedures, while excluding from the bodies of those loops routines that are not dominated by their entry points.

We approach the problem of finding interprocedural loops by concentrating on the call graph rather than the flow graph. We look for a "heavy" call, one that gets executed many times, and treat the procedure that makes that call as the loop header. We then try to move instructions from the called procedure, or procedures reachable from it, across the heavy call and into the loop-header procedure. Procedures in this loop region must be dominated by the loop-header procedure, so we can move invariant instructions there and be sure they will be executed whenever necessary. And no path between points in the loop region may pass outside the region, so that we can be sure that an available register really is unused throughout.

We begin by building a weighted call graph. An edge $p \rightarrow q$ is weighted by an estimate of how many times we will call q each time we call p. These weights are based on our discovery of loops and recursive calls. To compute the weight of edge $p \rightarrow q$, initialize it to zero. For each call from p to q, add 10^d where d is the number of loops in p that contain the call. Then for each recursive call to q from anywhere, multiply the weight of $p \rightarrow q$ by 10. Note that these weights do not accumulate as we move down the call graph; the weight on edge $p \rightarrow q$ is independent of the weights on paths leading to p.

A call edge $p \rightarrow q$ with a large weight implies that some kind of loop is present, because q is called more often than p. If the call graph were a tree, the loop region would contain q and all its descendants, these being the procedures reachable via a call from p to q. Real call graphs are not trees, however, so we must do two more things to this region. First we add paths that lead from

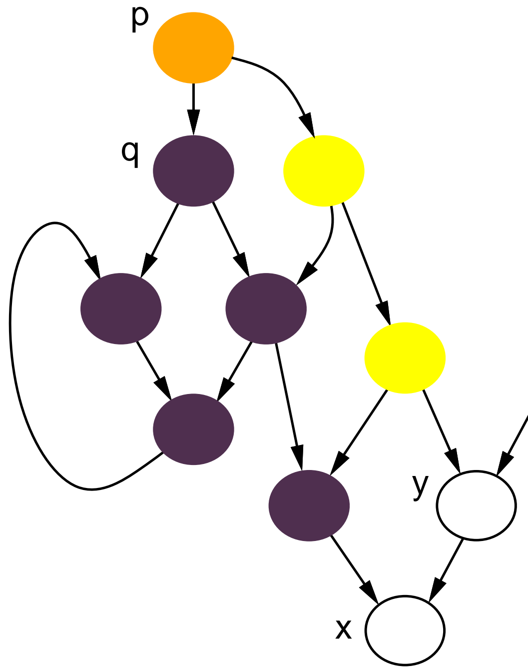

Figure 6: The loop region (all shaded nodes) induced by edge $p \to q$

p to the loop region without going through q. Then we prune the region to exclude procedures called from outside the region, so that all procedures in the region are dominated by p. Details of this algorithm appear in the Appendix.

Note two things about the resulting region. First, there may be calls from it to procedures outside the region. We call such a procedure a *stepchild* of the region. Stepchildren should perhaps be considered part of the loop, but they are also reachable from outside the loop, so it is not safe to move instructions from them to p.[3] Second, if p does not dominate q, we prune the region so thoroughly that it contains only p. All that is left is the part of the loop in p itself, and only instructions from p will be considered for code motion. If the loop arose because q is recursive rather than because of a loop in p, not even p will contain eligible instructions.

Figure 6 shows an edge $p \to q$ and its associated loop region. The darkest nodes are procedure q and those descendants that are dominated by p. The lightest nodes are those we added to include all paths from p to nodes in the region. Nodes x and y are stepchildren. They may be tentatively added to the region, but will later be pruned because they are not dominated by p.

Given this loop region, we look for invariant code to remove. This does not require a deep

[3]Cloning stepchildren would let us add them to the loop region, at the cost of increasing program size.

analysis, because we look only for load-address operations, which by definition have a constant result. Each candidate operation is given an estimated execution count by combining its loop depth in its own procedure with the weights on the call graph edges in paths leading to its procedure. These operations are ordered according to this estimate, so those with the highest payoff will be given registers first.

Finally we look for registers to hold the invariant values. Such a register has to be unmentioned throughout the region (except possibly for a save/restore around the body of p). An unmentioned register is available if it is dead at the exit of p and at the entry to each stepchild of the region, or if it is unmentioned in each stepchild and its descendants (except when protected by a save/restore). In the first case a stepchild or its descendants may independently define and use the register, but we can still consider it available if we insert a save/restore around the call to the stepchild. However, it is very likely that the compiler has already generated the necessary save/restore, in which case we need not. In the second case the register may be live throughout the region because its value is set before the call to p and used afterward. If it is live at p's exit, we must insert a save/restore around the body of p; dead code removal will excise this save/restore if it is redundant. The costs of inserted save/restores are factored into the decision to use a register; code motion with a small payoff might not outweigh the expense of the save/restores.

If we find n available registers, we can move the n heaviest operations out to the beginning of procedure p. In the original position of each we leave a register-register copy to get the value in the right place. These copies will be cleaned up later.

The three-step process of deriving the loop region, prioritizing invariant operations, and finding registers to hold their values is done for each call graph edge with a weight greater than 1.

5.3 Intraprocedural Loop-Invariant Code Motion

After interprocedural code motion, we perform a more traditional phase that moves invariant code out of loops within a single procedure. There are two reasons why this is effective even though the compiler has already optimized the code OM sees.

First, our interprocedural liveness analysis is more precise than a compiler can achieve looking at one module at a time. This means we can find available registers that the compiler could not.

Second, since we integrate register allocation with code motion, we can move very small pieces of invariant code that represent significant savings only if their results are kept in registers. At present we move only load-address operations and single loads, mostly of global or stack variables, that we are sure produce invariant values. In contrast, a compile-time optimizer may

be less thoroughly integrated with register allocation, and so concentrate on invariant code that pays off even if its result is kept in memory.

5.4 Cleanup and Procedure Ordering

After each of the two code motion phases, OM performs a cleanup phase consisting of copy propagation, common subexpression removal, and interprocedural dead store removal. This cleanup gets rid of the register-register copies we added, redundant save/restores, and duplicate pieces of invariant code that were moved to one spot from multiple locations. Because the code we start with was previously optimized, nearly all of the opportunities for copy propagation and common subexpression removal are those introduced by our code motion algorithm.

When all the optimization and cleanup is finished, we write the procedures in depth-first order, rather than the original order. This simple heuristic is intended to improve the locality and hence the cache behavior.

6 Where Do Link-Time Opportunities Come From?

Our invariant code motion depends on finding two things: a piece of invariant code that can be moved, and a dead register to keep it in. Why do either of these exist? Doesn't their presence mean the compiler dropped the ball? We do not think so.

As we discussed earlier, the compiler is hindered by the need to support separate compilation and by its own fragmentation into different phases. Our ability to analyze the entire program at once, all at the object code level, means that we can find opportunities easily that would have been hard for the compiler to find.

Consider dead registers first. In the MIPS compilers, some registers are designated as caller-save or callee-save, whether the resulting saves are really needed or not. One pair of registers is reserved for return values, even though many procedures return no results and few others return two-register results. Similarly, another register is reserved for short-term uses by the assembler, which may not arise at all in some part of the program. Intermodule analysis can show us where these conventions are unnecessary.

Next consider moveable invariant code. Some instances of this are visible to us because we allow such code to move across procedure boundaries; a compiler supporting separate compilation does not have that luxury. Even moving code within a single procedure, however, may be easy for us but not for the compiler, because we have a more complete picture of available resources.

Classical code motion is machine-independent, and therefore often looks only for code that is expensive enough to move even if its result must be kept in a memory location. Register allocation is likely to be done in an entirely different phase. Our approach, in a sense, is "machine-dependent code motion" in that it is integrated with knowledge of what registers are available. As a result, we can move tiny pieces of code whose result must be kept in a register for its motion to pay off.

An example of what this integration can buy is our handling of Fortran common blocks. Because the declared size of a common block may vary from module to module, the MIPS Fortran compiler leaves allocation of these areas to the linker. References to them are more expensive than references to normal variables, typically requiring an address calculation instead of a direct reference relative to a known base register. In our RTL we represent a reference to a common variable as a load-address of some nearby base address in the common area, followed by a reference relative to that base. This allows us to combine the redundant load-address parts of many different references to the same common block, by keeping their common base address in a register. In this way we can access variables in the common block much as the compiled code accesses variables in the "small data sections," which consist of small global scalars that are kept together in memory so that a single dedicated register, the global pointer, can be used to access them quickly.

7 Performance

We used OM to optimize nine SPEC benchmarks. Both OM and the benchmarks ran under Ultrix on a DECstation 5000 with 48 Mb memory. Object modules given to OM were produced by the standard MIPS compilers with optimization level -O2.

We measured the speed of an executable in two ways. The first was real user time, according to the system clock as measured by the *systime* facility. The second was cycles as measured by instrumenting the executable using *pixie* [MIP86]. The pixie tool counts instructions and pipeline stalls, and gives a theoretical cycle count assuming no cache, memory, paging, or I/O delays.

OM was able to improve the performance of each of the benchmarks, sometimes slightly and sometimes significantly. This is unsurprising, because the MIPS -O2 option causes only intraprocedural optimization. To compare our code with that of a peer, we also compiled the benchmarks using the MIPS options that cause interprocedural optimization. In one case we used the MIPS -O3 option by itself; in the other we gave it a dynamic profile and also invoked the -cord option, which reorders procedures based on the profile to improve cache behavior. The effects of the MIPS interprocedural was spotty; it was common for it to produce worse code than -O2 did.

	speedup with runtime measured by					
	systime			pixie		
	OM	MIPS (opt level)		OM	MIPS (opt level)	
doduc	14.40%	11.77%	(O3)	2.48%	1.39%	(O3p)
eqntott	5.13%	9.77%	(O3)	5.56%	10.91%	(O3p)
espresso	0.30%	0.55%	(O3p)	2.96%	1.13%	(O3p)
fpppp	12.51%	0.00%	(O2)	10.51%	0.00%	(O3p)
gcc1	1.03%	0.00%	(O2)	1.19%	0.66%	(O3p)
li	1.52%	0.00%	(O2)	0.63%	0.80%	(O3p)
nasa7	0.12%	0.00%	(O2)	0.85%	0.00%	(O2)
spice	12.36%	0.36%	(O3p)	17.30%	1.00%	(O3p)
tomcatv	2.57%	0.00%	(O2)	7.89%	0.00%	(O3p)
average	5.55%	2.49%		5.49%	1.77%	

Figure 7: Speedup relative to MIPS -O2

Figure 7 shows the reduction in runtimes for OM and for the best code produced by the MIPS compilers using -O2, -O3, or -O3 with a profile and -cord (denoted “O3p” in the figure). All these times are normalized by the times for code compiled with MIPS -O2, as measured by systime or pixie respectively. Thus a reduction of 0% means the interprocedurally optimized program took exactly as long as it did when optimized with only MIPS -O2; in many cases -O2 gave the best code the MIPS compilers could produce.

OM was able to improve performance over MIPS -O2 by about 5% on the average, and by 12% or more in some cases. Code motion accounts for nearly all of this, but precise attribution is difficult because the code motion depends on dead code removal, to remove useless operations and thereby free more registers and also to clean up the code afterwards. Even compared to the MIPS interprocedural optimization, OM looks good; only for eqntott was the MIPS optimizer able to do much better than OM.

Also of interest are the costs of using OM. Figure 8 shows the total time OM takes to process each benchmark. This time includes reading the object files and libraries, converting them into a single RTL structure, optimizing, and generating assembler code.[4] About 75% of the time is spent on the analysis and optimization, with the remaining time divided evenly between producing RTL from object and producing assembler from RTL. We have done essentially no performance tuning

[4]In the future we plan to generate an executable directly, which should be faster than generating a textual assembler file. This is also why we do not include assembly time in these figures.

	total OM time (secs)	one liveness analysis (secs)	phase 1 iterations	phase 2 iterations	static block count	static instr count
doduc	14.98	0.966	4	4	6759	47524
eqntott	4.85	0.384	4	4	2946	10200
espresso	19.80	1.683	4	4	11901	46452
fpppp	14.58	0.867	4	4	6013	41540
gcc1	88.95	7.583	6	4	46536	169608
li	9.08	0.833	4	4	5863	18884
nasa7	8.40	0.633	4	4	4697	21656
spice	37.05	2.833	5	6	16173	92956
tomcatv	6.53	0.534	4	4	3995	15100

Figure 8: OM processing time statistics

and do many things by brute force; for example, the liveness analysis is actually performed from scratch five separate times. Figure 8 also shows how long one complete execution of the liveness analysis takes. Each liveness analysis requires around eight iterations, correlating weakly with the static number of basic blocks in the program. It is interesting that each phase requires about the same number of iterations that are required to converge on a single procedure [ASU88], even though the whole-program graph is much larger and more deeply nested. Presumably this is because the absence of either all call or all return edges hides the global depth of loop nesting.

One reason we can do so much in a minute or less of processing is that we are quite free in our use of memory. Figure 9 shows how much memory OM used for each benchmark. In general it needs memory space equal to about 18 times the disk space occupied by the object code.

8 Conclusions

We have shown that very modest interprocedural optimization can deliver a significant improvement in performance even without inlining or cloning. To do this, it must be driven by precise interprocedural data flow analysis that allows information to flow in both directions between caller and callee. We have shown that obtaining information this precise need not be extravagantly expensive, even using brute force algorithms. Our system requires no explicit help from the compiler itself, and is thus relatively independent of both source language and compiler.

We plan to extend our prototype in several ways. First, a full register allocator that renames existing registers when possible without degrading pipeline performance would free more registers

	object size	OM memory	ratio
doduc	0.37 Mb	5.5 Mb	14.7
eqntott	0.10 Mb	2.3 Mb	23.0
espresso	0.40 Mb	6.9 Mb	17.3
fpppp	0.35 Mb	5.0 Mb	14.3
gcc1	1.31 Mb	24.0 Mb	18.3
li	0.20 Mb	3.7 Mb	18.5
nasa7	0.19 Mb	3.4 Mb	17.9
spice	0.73 Mb	10.8 Mb	14.8
tomcatv	0.14 Mb	2.8 Mb	20.0

Figure 9: OM memory requirements

for other purposes. Second, more ambitious recognition of loop-invariant code would give us more uses for these registers, and could reveal invariant code that was worth moving even if its result had to be kept in memory. Dynamic profiles could guide careful procedure cloning, allowing key procedures to be added to loop regions even when they are not dominated by the region header, which could lead to a more general and elegant construction of a loop region.

Beyond these short-term improvements, we expect to implement other traditional loop optimizations in our interprocedural framework. We hope OM will continue to be an effective platform for the study of both very global optimization and machine-dependent optimization.

Acknowledgements

We are very grateful to Mary Jo Doherty, Mary Fernandez, Joel McCormack, Scott McFarling, and Paul Vixie for careful reading and pertinent comments, and to Mick Jordan for insisting that we measure our performance improvement with pixie along with systime.

Appendix. Construction of an Interprocedural Loop

Input: A call graph C=(P,E) where P is the set of procedures and E is the set of call edges, a depth first spanning tree D of C, and an edge e = $(p_h \rightarrow p_e)$.
Output: LoopRegion(e), a subset of the procedures of C.
Method:

constant Family = $\{p \mid p$ is a descendant of p_h in D $\}$ /* LoopRegion $\subseteq$ the */
LoopRegion = $\{p_h, p_e\}$ /* descendants in D of p_h */
while changes to set LoopRegion **do** /* Build initial set of nodes */
    for each p in Family$-\{p_h\}$, in depth-first order, **do** /* reachable from p_e */
        if $p \in$ LoopRegion **then**
            for each procedure q that p calls **do**
                if $q \notin$ LoopRegion and $q \in$ Family **then**
                    LoopRegion = LoopRegion $\cup$ $\{q\}$

while changes to set LoopRegion **do** /* Add callers of nodes already */
    for each q in Family, in reverse-depth-first order, **do** /* in region, if all callers */
        if $q \in$ LoopRegion **then** /* are in Family */
            addcallers = **true**
            for each procedure p that calls q **do**
                if $p \notin$ Family **then** addcallers = **false**
            if addcallers **then**
                for each procedure p that calls q **do**
                    if $p \notin$ LoopRegion then
                        LoopRegion = LoopRegion $\cup$ $\{p\}$

while changes to set LoopRegion **do** /* Ensure region has a single */
    for each q in Family$-\{p_h\}$, in depth-first order, **do** /* entry point p_h */
        if $q \in$ LoopRegion **then**
            for each procedure p that calls q **do**
                if $p \notin$ LoopRegion **then**
                    LoopRegion = LoopRegion - $\{q\}$

References

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1988.

[All74] F.E. Allen. Interprocedural data flow analysis. *Proceedings of IFIP Congress 1974*, pp. 398–402. North Holland, 1974.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 29–41, January 1979.

[Bar77] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM 21*(9), pp. 724–736, September 1978.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 47–56. Published as *SIGPLAN Notices 23*(7), July 1988.

[CHK92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. *Proceedings of the 1992 International Conference on Computer Languages*, pp. 96–105, IEEE Computer Society Press, April 1992.

[DF84] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems 6*(4), pp. 505-526, October 1984.

[LR90] William Landi and Barbara Ryder. Pointer-induced aliasing: A problem classification. *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 93–103, January 1990.

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

[Lom77] D. Lomet. Data flow analysis in presence of procedure calls. *IBM Journal of Research and Development 21*,6, pp. 559–571, 1977.

[McF91] Scott McFarling. Procedure merging with instruction caches. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 71–79. Published as *SIGPLAN Notices 26*(6), June 1991.

[MIP86] MIPS Computer Systems, Inc. *Language Programmers's Guide*, 1986.

[Mye81] E. Myers. A precise inter-procedural data flow algorithm. *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 219–230, January 1981.

[Ros79] B. Rosen. Data flow analysis for procedural languages. *Journal of the ACM 26*(2), pp. 322–344, April 1979.

[Ric91] S.E. Richardson. *Evaluating Interprocedural Code Optimization Techniques*. Ph.D. Thesis, Stanford University, Technical Report No CSL-TR-91-460, February 1991.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, N.J, 1981.

[Sri91] Amitabh Srivastava. Unreachable procedures in object-oriented programming. WRL Technical Note TN-21, November 1991.

[Sta92] Richard Stallman. Using and porting GNU CC. Free Software Foundation, 1992.

[Wal92] David W. Wall. Systems for late code modification. *Proceedings of the CODE 91 Workshop on Code Generation*, Springer Workshops in Computer Science, to appear. Also available as WRL Research Report 92/3, May 1992.

WRL Research Reports

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

WRL Technical Notes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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