FEBRUARY 1994

WRL
Research Report 94/1

Link-Time Optimization of Address Calculation on a 64-bit Architecture

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

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

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

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

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

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

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

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

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

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

Link-Time Optimization of Address Calculation on a 64-bit Architecture

Amitabh Srivastava
David W. Wall

February 1994

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

Abstract

Compilers for new machines with 64-bit addresses must generate code that works when the memory used by the program is large. Procedures and global variables are accessed indirectly via *global address tables***, and calling conventions include code to establish the addressability of the appropriate tables. In the common case of a program that does** *not* **require a lot of memory, all of this can be simplified considerably, with a corresponding reduction in program size and execution time.**

We have used our link-time code modification system OM to perform program transformations related to global address use on the Alpha AXP. Though simple, many of these are *whole-program* **optimizations that can be done only when we can see the entire program at once, so link-time is an ideal occasion to perform them.**

This paper describes the optimizations performed and shows their effects on program size and performance. Relatively modest transformations, possible without moving code, improve the performance of SPEC benchmarks by an average of 1.5%. More ambitious transformations, requiring an understanding of program structure that is thorough but not difficult at link-time, can do even better, reducing program size by 10% or more, and improving performance by an average of 3.8%.

Even a program compiled monolithically with interprocedural optimization can benefit nearly as much from this technique, if it contains statically-linked pre-compiled library code. When the benchmark sources were compiled in this way, we were still able to improve their performance by 1.35% with the modest transformations and 3.4% with the ambitious transformations.

This paper will appear at the SIGPLAN '94 Conference on Programming Language Design and Implementation. It replaces Technical Note TN-35, an earlier version of the same material.

1 Introduction

Obtaining addresses of procedures or global data on a new 64-bit RISC machine such as the Alpha AXP [4] presents an elementary but unavoidable problem. Traditional 32-bit RISCs have evaded the worst aspects of the problem by using various architectural tricks, such as special-format instructions like the SPARC Call [2], which packs a 2-bit opcode and a 30-bit word address into a 32-bit instruction, or special-purpose instructions like the MIPS Load-Upper-Immediate (LUI) [3], which facilitates the construction of a 32-bit address from two 16-bit immediate fields. These techniques break down when we move to machines with 64-bit addresses and 32-bit instructions.

The solution used by the Alpha AXP compilers running Alpha/OSF is a typical one for the coming generation of 64-bit machines. Global addresses are collected together into one or more *global address tables* (GATs). Each separately-compiled module has its own GAT, but at link-time the GATs are collected together into one GAT section. The generated code accesses the GAT via a dedicated register called the *global pointer* (GP) together with a 16-bit immediate displacement.

Accessing a program object then involves two steps. First, the address of the object is loaded from the GAT section. We refer to this load as an *address load*, to distinguish it from the traditional *Load-Address* operation, which adds a literal displacement to a base register and does not access memory at all. The address obtained by this address load is then used to load or store the variable, or jump to the procedure, that it refers to.

In general, each routine must have an associated value for the global pointer, which may differ from the value for some other routine. There are two reasons for this. First, for large programs, the global address table may be so large that it cannot be accessed via a single unchanging global pointer. Second, a module from a dynamically-linked shared library must have its own global address table, which may be mapped to an address far from the table for the rest of the program.[1]

A compiler looking at a single module does not know whether the complete program is large or small, nor whether the module is destined to be part of a shared library. It must generate code that will work regardless, even if this general code is not the fastest code in some cases. Improving this code is straightforward, but in general requires analysis and code generation that

[1]It might seem that procedures in the same compilation unit would always use the same GP, but the semantics of shared libraries means that a call that the compiler thinks is intra-unit might turn out to be inter-unit at dynamic-link-time. In particular, a module with exported routines P1 and P2 can be dynamically linked with a program even though a different implementation of P1 is already present in the executable; if P2 calls P1, it gets the version already present, not the one in the same module [1]. Under static linking, of course, this situation would result in a link error because the symbol P1 is multiply-defined. Note that it *is* possible to optimize a call to an *unexported* routine in the same module at compile-time.

crosses module boundaries; a thorough job requires this to include even library routines, which usually were compiled long before a particular application.

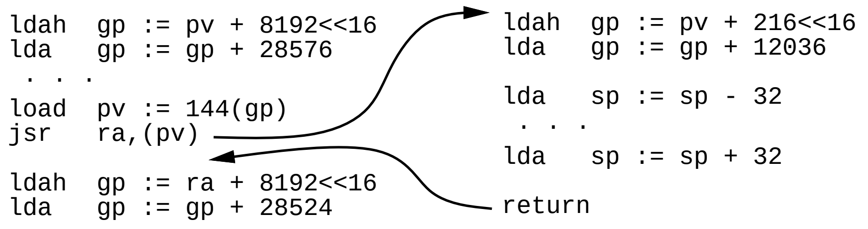
Link-time code optimization can be very helpful in dealing with this problem. At link-time, the optimizer can analyze the entire program, see how big the address table needs to be, and simplify address calculation when possible. How thoroughly the linker can do this depends primarily on how hard it is willing to work. Some improvement is possible even for a “traditional” linker that limits each change to the replacement of one instruction by another. Even more improvement is obtained by giving the linker the ability to move instructions around, because it can therefore exploit optimization opportunities better than the traditional linker, and also find opportunities the traditional linker could not.

This paper describes our tool for link-time address optimization, and shows how much performance improvement is achieved. We start by detailing our assumptions about code generation on 64-bit machines, and on the Alpha AXP in particular. We then discuss how the resulting code can be improved, and describe our system for doing so. Finally we present the static and dynamic measurements of how much we were able to improve the code.

2 Generating code for 64-bit machines

Our model for compiling on a 64-bit processor follows from the considerations described above. We assume that a reference to a procedure or variable is generally done by loading the address from a global address table (GAT) and following the resulting pointer to get to the object in question. A dedicated register called the global pointer (GP) is used to maintain the addressability of the global address table. The different GATs need not be adjacent in memory (though they may be); in particular, modules linked dynamically from shared libraries may have their code, GAT, and data in memory far from the statically linked part of the program. At link-time, each separately-compiled module has one or more GATs used by the procedures in that module. When the linker combines these modules, it treats these GATs as literal pools, removing duplicate addresses and merging the individual GATs into a single large GAT if possible.

All values in a GAT must be accessible via its GP value and a machine-dependent displacement, so merging into one large GAT will not always be possible. As a result, procedure-calling conventions must suppose that the caller and callee will need different values of GP, and must therefore include code to set up a new value for GP when a procedure is called, and code to restore the GP value for the caller upon return. Either of these pieces of code may be conventionally done by either the caller or the callee.



```
ldah  gp := pv + 8192<<16          ldah  gp := pv + 216<<16
lda   gp := gp + 28576             lda   gp := gp + 12036
 . . .
load  pv := 144(gp)                lda   sp := sp - 32
jsr   ra,(pv)                       . . .
                                   lda   sp := sp + 32
ldah  gp := ra + 8192<<16
lda   gp := gp + 28524             return
```

Figure 1: Calling conventions on the Alpha AXP

On the AXP under Alpha/OSF, each procedure is responsible for setting up its own proper GP value, both when the procedure is entered and when control returns to the procedure after it makes a call. It does this by adding a 32-bit displacement to a register containing a nearby code address. Calling conventions ensure that such a register exists, and memory layout conventions ensure that a procedure's GAT is close enough. Adding a 32-bit displacement to a register value can be done in two instructions: the Load-Address-High (LDAH), which adds the upper 16 bits of the displacement, and an ordinary Load-Address (LDA) to add the lower 16 bits.

Figure 1 shows a typical AXP calling sequence. The routine on the left sets its GP on entry and resets it after return from the contained call. On entry it computes the GP from the value of the PV register, which by convention contains the procedure entry address; after the return it uses the return address register RA. The 32-bit displacement is different between these two operations by exactly the difference between the two code addresses on which the computation is based. The called procedure sets its own GP in the same way, and need not save the caller's GP. The address of the destination procedure is obtained by loading it from the GAT as we described previously.

The AXP has a limited-range call instruction whose destination is specified by a 21-bit word displacement from the program counter. If the destination is near enough in the same compilation unit, the compiler can use this BSR instruction instead of the more general JSR. The BSR does not require us to load the destination address into PV, but the compiled code normally does so anyway, because the called procedure needs the PV in order to set up its value for GP.

With the calling conventions properly setting the GP, accessing global variables is easy. Obtaining the address of a variable is done by an address load from the GAT, in the same way that

```
load  r1 := 188(gp)      load  r1 := 188(gp)      load  r1 := 188(gp)
                         load  r2 := 0(r1)        store 0(r1) := r2
```

(a) get address of V   (b) fetch value of V   (c) assign value to V

Figure 2: Global variable access on the Alpha AXP

we obtain the address of the destination procedure in a call. A fetch from or assignment to the variable consists of the address load followed by a load or store through the address just loaded. Figure 2 shows these three code fragments.

3 Improving the compiled code

The code generation conventions described above are conservative: the compiler does not know how big the eventual program will be, so it generates code that will work regardless. In a typical reasonably-size program, however, the resulting code may be slower than necessary. If we can see the whole program at once, there are several ways to improve this code.

First, the GAT section is often small enough that nearby user variables can be referenced directly via the GP, rather than indirectly through addresses in the GAT. (This is even more effective if the compiler segregates the small data into its own data section, even if the compiled code accesses small and large data identically.)

Second, most programs are small enough that a few distinct GATs suffice; most often one is enough. We can omit GP-resetting after calls between routines that use the same GAT.

Third, we can replace the general call instruction by the BSR instruction, if the destination procedure is near enough. If the two procedures also share the same GAT, we can change the destination of the BSR to skip the GP-setting instructions in the destination procedure prolog. This in turn means that we have no more use for the destination address in the PV, and can omit that address load.

Fourth, removing address loads for the above reasons means that we can remove many addresses from the GAT altogether. This *GAT-reduction* acts to increase data locality, but also means that the GAT gets smaller, perhaps enabling a fresh round of the other improvements.

Finally, optimizations cause widespread changes to the program code, including the removal of many multi-cycle load instructions. The code we started with had been pipeline scheduled at compile-time, based in part on the presence of these loads. It is reasonable to suppose that we can improve the code still further by performing pipeline scheduling as a final step.

Most of these modifications are examples of *whole-program* optimizations. They can be done only if we can examine the entire program at once, or at least (if shared libraries are used) the entire statically-linked part of it. Otherwise, for example, we do not know how big the GAT and data sections will eventually be, and therefore cannot change GAT-based variable references into GP-based references. Even a compiler that can compile and optimize many modules together will be unable to perform these optimizations, if the program contains modules in other source languages or library modules compiled at an earlier time.

We therefore advocate doing these optimizations at link-time. This requires the linker to have a rather deeper understanding of the program control flow than has hitherto been typical for linkers, but this analysis is not difficult, especially since the loader format normally provides us with a few hints. References to the GAT section must be marked for relocation, because otherwise normal linking will not work. For similar reasons, the AXP compilers include links between an instruction that loads an address and the subsequent instructions that use it. Finally, the loader format identifies procedure boundaries and specifies the correct value of GP for each procedure.

4 Our study

We have built an optimizing linker for the Alpha AXP using our object-code modification system OM. A more complete description of how OM works appears in our previous paper [6]. The OM linker translates the object code of the entire program into symbolic form, recovering the original structure of loops, conditionals, case-statements, and procedures. It then analyzes this symbolic form and transforms it by instrumenting or optimizing it, and generates executable object code from the result. It can be thorough but still conservative in understanding the input object code because it can use the loader symbol table and the relocation tables to clarify the code.

The key idea behind OM is the translation into symbolic form and back. This means that we can perform interesting transformations like code optimization on object code without having to keep track of the effects of these transformations on address constants and branch displacements. (The symbolic form is also rather more amenable to dataflow and other analysis, though these were not needed in this study.)

Because OM lets us work with a symbolic form, we can easily delete and reorder instructions. Both of these capabilities are important. Deletion lets us get rid of the instructions we don't need. Reordering lets us create opportunities that would not exist otherwise. For example, the instructions to set the GP on entry to a procedure are often moved later in the code by the compile-time pipeline scheduling, preventing us from changing the BSR instruction to skip them; if we can restore them to their logical place at the beginning of the procedure, we can avoid executing them on most or all of the calls.

On the other hand, translation to a symbolic form is rather more work than a "traditional" linker normally does. If we are unwilling to delete or reorder the code, there are still gains to be had. Instead of deleting unneeded instructions, we can replace them by no-ops. This has two benefits. First, changing normal instructions to no-ops removes data dependencies. A no-op might cost nothing even if we don't delete it, because it might fit into some other instruction's latency delay or in a multiple-issue slot unavailable to the original instruction. Second, replacing address loads with no-ops removes the danger of a cache miss or page fault. The GAT might be used consistently enough to stay resident, but removing a reference altogether renders the question moot.

For this study, we have therefore used OM to implement two different levels of address optimization: *OM-full* and *OM-simple*.

OM-simple consists of the address-calculation optimizations that a traditional linker can do using only local analysis and no code motion. Unneeded instructions are removed by changing them to no-ops, and the order of the remaining instructions is never changed. If a variable is sufficiently close to the GAT, we change references to it into GP-relative references; this includes even references within the reach only via a 32-bit displacement, because the LDAH instruction lets us make a direct GP-relative reference in the same number of instructions as an indirect reference via the GAT. When possible, we change GP-swapping instructions to no-ops and call instructions to BSRs. We sort the `common` symbols by size and place them with the small data sections near the GAT, and use a simple heuristic to pick a good value for the GP.

OM-full consists of the full set of address-calculation optimizations discussed previously. These require OM to understand the control structure of the program, finding all the transfers of controls and all the places where code addresses are used either explicitly or implicitly. OM-full can optionally include rescheduling the code after all the other optimizations are finished. Rescheduling includes quadword-aligning instructions that are the targets of backward branches, which is intended to improve the behavior of the AXP's dual-issue and cache.

We applied OM-simple and OM-full to code produced by the standard Alpha/OSF optimizing compilers.[2] The only changes OM made to the code were the address-calculation optimizations described above. OM's analysis and optimization encompassed all the modules in a program, including user modules and non-shared versions of all library modules, because we wanted to maximize the performance improvement and because OM does not yet support shared libraries. The programs we used are the SPEC92 suite with the exception of gcc, which we could obtain only in 32-bit mode.

[2]DEC Fortran v3.3 for Alpha/OSF, and the DEC OSF/1 T1.3-1 (Rev. 19.3) C compiler.

5 Results

We measured OM's effectiveness on two different versions of each benchmark, which we call *compile-each* and *compile-all*.

Compile-each was obtained by compiling the benchmark's source files separately with only intraprocedural global optimization.[3] We used OM to link the resulting object files and the standard libraries, performing either OM-simple or OM-full in the process. We then took static and dynamic measurements of the resulting code, and compared them to measurements of code produced by linking the compile-each version with no link-time optimizations.

The *compile-all* version was obtained by linking all of the benchmark's source files together into a single object file with the maximum level of compile-time optimization.[4] This optimization level does both intraprocedural and interprocedural optimization, including some of the address-calculation optimizations OM does. As before, OM then linked the compile-all version with the pre-compiled standard libraries, performing either OM-simple or OM-full. We took static and dynamic measurements of the resulting code, and compared them to measurements of code produced by linking the compile-all version with no link-time optimizations.

Finally, we measured the processing time required by OM and by the standard linker for the supported levels of optimization.

5.1 Static measurements

We measured three static properties of the test suite after optimization by OM-simple and OM-full, compared to the original program. We first measured how many address loads appeared. We then measured how often the various parts of the procedure-calling conventions were required. Finally we simply measured the total number of instructions.

We first counted how many address loads were eliminated by OM-simple and OM-full. An address load can be eliminated in the two ways we discussed earlier. It can be *converted*: changed to a load-address operation in the form of an LDA or LDAH. Or it can be *nullified*: changed to a no-op (in OM-simple) or deleted altogether (in OM-full). Figure 3 shows the static fraction of address loads that were eliminated for each program. The two leftmost bars give the changes for the compile-each versions of the programs, the two rightmost for the compile-all versions. The dark part of each bar shows how many address loads were converted, and the light part shows how

[3] Compiler option -O2.

[4] Compiler option -O4 for Fortran and -O3 -Hb for C.

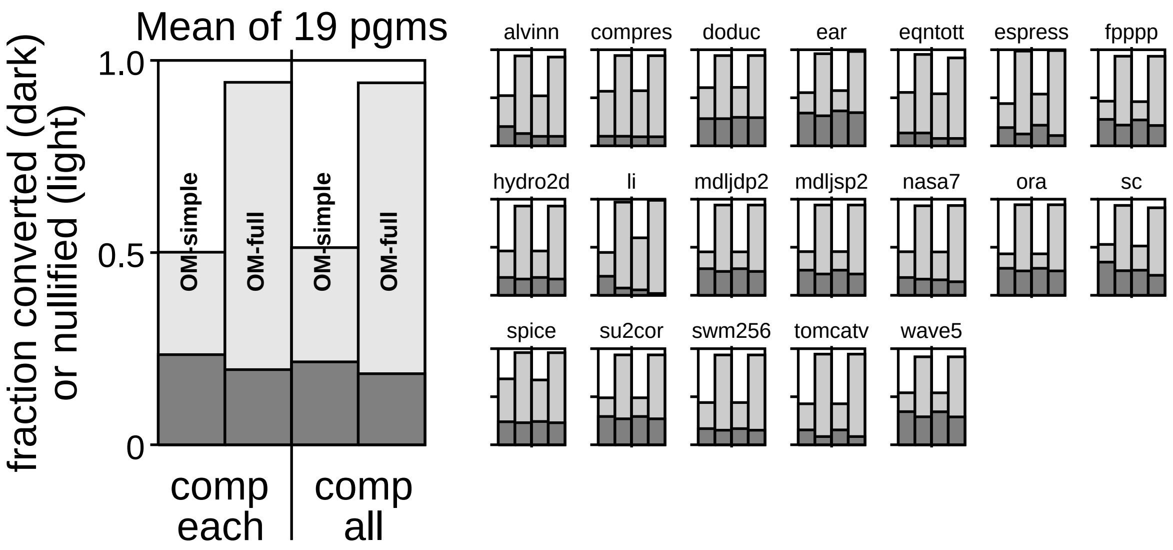


Figure 3: Static fraction of address loads removed, whether converted (dark) or nullified (light)

many were nullified. The bar lengths in the large key to the left show the unweighted arithmetic means of the values in the 19 test programs.

OM-simple finds essentially all of the available opportunities for converting address loads to load-addresses. (This should happen for any program whose static data area is less than 2^{32} bytes.) On average about the same number of address loads are nullified into no-ops. About half of all address loads are either converted or nullified.

OM-full manages to eliminate nearly all of the address loads. The number of address loads converted to load-addresses decreases slightly: GAT-reduction lets us reach more of the data via the GP, so OM-full can nullify distant references that OM-simple can only convert.

An important source of improvement is the bookkeeping code involved in procedure calls. An unoptimized call site has four instructions: one to load the PV with the destination address, one for the JSR, and two to reset the GP after returning. In the right circumstances, OM can change the JSR to a BSR and nullify the other three instructions. Figure 4 shows how well it could do this.

The compiler itself can perform these transformations for calls to an unexported procedure in the same compilation unit, but the "no OM" results show that there are few such calls. Even when the benchmark source files are compiled together with interprocedural optimization, statically around 85% of the calls still require all the bookkeeping code.

In contrast, even OM-simple can change essentially all JSRs in the test programs to BSRs;

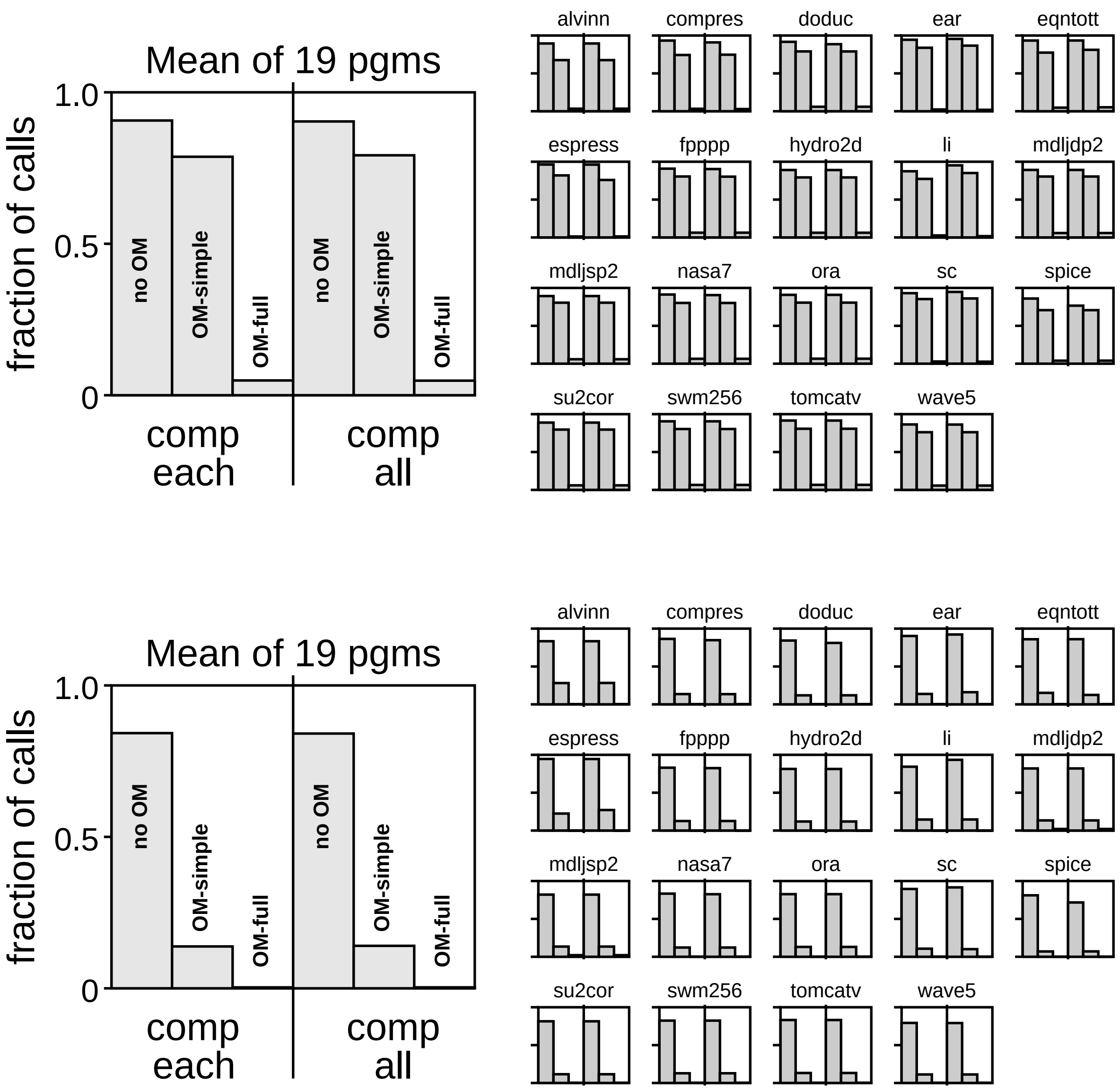


Figure 4: Static fraction of calls requiring PV-loads (top) and GP-reset code (bottom)

this requires no analysis at all except to look up destinations in the GAT and see if they are close enough. If the GP-setting instructions of the called procedure appeared first, it would also be easy to change the BSR to skip them. Unfortunately, compile-time scheduling often moved them, preventing OM-simple from doing this. This in turn prevents OM-simple from nullifying the PV-load at the call site, because the PV is needed so that the GP-setting code in the callee will do the right thing. OM-full removes all but a few PV-loads; those remaining are in calls through procedure variables, preventing us from examining the destination procedure.

OM-simple does a better job at nullifying the two GP-resetting instructions. It can usually change them to no-ops even if they were moved by compile-time scheduling. OM-full does a little better, using its understanding of the control structure to find a few cases OM-simple misses.

Surprisingly, the results without using OM show that there is little difference in these measures between code that was interprocedurally optimized by the compiler (compile-all) and code that was not (compile-each). Compile-time interprocedural optimization can improve calls from one user routine to another, but cannot help calls to previously compiled library routines.[5]

The combination of improvements to variable-access code and to procedure-calling overhead results in a substantial reduction in the static number of useful instructions required. Figure 5 shows that OM nullifies a substantial fraction of the code. Each graph shows first how well OM-simple and OM-full did on the compile-each version of the program, and then how well they did on the compile-all version. Again, the key on the left side shows the unweighted arithmetic means over the set of programs.

Even OM-simple nullifies (but does not delete) around 6% of the instructions. OM-full deletes an astonishing eleven percent of the instructions on average, and often more.

OM's ability to improve the code is not dependent on whether the code was originally compiled with interprocedural optimization: the average improvement of compile-all code was nearly equal to that of compile-each code. Compile-time interprocedural optimization does not have access to the whole program and can therefore do nothing to improve access to variables. Moreover, while it can improve calls from one user routine to another, it cannot be as helpful on calls to previously compiled library routines. These calls are in fact quite common; in the spice benchmark, for example, statically half the calls are from one library routine to another!

OM-full reduced the size of the GAT by an entire order of magnitude, reducing it to between

[5]This lack of difference is partly irrelevant. One effect of interprocedural optimization is the inlining of user routines; if a multiply-inlined user routine contains a library call then that call will be replicated. This should not harm performance because the library routine will be called the same dynamic number of times, but it will tend to increase the static number of calls that cannot be improved.

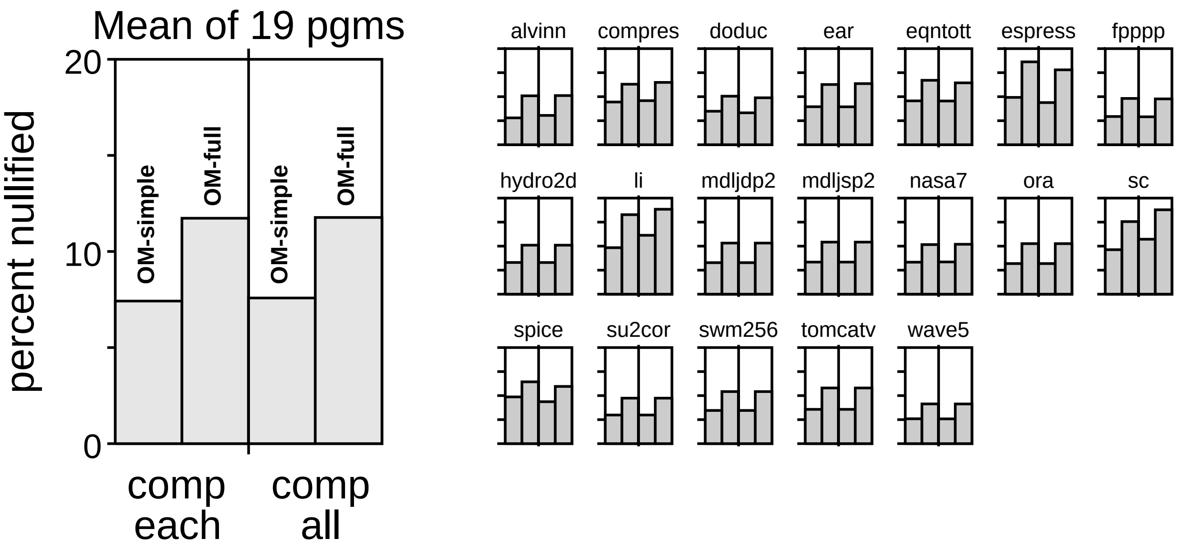


Figure 5: Static fraction of instructions nullified

3% and 15% of its original size. In was slightly more effective on compile-each versions than on compile-all versions, because compile-all does a little GAT-reduction of its own before OM gets a chance.

5.2 Dynamic measurements

We ran the different versions of the benchmarks on a DECstation 3000 Model 400 AXP, a dual-issue machine, and measured the user and system time according to the system clock as measured by the *systime* facility. To provide standards to compare to, we first linked the compile-each and compile-all version of each benchmark with no link-time optimization at all, and timed the result. We then linked using OM-simple and OM-full, and compared the resulting execution time to the corresponding standard time.

Figure 6 shows the improvement of code optimized by OM at link-time over code not optimized at link-time. As before, each graph shows first how well OM-simple and OM-full did at improving the compile-each version of the program, and then how well they did at improving the compile-all version. The key on the left side shows the unweighted arithmetic means over the set of programs.

The dynamic improvements were smaller than the static improvements, for a couple of reasons: cache misses and page faults mean that many cycles are spent doing things other than user instructions, and the dual issue of this model of the AXP means that some instructions come "free" so that deleting them does not help. Nevertheless the results are satisfying given how easy they are to attain.

OM-simple improved the performance of the compile-each programs by 1.5% on average. Seven benchmarks improved by more than 1%; the median was 0.6%. OM-full did considerably better, improving the code by 3.8% on the average and by more than 5% in six cases; the median was 2.8%. Unfortunately, OM-full made the code slightly slower in two cases. This was probably because of cache or paging effects, since loop alignment and scheduling didn't eliminate the problem.

OM's payoff was nearly as high even on the compile-all versions. OM-simple improved code by an average of 1.35%, and OM-full by 3.4%; this is 90% of the improvement seen on the compile-each versions. Again, OM-full improved six programs by more than 5%, and the median was again 2.8%. Even if interprocedural optimization speeds up the program, it still leaves plenty for a link-time address-calculation optimizer to find.

We expected that rescheduling the code after finishing the address-calculation optimizations

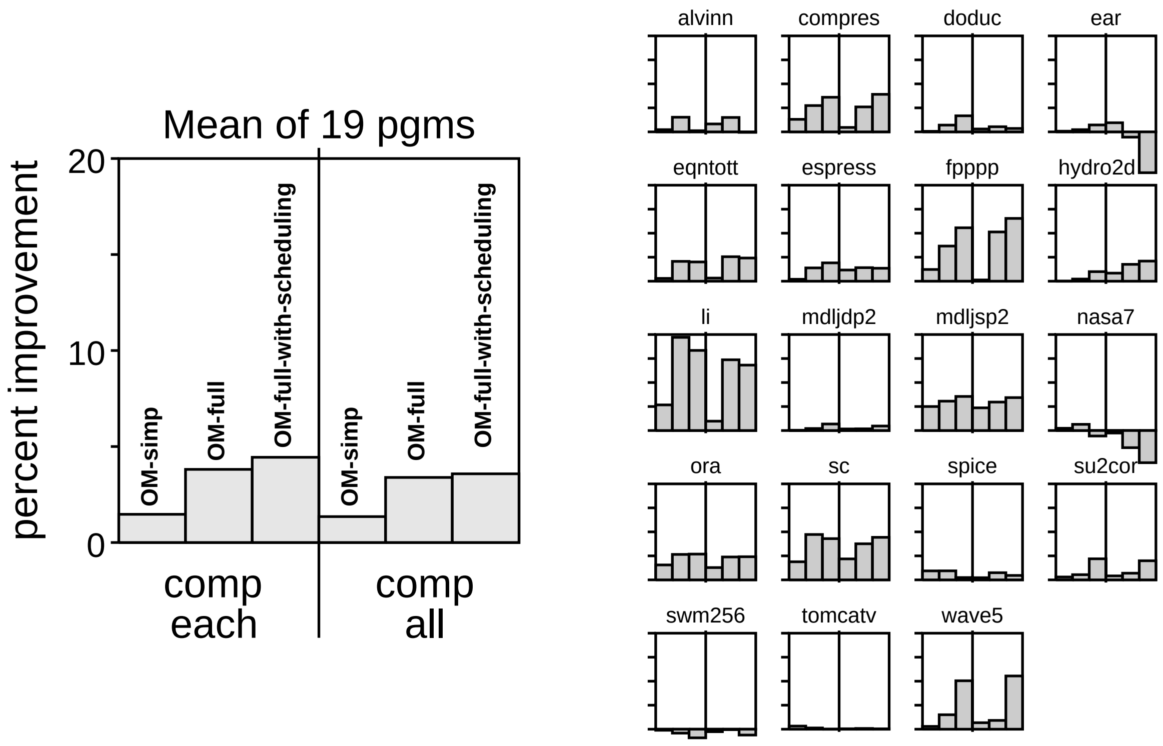


Figure 6: Improvement in performance relative to program without link-time optimization

would greatly improve the performance. Although the code OM starts with was scheduled at compile-time, the scheduling was done in the presence of a large number of address loads that OM later removed. Load operations have a significant latency even when they hit in the cache, and removing that latency allows the instructions that fill it to be used for other kinds of latency. To take advantage of this, OM quadword-aligned instructions that were the targets of backward branches and then scheduled each basic block using a version of the standard AXP/OSF scheduler[6] very similar to the scheduler used by the assembler and thus indirectly by the OSF C compiler.

To our surprise, however, scheduling made only a small difference, raising the average improvement given by OM-full from 3.8% to 4.2% for compile-each code, and from 3.4% to 3.6% for interprocedurally optimized compile-all code. We are still unsure why this is so. It may be that the initial schedule is still good enough even after our changes. Also, the schedulers used by either compiler make some use of memory alias information that is unavailable to us unless we do an expensive data flow analysis, so it may also be that our scheduler is just inferior to the compile-time schedulers.

[6]Thanks to Mike Rickabaugh for providing this.

In two cases, scheduling noticeably degraded performance. In the ear benchmark, the loop-target alignment is at fault; when we scheduled it without alignment the performance was improved. In nasa7 even the unscheduled code was degraded; we suspect this is because deleting unnecessary code caused two parts of the program to fight over the cache.

5.3 Processing time

Figure 7 shows the time required to build each benchmark in a variety of ways. The first column shows the time to do a traditional link using the standard linker on separately compiled object files and standard libraries. The second column shows the time to build the benchmark from its source files, by compiling them with interprocedural optimization and linking them with the standard libraries. The next four columns show the time to link the benchmark using OM on separately compiled object files and standard libraries, with no OM optimization, OM-simple, OM-full, and OM-full with scheduling.

OM's technique of working on a symbolic intermediate form is intended to support a variety of link-time whole-program optimizations. Using OM does introduce some overhead, as we can see by comparing the processing time for OM without optimization to that of a standard link. Nevertheless, OM's processing time is not extravagant. Even OM-full can handle any of these programs in a couple of seconds. A rebuild using interprocedural compiler optimization, in contrast, often takes many tens of seconds.

Link-time scheduling proved to be rather expensive, particularly for applications like fpppp and doduc with very large basic blocks. This is not tremendously surprising; scheduling is usually a superlinear process, since it essentially requires sorting the instructions in a basic block according to a dependency relation. The minor payoff we saw from scheduling suggests that rescheduling is often not worth the cost.

A full compile from sources with interprocedural optimization is nearly always more expensive than OM. This is unsurprising (and not entirely fair), since this kind of compilation does much more analysis and optimization than OM needs for the address-calculation optimizations. Nonetheless, it suggests that address-calculation optimizations are quite cheap and could profitably be done without bringing a full interprocedural optimizer to bear.

	standard link	interproc build	OM linker no opt	OM linker simple	OM linker **full**	OM linker w/sched
alvinn	0.07	2.12	0.28	0.29	**0.33**	1.72
compress	0.06	6.12	0.20	0.20	**0.22**	1.22
doduc	0.28	41.49	1.34	1.43	**1.64**	18.58
ear	0.12	10.12	0.39	0.41	**0.47**	2.59
espresso	0.19	67.84	0.88	0.92	**1.02**	5.46
eqntott	0.08	12.27	0.28	0.29	**0.33**	1.77
fpppp	0.25	16.77	1.19	1.25	**1.43**	68.12
hydro2d	0.26	19.66	1.23	1.29	**1.48**	8.71
li	0.15	54.67	0.55	0.59	**0.64**	3.09
mdljdp2	0.27	9.95	1.23	1.29	**1.48**	8.44
mdljsp2	0.28	9.48	1.23	1.29	**1.47**	8.39
nasa7	0.27	11.38	1.22	1.28	**1.45**	9.03
ora	0.24	2.21	1.03	1.09	**1.26**	7.05
sc	0.21	42.16	0.94	1.00	**1.11**	5.83
spice	0.35	84.38	1.89	2.00	**2.31**	17.90
su2cor	0.27	25.81	1.32	1.40	**1.61**	10.68
swm256	0.24	4.61	1.08	1.14	**1.29**	7.45
tomcatv	0.20	2.95	0.94	0.99	**1.13**	6.48
wave5	0.28	57.86	1.47	1.57	**1.81**	12.95

Figure 7: Build times in seconds for ld from objects, compile from sources with maximum optimization, and OM from objects

6 Discussion

The whole-program optimizations discussed here require us to examine the entire statically-linked part of the program at once.[7] It is not coincidental, therefore, that we describe these as link-time optimizations: only when we are sure that there is no more code to be included can we know whether these optimizations can be performed. The program we examine must contain everything that will be included in the final executable, including any statically-linked library code. Without this assurance, for example, we do not know how big the GAT and data sections will eventually be, and therefore cannot change GAT-based variable references into GP-based references.

Given that we need to examine the entire statically-linked part of the program, it seems quite natural to do it in the linker. There are alternatives, but each has considerable drawbacks. The following sections discuss them in turn.

Monolithic compilation

Monolithic compilation is an old technique with a venerable history. We give the compiler the ability to compile several modules at once, producing a single big object file. To perform whole-program optimizations, the monolithic compile would have to include the sources files for all the modules being linked, including library modules. In other words, we would have to do all our compilation just prior to linking, thus making compile-time and link-time essentially synonymous. Monolithic compilation may be a good approach for some kinds of very aggressive interprocedural optimization, because high level structure like declarations, types, pragmas, and other information about the programmer's intent are readily available. Nevertheless, for whole-program optimizations it has three problems.

First, monolithic compilation can be very expensive. Recompiling all of the source modules can take one or two orders of magnitude more time than even a very ambitious optimizing link. Monolithic compilation might be able to improve the code more than an optimizing linker can, but the address-calculation optimizations we discuss here are very cheap to do at link-time, and a user might quite reasonably want to have them but not be willing to pay the price of a full-blown global optimization.

[7]Though OM does not currently support calls to shared libraries, there is no fundamental problem with doing so, since the code and data for shared routines is dynamically linked in to a different part of the address space than the statically-linked part. Supporting such calls requires maintaining the right symbolic information, but it does not affect our analysis of the statically-linked part. Barring run-time optimization, however, calls to dynamically linked library routines cannot be optimized as statically linked calls can.

Second, monolithic compilation can be awkward. If any libraries are statically-linked, the compilation must include the sources of the library modules. These sources may not even be available to a user, and even if they are it is a genuine burden to have to pick through them finding exactly those library modules needed. In practice libraries are rarely included in monolithic compiles.[8] As a result, monolithic compilation can be used for interprocedural optimization, but not for whole-program optimization.

Third, in some cases it really is easier for a processor outside the compiler to do a complete job. Some compilers include machine-independent phases that are insulated from the machine-dependent parts. Others compile into an abstract assembly language, relying on the assembler for low-level instruction selection and pipeline scheduling. At the other extreme, many programs have modules in more than one language. Even if a monolithic compilation system can accept a collection of modules in different languages and can compile it all the way down to the final machine language, the possibility remains that an unexpected language or compiler will emerge that isn't included in the scheme. A third-party C++ compiler, for example, has no way of benefiting from optimizations that an in-house monolithic compiler performs.

Intermediate-language linking

A second alternative to link-time optimization is intermediate-language linking. At compile-time, individual modules are compiled only as far as an intermediate language. "Object files" are then really IL-files, libraries are full of IL-modules rather than true object modules, and the final code generation is always done at link-time.

In its pure form, this approach is usually expensive as well. Postponing all code generation until link-time shortens compile time but greatly increases link time. This lessens the advantages of separate compilation and makes a fast build nearly impossible even when the user has no need for intermodule optimization. As a result, this approach is rarely followed in its pure form. Two different compromises are common. In one, parallel versions of each library are maintained, one in IL-form and one in object-form, requiring twice the disk space and encouraging inconsistencies and error. In the other, IL-linking is done by recompiling all the user modules into IL-form and IL-linking the results, which reduces this to the case of monolithic compilation.

[8]We followed this practice when we measured OM's improvement of compile-time interprocedurally optimized code: all of the benchmark sources were compiled as a unit, but not the library sources. In fact, we have no sources for the library routines, so we could not have included them in any case. This situation is typical of most users.

Optimistic compilation

A third alternative to link-time optimization is what we might call *optimistic compilation*. To compile a module, we make optimistic assumptions about the results of hypothetical link-time optimizations, and compile the code to match these results. Then all the linker must do is confirm that the assumptions were true. If they were not, the linker reports the fact (and refuses to link) perhaps presenting advice about the switches that will tell the compiler to make less optimistic assumptions. The -G option of MIPS-based compilers is an example of this: modules are compiled such that variables with sizes below a certain default threshold are included in the quickly-accessed "small" sections, but if there are too many such variables the program will not link, and recompilation with a lower threshold is required.

Optimistic compilation has several disadvantages. First, selecting a set of optimistic assumptions is difficult; the best set of assumptions differs from program to program. Picking a default set that are usually satisfied means that final executables seldom run as fast as they could. Programmers whose programs violate the default assumptions are burdened with the need to understand messy details about how the compilers and linker work. In essence it places responsibility on the programmer to make tradeoffs that an optimizing linker is in a better position to make, and requires the development of an interface to let those tradeoffs be made. The programmer's task is complicated further by the likely need to specify different assumptions for different modules.

What's left?

We know of no other alternative approaches in wide use. Link-time optimization has several advantages over these alternatives. First, whole-program optimization can be done in one place, essentially independent of the compiler and source language. Second, even optimizations that do not require the whole program to be present, such as conversion of JSRs to BSRs, can be done more safely by the linker: the compiler must be told whether the compiled code will be statically-linked or included in a shared library, but the linker already knows. Third, statically-linked library code can be included in whole-program optimization in the most natural way, since the linker handles it in exactly the same way that it handles user code. Fourth, when the compiler has important information to offer, it can pass it to the linker through relocation or symbol tables; getting information from the linker back to the compiler is rather messier. Finally, this approach opens the door to other use link-time transformations, such as the movement of very small pieces of loop-invariant code across procedure boundaries [6], or flexible program instrumentation tools [5].

Every program build includes a link step. One naturally wants a linker to be fast under normal circumstances: we probably do not want the linker routinely to perform expensive optimizations that require flow analysis or other “compiler-style” algorithms. The optimizations discussed here are *not* of this kind: they are cheap, fast optimizations that should be available to the user routinely, even during the build-debug-rebuild cycle of a program under development.

7 Conclusions

The conservative address-calculation code that a compiler must generate on 64-bit architectures can be measurably improved by link-time optimization. Even a quite modest approach can make a difference; this approach never moves instructions and therefore fits well in a standard linker. Bigger gains are possible at low cost by being willing to move code. In particular, programs can be made 10 percent smaller and several percent faster by bringing to bear relatively simple analysis of the program’s control structure. Moreover, this machinery in turn enables more sophisticated link-time optimizations [6] that can improve the code still further over what is possible through compile-time optimization alone.

8 Acknowledgements

We are very grateful to all of the Digital compiler people who have helped us with or commented upon this paper. We should mention in particular Mike Rickabaugh and Ken Lesniak for their explanations and help in understanding and tracking the product linker. Mike and Ken began to explore these issues independently of us, and had implemented most of the OM-simple optimizations by the time we heard about it. Kent Glossop gave us valuable information about the GEM compilers, and Mark Himelstein was invaluable in explaining the AXP exception-handling protocols. Russell Kao and Jeff Mogul asked penetrating questions about dynamic linking and the semantics of shared libraries. All of these people, as well as the anonymous PLDI reviewers, made very useful comments on earlier drafts, which we hope we have used to greatly improve the resulting paper. Our thanks to you all.

References

[1] Digital Equipment Corporation. *DEC OSF/1 Programmer's Guide*, section 3.2.3: "Name Resolution." Digital Equipment Corporation, 1993.

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

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

[4] Richard L. Sites, ed. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[5] Amitabh Srivastava and Alan Eustace. ATOM – A System for Building Customized Program Analysis Tools. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, to appear. Also available as WRL Research Report 94/2, March 1994.

[6] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages 1*(1), pp. 1-18, March 1993. Also available as WRL Research Report 92/6, December 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.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

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.