AUGUST 1998

WRL
Research Report 98/5

Efficient Dynamic Procedure Placement

Daniel J. Scales

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

Efficient Dynamic Procedure Placement

Daniel J. Scales

August 1998

Abstract

Commercial applications such as database servers often have very large instruction footprints and consequently are frequently stalled due to instruction cache misses. A large fraction of the i-cache misses are typically due to conflicts in the relatively small direct-mapped on-chip instruction caches. A variety of tools have been developed to try to order the procedures of an application to minimize these conflicts. Such tools often make use of profile information to place procedures so that procedures that frequently call each other do not conflict in the i-cache. However, users often avoid using any kind of tool that requires them to do extra profiling and linking steps to optimize their application. In addition, any tool that does a static layout of procedures (whether using profiling information or not) cannot adapt to varying application workloads that cause very different application behavior.

We have developed a method called DPP (dynamic procedure placement) for placing procedures at run time for good i-cache performance and have implemented it for Digital Unix on Alpha servers. Our system uses a run-time loader similar to a shared-library loader and is completely transparent. Our method efficiently invokes the loader at procedures calls and also correctly handles indirect procedure calls. We have developed a variety of extensions, including run-time code optimizations and a way of restarting the procedure placement while an application is running. In this paper, we describe our method, explain the difficult implementation issues, discuss our extensions, and give performance results for a variety of benchmarks and the Oracle database server. We also provide detailed simulation results using the SimOS-Alpha full-machine simulator.

Efficient Dynamic Procedure Placement

Daniel J. Scales

Western Research Laboratory

Compaq Computer Corporation

scales@pa.dec.com

Abstract

Commercial applications such as database servers often have very large instruction footprints and consequently are frequently stalled due to instruction cache misses. A large fraction of the i-cache misses are typically due to conflicts in the relatively small direct-mapped on-chip instruction caches. A variety of tools have been developed to try to order the procedures of an application to minimize these conflicts. Such tools often make use of profile information to place procedures so that procedures that frequently call each other do not conflict in the i-cache. However, users often avoid using any kind of tool that requires them to do extra profiling and linking steps to optimize their application. In addition, any tool that does a static layout of procedures (whether using profiling information or not) cannot adapt to varying application workloads that cause very different application behavior.

We have developed a method called DPP (dynamic procedure placement) for placing procedures at run time for good i-cache performance and have implemented it for Digital Unix on Alpha servers. Our system uses a run-time loader similar to a shared-library loader and is completely transparent. Our method efficiently invokes the loader at procedures calls and also correctly handles indirect procedure calls. We have developed a variety of extensions, including run-time code optimizations and a way of restarting the procedure placement while an application is running. In this paper, we describe our method, explain the difficult implementation issues, discuss our extensions, and give performance results for a variety of benchmarks and the Oracle database server. We also provide detailed simulation results using the SimOS-Alpha full-machine simulator.

1 Introduction

The code sizes of commercial applications are often extremely large, because they provide a large number of features and extensive safeguards for security and reliability. These applications include personal productivity software, such as word processors and spreadsheets, workflow applications, such as SAP R/3, and database servers. On any given run, these applications will likely execute only a fraction of their code, but the instruction footprint is often still very large. Because of their large footprints, such applications incur frequent instruction cache misses. While some of the cache misses are unavoidable capacity misses, many of the misses are due to conflicts in relatively small on-chip instruction caches which are direct-mapped or have small set associativity.

For example, the Alpha version of the Oracle 7.3 database includes 9 megabytes of instructions. When Oracle runs an on-line transaction processing (OLTP) application similar to the TPC-B benchmark or a decision support (DSS) application similar to the TPC-D benchmark, the instruction footprint is about 1 megabyte. During these runs, instruction stalls can account for 25-30% of the execution time of the Oracle database server on an AlphaServer 4100 multiprocessor [1].

A variety of tools have been developed to try to order the procedures of an application to minimize these i-cache conflicts and reduce stalls. Some tools simply make use of static information such as the structure of the call graph, while other tools use profile information from a previous run that contains the calling frequency between each pair of procedures or even full traces. Various algorithms have been developed to use these types of information to place procedures so as to reduce conflicts [4, 5, 8]. However, users often avoid using profiling tools that require them to do an extra profiling run and re-linking step in order to optimize their application. More importantly, any tool that does a static layout at compile or link time (whether using profiling information or not) cannot adapt to varying application behavior caused by differing inputs. This limitation is especially problematic for applications such as databases which have so many distinct components and features. For example, the code exercised in the Oracle server is quite different when executing an on-line transaction processing (OLTP) application versus a decision support (DSS) application. A fixed procedure layout cannot possibly be optimal for these different workloads.

We have instead developed a method for doing placement of procedures at run time. Our method, which we call DPP (for *dynamic procedure placement*), is completely transparent. The user does not have to do any extra profiling or relinking steps, and the image of the application on disk is left unchanged. In addition, because the layout occurs at run time, DPP produces different layouts for each individual run that are tailored to that run. In the default setup, DPP places procedures consecutively in a text segment (i.e. code segment) as they are called. Therefore, any procedure that is not called in a particular run is never placed in the text segment. In addition, procedures that call each other are likely to be placed near each other. However, we have developed a number of extensions to the default arrangement. We have designed a way of restarting the procedure layout in the middle of a run, so that the procedures used in the core part of the run can be placed together. In addition, we have developed a method of doing automatic profiling at run time, and then using that profile information during the same run to ensure that the most important procedures are placed so as to reduce conflicts.

We have implemented DPP for Digital Unix on Alpha workstations and servers using a run-time loader similar to a shared-library loader. The basic method initially maps the application code in its normal location in the address space, and creates a new empty text segment. The loader then copies the starting procedure of the application to the new segment, but changes all procedure calls to call back to the loader instead. As the application runs, the loader will be invoked whenever a procedure attempts to call another procedure at a new call site. The loader then determines what procedure is being called, and copies that procedure to the new text segment, if not already there, again replacing procedure calls. Complexities such as indirect procedure calls and branches outside of a procedure are also handled. Our method is fully implemented, and we have used it for a variety of complex applications, including an Oracle database.

Chen and Leupen [2] have previously proposed a method for doing procedure placement at run time called "Just-In-Time Code Layout" (JITCL). We have adopted their *activation-order* heuristic of placing procedures consecutively as they are called as our default, but use a fundamentally different approach to catch calls to unplaced procedures. In addition, they did not develop an actual implementation of JITCL, but instead relied on simulation to determine potential performance benefits. Because they only simulated their method, they apparently did not solve several important problems for real applications, such as catching the use of indirect procedure calls. As described above, we have also developed a number of extensions to the basic layout heuristic.

In the following section, we describe our basic method for dynamic procedure placement and our solutions for some difficult problems encountered in real applications. We also describe the limitations of our approach. Next, we describe a number of extensions to our basic method. We then give performance results when using DPP, both on SimOS-Alpha, a complete machine simulator, and on actual machines, for a number of standard benchmarks and for a database application. Finally, we describe related work and conclude.

2 Design and Implementation

In this section, we describe the design and implementation of DPP (dynamic procedure placement). In the first section, we describe the basic method. We then describe the additional problems that must be handled to run complete applications. Finally, we discuss some of the limitations of our method and compare with the approach taken in [2] and other possible approaches.

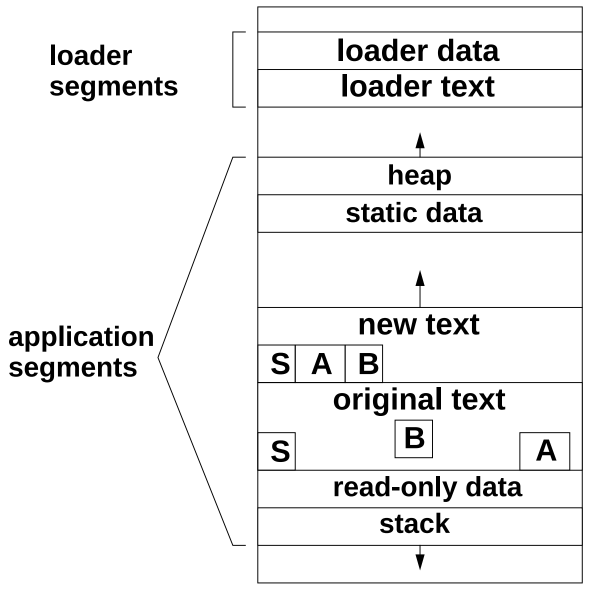

Figure 1: Memory layout for DPP

2.1 Basic Method

Our method makes use of a dynamic loader that is similar to the shared-library loader that is used in most current UNIX systems to link applications with shared libraries at run time. This dynamic loader gains control when an application marked for dynamic procedure placement is invoked. The loader maps in all the appropriate segments of the application, but creates an empty text (code) segment immediately following the original application text segment. Figure 1 illustrates the layout of the application and loader segments. The loader then copies the starting procedure S of the application from the original text segment into the start of the empty text segment. While copying, the loader replaces all procedure call instructions by calls back to itself, and also does all necessary relocations, as indicated by relocation entries in the application executable. The loader then calls the starting procedure at its new location. As the application runs, the loader will be invoked when a procedure A attempts to call another procedure B at a new call site. If B has not been yet been placed, then the loader copies B to the end of the new text segment, again replacing procedure call instructions and doing relocations. In either case, the procedure call instruction at the call site in A is updated to call B directly, if it was originally a direct procedure call. The loader than allows execution to continue. Eventually, the application reaches a steady state in which there is no loader overhead, because all the procedures it is using have been placed and all the call sites have been patched.

The main difficulty with this approach is dealing with indirect procedure calls. Most RISC processors have both direct and indirect procedure call instructions. Though our method is not specific to the Alpha, we will reference these instructions via their name on the Alpha processor: BSR (direct) and JSR (indirect). BSR instructions call a specific procedure, usually specified by a relative offset from the current program counter. JSR instructions call a procedure whose absolute address is specified by the contents of a register. JSRs are frequently used to call specific procedures when the distance between the calling and called procedures is too large or unknown until link time. However, JSRs also provide the flexibility to call many different procedures at a single call site based on the current value of a register. We will refer to a data value that contains the starting address of a procedure and can be used as the argument of a JSR as a *procedure pointer*. Procedure pointers are used in implementing a similar concept in higher-level languages, known as *function pointers* in C and other languages.

Our problem is that any memory location or register can contain a procedure pointer. Therefore, the loader cannot possibly correctly update all procedure pointers that contain the old address of a procedure A so that they now point to the new address of procedure A in a new text segment. Depending on the relocation or type information available in the executable, the loader may know the location of all procedure pointers at the start of the program, but it does not yet know the new locations of procedures, so it cannot update the procedure pointers. The procedure pointers may then be copied arbitrarily into newly allocated data structures before the procedures that they reference are placed in the new text segment.

Our solution is to catch indirect procedure calls to “old addresses” via virtual memory protection. We map the original text into the address space at its default location, but we protect it as non-executable (using the Unix *mmap* system call). We use this mapped region as the source for procedure instructions when copying procedures to the new text region. However, if an indirect procedure call is made using a procedure pointer that references an “old address” in the original text region, then a memory protection fault will occur. Under Digital Unix, the loader can obtain control via a signal handler and then take the appropriate steps to copy the destination procedure, if necessary, and update the procedure pointer. Note that we don't use this method to trap direct procedure calls, because we wish to avoid the faulting overhead in the common case.

We may now give our method in detail as shown in Figure 2. In step 2, the calls to the loader use the same return register as the original call, so no register contents are lost. Different entry points to the loader are used for different return registers. Note also in step 4 that the dynamic loader returns to the call site and executes the newly replaced call instruction, rather than calling procedure B directly. The reason is that the dynamic loader would have to use a JSR to call B, and therefore a register would have to hold B's address. Therefore, that register's contents would not be properly restored to what it was at the call site. Instead, the loader restores all registers and

1. Map all of the text and data segments of the application in their standard locations in the address space, but protect the text segment so that it is not executable.

2. Create a new, empty text segment and copy the starting procedure of the application from the original text segment to the start of the new segment. Replace all procedure call instructions by calls to the loader and do any other necessary relocation changes based on the new location of the procedure.

3. Begin execution of the copied procedure.

4. Whenever the dynamic loader is called from the application, do the following:

- based on the return address, original call instruction, and current register contents, determine the call site A that invoked the loader and the procedure B that the application would have called at that site.
- if B is not already placed in the new text segment, copy B to the current end of the new text segment, again replacing all procedure call instructions and doing all relocations. Also update any statically-known procedure pointers to B in the read-only data segment so that they point to the new address of B. If the original call was a JSR, then also update the contents of the register that will be used by the JSR to be the new address of B.
- replace the instruction at the call site with the instruction of the original application, if it was a JSR, or by a direct call to the new address of B, if it was a BSR.
- return to the original call site and execute the newly replaced call instruction.

5. Whenever the dynamic loader is invoked by a signal handler because of a JSR to an address in the original text segment, do the following:

- based on the PC, original call instruction, and current register contents, determine the call site A that caused the fault and the procedure B being called.
- as above, if B is not already placed in the new text segment, copy B to the current end of the new text segment, making the usual changes.
- update the contents of the register that will be used by the JSR to the new address of B.
- return from the signal handler (which will cause the original call instruction to be re-executed)

Figure 2: Basic Algorithm for DPP

```
    or      zero, zero, r18
L:  ldq     r27, 48(r14)
    ldq     r19, 848(sp)
    ldq     r16, 56(r14)
    jsr     r26, (r27)
```

Figure 3: Sample Code for Indirect Call

returns to the call site via the same return register as the original call instruction. Since that register is immediately used as the return register on the call to B, all register contents will be correct on entry to B.

2.2 Complexities

In this section, we discuss a few more implementation complexities that must be resolved to run complete applications.

First, some applications have code (in just a few places) that actually branches to another procedure without saving any return address. Since the destination procedure may not be placed yet, we must call the loader in this case, but there is no obvious register to use for saving the return address. One solution is to do some dataflow analysis when placing procedures containing such code to determine which registers are not being used. This option can potentially add a fair bit of computational overhead and extra code to the loader. Our approach is to modify the branch to jump to new code added at the end of the procedure that explicitly saves a register before calling the loader. By placing the new code at the end of the procedure, we avoid having to update relative branches within the procedure and we also avoid having the added code interfere with the execution of the procedure after the branch is restored to call the new location of the destination procedure.

We also encountered some applications which has a large number of virtual memory faults because of the repeated use of procedure pointers that the loader had not updated with a new address. This problem typically occurs when the procedure pointer is copied from its original static location to one or more locations in data structures. When the procedure referenced by the procedure pointer is finally placed, the loader cannot update all instances that refer to the procedure. Although the number of such traps still did not cause significant overhead, we were interested in reducing the number of traps where possible. Fortunately, it is usually easy to do simple analysis of the code at a call site to determine where the procedure pointer that is being used is located. Whenever a trap occurs, the loader therefore inspects the instructions preceding the call site. The load instruction L used to load the procedure pointer can almost always be identified. Figure 3 gives an example of some code preceding an indirect call. Registers `r16`, `r18`, and `r19` are loaded with values that are arguments to the called procedure. `r27` is loaded with a procedure pointer that contains the address of the called procedure. `r26` is the return register for the call. Once it has identified the proper load instruction, the loader can calculate the memory location of the procedure pointer and update it with the new address of the procedure, as long as a few conditions hold. First, the registers used in calculating the effective address in instruction L cannot have been modified between L and the JSR instruction. Similarly, no stores can occur between L and the JSR. Finally, there cannot be any branches to instructions between L and the JSR. This method of updating procedure pointers is highly effective in eliminating almost all virtual memory faults once the application has reached a steady state.

Another implementation problem that we encountered was determining the call site when the loader was invoked by a signal. The signal results because a particular call site used a JSR to call an address in the original text segment. However, in Digital Unix, the information provided to the signal handler does not include a direct indication of the call site address. In most RISC systems, a standard register is used to store the return address of the call. Therefore, the signal handler can typically determine the call site based on the contents of the standard return register. We ensure that this method always works by modifying the very few procedures that do not use the standard return register when doing a JSR as the loader copies them to the new text segment.

2.3 Limitations of our Approach

While the large majority of all applications can be handled transparently by our method, there are some restrictions on the applications that can be executed. First, applications that do arithmetic calculations on procedure pointers (or function pointers in C) may not work correctly, since the offsets between various procedure pointers will be different in the original static layout and the dynamic layout. Arithmetic on function pointers is, of course, not a good programming practice, and is allowed only in a few higher-level languages like C that do not have strict type systems.

Second, applications that do comparisons between procedure pointers may not work correctly. Certain procedure pointers may be updated when the referenced procedure is first placed, while other procedure pointers are never updated or only updated later (because they can't immediately be located by the loader). Therefore, a comparison of procedure pointers that really reference the same procedure may fail incorrectly, because one pointer contains the old address of the procedure and the other pointer contains the new address. Again, applications are very unlikely to do comparisons of procedure pointers, since comparison of associated explicit tags would be clearer.

Third, applications that make use of a signal handler to detect memory protection faults may not work correctly, since the loader installs its own handler for memory protection faults. The loader's handler calls any handler installed by the application if the fault does not appear to happen because of non-executable text. Therefore, the likelihood of a problem is minimal.

2.4 Comparison with Alternate Approaches

Chen and Leupen [2] propose a different approach to placing procedures dynamically that they call Just-In-Time Code Layout (JITCL). In their approach, each procedure call is replaced as it is copied with a call to a “thunk routine” that corresponds to the procedure being called. A thunk for a particular procedure is a piece of code that copies the procedure into the new text segment if the procedure has not been placed yet, patches the call site to call the procedure at the new address, and then jumps to the new procedure. The use of customized thunk code for each different procedure may potentially reduce some of the overhead of our approach, in which all procedure calls initially enter the loader and then the loader determines which procedure was called based on register contents. On the other hand, the thunk approach has extra overhead in creating the thunks and a bigger instruction footprint because of all the different thunks (which may each be 20-30 instructions) that are executed.

More importantly, this approach does not work at all in the presence of indirect procedure calls, where the procedure being called cannot be determined statically. The procedure call cannot be directed to call a particular thunk, and, just as importantly, the call site cannot be patched to call the destination procedure directly. Since most current systems (such as Digital Unix) use procedure pointers both for initialization and some standard library routines, this approach would not work for any applications, even applications that did not explicitly use procedure pointers.

However, one way of modifying the thunk approach to deal with indirect procedure calls is as follows. We place the thunks at the original address of the procedures that they represent. Therefore, we do not have to change procedure calls at all when procedures are copied to their new location (except for appropriately updating relative offsets in BSRs). When first encountered, procedure calls still call procedures at their original address and therefore invoke the appropriate thunk. The thunk executes as described above, except that it only patches the call site to call the procedure at its new location for direct procedure calls. Indirect procedure calls are handled correctly, since they will always call the thunk at the original procedure address, and the thunk will jump to the procedure at its new location. The disadvantage is that indirect procedure calls will always incur the overhead of the thunk code, both in terms of executing extra instructions and extra occupancy of the cache and TLB.

In addition, there is extra overhead in creating the thunks. A naive approach would create a thunk for every procedure at startup time However, that method would cause extensive initialization overhead, as a large number of virtual memory pages were touched to create thunks for many procedures that were never used. A more sophisticated approach would use virtual memory protection to allow thunks on a particular virtual memory page to be created only when a procedure on that page was first called. For most processors (including Alpha), it may also be necessary to add additional bridge code so BSR calls in the new text segment, with their limited range, can actually reach the thunks in the original segment.

Note that the first two limitations described in the Section 2.3 are avoided with this modified approach. These limitations arise because we modify procedure pointers to point to the new location of a procedure when possible. In this approach, we never modify procedure pointers, so arithmetic and comparisons on procedure pointers operate as expected. However, we decided that the added overhead of the thunk approach was likely to be much too high to justify using it to eliminate these two minor restrictions.

3 Extensions

In this section, we describe a number of extensions to the basic DPP system that we have implemented.

3.1 Code Optimizations

Given that we are already copying code at run time, it is fairly easy for us to do a number of runtime code optimizations to further improve the performance of the application. One optimization is to replace JSR instructions by BSR instructions when possible, because procedures that call each other are now closer together. JSR instructions are not only used for indirect procedure calls, but also for direct calls that cannot use BSR instructions because the two procedures are too far apart in the virtual address space. When we use DPP, most procedures that call each other are typically placed much closer, and therefore may be able to use a BSR call, even though they required a JSR call in the original application. The main difficulty with this optimization is determining if a JSR call is a simply a direct call between two far-apart procedures, or really an indirect procedure call. Under Digital Unix, the loader is usually able to distinguish these two cases by trying to determine from what location the procedure address used by the JSR is loaded. If the loader can determine the location, and that location is in a read-only segment of address constants, then the call is really a direct call to a fixed procedure. Otherwise, the loader does not do the optimization. In the case when the loader can convert the JSR to a BSR, it may also be able to eliminate (replace with no-ops) some instructions, such as the instruction that loads the procedure address used by the JSR.

When we cannot eliminate the JSR, we can still try to improve its behavior on some processors. On the Alpha, the JSR instruction includes a field which gives a “hint” of the low-order bits of the destination address, in order to avoid delays in fetching the next i-cache line. For an indirect procedure call, the compiler cannot fill in this hint, since it has no idea what procedure might be called. However, when the loader is invoked for a call site that is doing an indirect procedure call in the original application, the loader can fill in the hint with a value corresponding to the actual target procedure on this call. In the cases where an indirect call actually calls the same procedure over and over, filling in the hint will speed up later calls at the same call site.

3.2 Restarting Procedure Placement

Although DPP readily adapts to different uses of an application, we have developed a method of restarting procedure placement that allows DPP to adapt better to changing behavior of an application during a run. This technique can be useful for almost any application, since most applications include an initialization phase. The initialization phase will often execute procedures that are only used during initialization and may cause some procedures that will no longer be used to be placed among procedures that will often be used. It is therefore useful to restart the procedure placement after initialization. This technique is also useful for applications which may have long phases of very distinct behavior. In this case, it may be beneficial to restart the placement whenever a new phase starts.

The method for doing a restart is as follows:

1. Change the protection of the current text segment so it is non-executable and create another empty text segment that is executable.

2. Reset the loader's table of procedure mappings, so that all procedures will be placed again in the new text segment.

3. Copy the current procedure to the new text segment, again replacing all procedure calls to invoke the loader and doing the necessary relocations.

4. Restart execution at the appropriate spot in the new copy of the current procedure.

When the loader is invoked by a call or a signal, it handles these events as in steps 4 and 5 in Figure 2, with some extensions. First, it is now possible that a signal will occur because of call through a procedure pointer that references the previous text segment (not the original application text). Since that segment has now been made non-executable, a fault will occur and the loader must maintain information so it can determine which procedure was being called. In addition, there must be some method of handling procedure returns, since the return addresses on the stack refer to the previous text segment. The simplest method is to again use virtual memory protection. When a procedure return attempts to jump to an address in the previous text segment, the loader will be invoked via the signal handler. The loader must then distinguish the case of a return from the case of an indirect procedure call, both of which can cause a signal. If the signal was caused by a return, then the loader copies the returned-to procedure to the latest text segment, as necessary, and then returns to the appropriate location in the new copy of the procedure. Unfortunately, under Digital Unix, it is impossible to distinguish the two reasons for a signal in all cases. Therefore, our implementation actually replaces all return instructions of the current procedure with extra code that invokes the loader at a special entry point. When this entry point is called, the loader copies the returned-to procedure to the latest text segment, again replacing its return instructions. In this way, only returns by procedures on the stack at the time of the restart are trapped.

We currently allow the user to specify when a restart should occur in two different ways. First, if procedure names are available in the application, the user can specify the name of a procedure which should cause a restart when it is called for the first time. Second, the user can specify that a restart should occur when a particular signal (the UNIX "hangup" signal in our current implementation) is sent to the application.

Our loader also provides a useful option in which the original unmodified application code is executed until the hangup signal is sent to the application. At that point, the application code is protected as non-executable and the procedure placement proceeds as described in Section 2.1. In this way, the overhead of the procedure placement can be completely avoided during the initialization and early phases of an application, when many procedures that are called only once are executed and DPP is likely to be least effective. DPP can then be started and the minimal number of procedures placed once the application is in its "main loop".

3.3 Run-time Profiling

We have also developed an extension of DPP that records profiling information dynamically during an application run and uses that information to produce a better procedure layout during the *same* run. We simply start the loader in a mode where it leaves unchanged the call to the loader at the application call site whenever it is invoked. The loader is therefore invoked on every single procedure call, and it records a profile of the procedure calls. After the certain amount of time or a signal from the user, the loader uses the profile information to place the most important procedures in an optimized way in a new text segment. It can use any desired static placement algorithm, such as those described in [4, 5, 8]. It then continues in a normal way, so that any other procedures that are called are placed as usual in a dynamic way. Because of the profiling information, the static algorithm can do a better job than the simple dynamic algorithm in ensuring that the most important procedures are placed so that they do not conflict.

During the profiling process, execution speed can be slowed down by a factor of ten or more, but the profiling is clearly worthwhile if it is used only during a small warmup period and improves the performance of a production application that will run continuously for a much longer time.

We can potentially control these restarts or profiling periods using an adaptive timing method. For example, we can periodically do restarts on a long-running production application (say, once an hour). If the set of procedures that are placed in the current restart period is similar to the set of procedure placed in the preceding period, then the application has been executing similar code, so we increase the time between restart periods. If the sets are very different, then the application has been doing different kinds of computations (perhaps of different phase of a workload), so we should either decrease the restart period or keep it the same.

4 Performance Results

In this section, we provide performance results for applications using DPP. We first describe some of the expected performance effects and our experimental setup, and then give both simulated results obtained using SimOS-Alpha, a full machine simulator, and actual results for runs on an AlphaServer.

4.1 Expected Effects

Although we expect performance benefits from the use of DPP, there are many conflicting effects that are involved. The main expected performance benefit is a reduction in first-level i-cache misses. We would also expect a decrease in iTLB faults, since the procedures actually used by the applications are packed together on fewer code pages than in the original executable. The resident set of the application should also decrease, thereby likely improving performance when it executes in a heavy timesharing mix of applications.

However, DPP has a large initialization cost, when it reads in and sorts symbol and relocation information from the application executable. The loader also introduces instruction overhead and pollutes the caches with its instructions and data each time it is invoked at a new call site. DPP will therefore provide a performance benefit only if the application executes for a long time in a steady state in which it is reusing largely the same code. Note also that DPP touches every instruction page that the original application touches, since it must copy each active procedure from its original code page to the new segment. In fact, it may touch more of the application code pages, since it copies whole procedures, and it additionally must touch the code pages of the new text segment. Again, a decrease in iTLB faults will only occur if the application executes for a long time with good instruction locality. Since DPP cannot control any misses caused by kernel code, TLB fault handlers, etc., DPP may also not help applications which have a significant amount of system time.

An interesting effect of DPP is that there may be more conflicts between instructions and data in the combined second-level cache, because the instructions are making more effective use of that cache. Because of DPP, there may be fewer conflicts between instructions in the second-level cache. There will then tend to be more instructions resident in the second-level cache and therefore potentially more conflicts with data. This effect may be especially pronounced for the Alpha 21164, which has an on-chip combined 96 Kbyte second-level cache.

Finally, unfortunately, DPP can have negative effects on performance because of chance placement decisions. For example, two crucial procedures that were not conflicting in the original static layout may happen to conflict in the dynamic layout, even though the rest of the dynamic layout is better for i-cache behavior. As another example, the consecutive layout of procedures in the new text segment may happen to conflict badly in the second-level cache with a particular application data structure.

4.2 Experimental Setup

Our application set includes three applications from the SPLASH-2 parallel application suite [11] (but always run on one processor), four applications from the SpecInt95 collection, and the Oracle 7.3 database server running a decision support (DSS) query. The applications are listed in Table 1, along with the number of procedures and code length in the second and third columns. For RT (RayTrace), we use two input sets (teapot and balls4), because the cache behavior for the two inputs is quite different. For the first four applications and Oracle, we isolated the initialization and finalization phases and therefore can give statistics on just the main part of the application. The Oracle execution is for a particular decision support query, which is repeated several times after database initialization. We measure the query after it has already been executed once, so the software database cache is fully warmed up and no disk operations are required after the restart.

We have implemented the DPP method and extensions described in the previous section for Alpha servers running Digital Unix. However, our implementation does not correctly handle applications with kernel threads and does not currently work with shared libraries. All of the applications are therefore statically linked. All applications are compiled with maximum optimization available under the Digital Unix compiler, excluding profiling optimizations.

We do our simulated runs using SimOS-Alpha, a port of SimOS to the Alpha. SimOS [9] is a full machine simulator that completely simulates the available hardware and can run complete operating systems and application workloads without any changes (except possibly for minor changes to device drivers). SimOS fully simulates all activity on the machine, including user processes, kernel activity, TLB fault handling, etc. In our runs, we simulate a 200 MHz Alpha processor with separate 8 Kbyte direct-mapped first-level instruction and data caches and a 2-way associative 1 Mbyte board cache. The processor has a 48-entry instruction TLB and a 64-entry data TLB.

We are able to run our DPP loader on SimOS without any changes to the loader or to SimOS. Most other user-level simulators would require extensive changes to simulate dynamic procedure placement, since they assume a fixed layout of procedures as indicated by the executable and they do not typically allow an extra executable (the loader) to be mapped into the application's address space. In addition, SimOS correctly handles many events, such as signals, memory mapping, cache flushes, etc., that are often not allowed or fully simulated by other simulators.

We do real performance runs on an AlphaServer 4100. The AlphaServer has four 300 MHz 21164 processors, which each have 8 Kbyte direct-mapped on-chip instruction and data caches, a 96 Kbyte 3-way associative on-chip combined second-level cache, and a 2 Mbyte direct-mapped board-level cache. Like the simulated processor above, the Alpha 21164 has a 48-entry instruction TLB and a 64-entry data TLB. The individual processors are rated at 8.1 SpecInt95 and 12.7 SpecFP95, and the system bus has a bandwidth of 1 Gbyte/s.

4.3 Application Results

In Table 1, we list the applications and give statistics for static and dynamic placement of procedures. The second and third columns give the number of procedures and code length (in bytes) of the statically linked executable. The fourth column gives the number of 8 Kbyte code pages in the static executable that are touched during execution. The fifth and sixth columns give the number of procedures actually placed by standard DPP during a full run and the total length of these procedures. For most applications, less than 50% of the procedures and 50% of the code in the executable are actually used, while in the Oracle run, less than 20% of the procedures and code are used. The seventh column gives the number of code pages in the new segment touched with dynamic placement (i.e. roughly the sixth column divided by 8192). In comparison with the fourth column, the “page footprint” is much smaller for all applications when DPP is used. Many pages in the original code segment are also touched during layout as procedures are copied to the new segment, but these pages are rarely touched once all the necessary procedures have been placed.

	Static			Dynamic					Restart	
	no. of procs	**code length**	**pages touched**	**no. of procs**	**code length**	**pages touched**	**no. of traps**	**no. of faults**	**no. of procs**	**code length**
LU	379	163840	16	93	45836	6	166	4	10	4428
water	408	221184	20	131	87396	11	354	6	36	39916
RT/teapot	524	262144	25	203	108604	14	476	113	42	21180
RT/balls4	524	262144	25	203	105796	13	464	7394	42	19964
m88ksim	607	278528	29	204	108852	14	485	17	90	62428
perl	674	532480	49	175	244148	30	550	4	NA	NA
gcc	2282	1630208	175	1080	1078260	132	6544	70	NA	NA
go	654	360448	41	409	265348	33	1826	5	NA	NA
Oracle/DSS	12670	10330112	635	2479	2231204	273	8235	470	465	503812

Table 1: Applications and Placement Statistics

The eighth and ninth columns give the number of traps (normal calls to the loader) and faults (calls to the loader via the signal handler). The number of traps is essentially the number of call sites that are encountered, since a trap can only happen once at each call site. The number of faults is quite low for all applications except RT (RayTrace). RayTrace makes extensive use of function pointers for manipulating various kinds of geometric objects, and the function pointers used at one particular call site cannot be patched, as described in Section 2.2, because of a modification to the register used to load the function pointer. With some additional analysis, the code at this call site could potentially be modified so that the address of the function pointers could be determined properly at run time.

The last two columns of Table 1 give the number of procedures placed (and their total length) in the main part of the application when the loader does a restart after application initialization. Clearly, a large number of procedures are only used for initialization and finalization, and the code being used in the main part of the application is often quite small. Hence, restarting the placement after initialization should improve the effectiveness of DPP by ensuring that the procedures used by the main computation are placed near each other.

Table 2 gives the results of simulating the applications on SimOS. The first row for each application is for a normal run and the second row is for a run using DPP. As mentioned above, the statistics for perl, gcc, and go are for entire runs, while the statistics for the remaining applications exclude initialization and finalization phases (which often include I/O and other interaction with the operating system.) All statistics include all activity on the machine, including TLB faults, kernel activity, clock interrupts, etc. The statistics for the normal runs show that LU, Water, and RT/balls4 have very good i-cache performance, while the remaining applications show extensive

	Instruction				Data				Execution Time
	L1 stall time	L2 stall time	Stall cycles	TLB faults	L1 stall time	L2 stall time	Stall cycles	TLB faults	
LU	0.27% 0.22%	0.06% 0.07%	1.9M 1.7M	19 49	29.7% 29.8%	4.8% 4.8%	201M 202M	107K 107K	2.914s 2.915s
Water	1.72% 0.87%	0.01% 0.01%	9.1M 4.6M	10 33	8.6% 8.7%	0.01% 0.01%	45.2M 46.0M	289 707	2.635s 2.615s
RT/teapot	10.69% 8.83%	0.20% 0.06%	42.4M 33.8M	750 489	23.0% 23.4%	1.05% 1.18%	93.4M 93.4M	174K 174K	1.945s 1.901s
RT/balls4	0.77% 0.65%	0.98% 1.22%	324M 350M	2824 1655	17.4% 17.3%	40.8% 41.1%	10814M 10909M	8.59M 8.59M	92.78s 93.37s
m88ksim	21.1% 19.8%	0.00% 0.00%	22144M 20473M	22.4K 10.0K	7.80% 7.93%	0.09% 0.09%	8292M 8279M	916K 911K	525.0s 516.5s
perl	18.7% 12.1%	0.00% 0.00%	5252M 3130M	296 486	15.5% 16.8%	0.00% 0.01%	4359M 4356M	288 3323	140.3s 129.8s
gcc	20.2% 19.8%	1.05% 1.25%	708M 730M	188K 73K	16.0% 15.6%	5.0% 4.9%	699M 711M	956K 1021K	16.63s 17.35s
go	11.9% 13.2%	0.03% 0.04%	6687M 7577M	533 1224	33.7% 33.2%	0.03% 0.06%	18937M 18961M	61614 86724	280.7s 285.4s
Oracle/DSS	21.0% 15.8%	0.11% 0.13%	675M 475M	3000 4921	14.8% 15.2%	4.3% 4.6%	611M 591M	125574 125701	16.01s 14.92s

Table 2: Simulated Application Performance (normal vs. with DPP)

misses in the first-level i-cache.

The stall cycles due to i-cache misses go down by 8-50% when using DPP for all applications except RT/balls4, gcc, and go. The overall execution time also goes down for these same applications, but mostly by only 2-3%. (For LU, the execution time essentially stays the same.) However, the execution time does go down significantly (7-8%) for perl and Oracle.

For RT/balls4, the increased instruction stall cycles seem to be due to a conflict with the application data in the second-level cache, since the time due to L2 instruction misses increase. The application accesses a huge amount of data, as indicated by the large number of data TLB faults. This result is confirmed by an additional simulation of RT/balls4 in which the second-level cache has 4-way associativity. In this run, the instruction stall cycles go down from 179M to 164M. In gcc, the increased instruction stall cycles is probably due to the execution of the loader itself. The loader time is less that 0.1% of the total run time for all applications except RT/teapot, where it is 0.26%, and gcc, where it is 1.39%. The high loader time in gcc is due to the large number of traps to the loader, which in turn is due to the large number of distinct call sites that are executed during the different passes of the compiler. The repeated execution of the loader during the short gcc run is likely to cause cache pollution that affects the performance of the application. It is not obvious why the instruction stall cycles and run time increase for go; these effects may be due to some conflict between crucial procedures that happens to be introduced by the dynamic placement.

Although largely insignificant to the execution time, the stall cycles go down dramatically for water. A large part of this effect is due to removing a conflict between a crucial application routine and a library routine, *fabs()*, that computes the absolute value of a floating-point value. This routine is placed far away from the application routines that use it by the linker, because it is a library routine, but is automatically placed near these routines by DPP.

The number of iTLB faults goes down dramatically for m88ksim and gcc, both applications that have large numbers of iTLB faults. The dramatic decreases in iTLB faults are likely due to the fact that the active procedures of these applications are spread out over a large number of pages in the static executable, while DPP places all active procedures together on a much smaller set of pages. The iTLB faults increase somewhat for perl, go, and Oracle (but hardly affect the execution time), possibly because of execution of the loader itself.

In Table 3, we give the actual run times for the applications on our AlphaServer when using static and dynamic placement. For Oracle, we give results when using one, two, three, and four server processes, respectively, to do the decision support query in parallel. As before, the results for perl, gcc, and go are for the entire application run, while the results for the other applications are only for the main part of the application. The obvious result is that times go down or stay roughly the same in almost all cases, and the Oracle runs have quite significant decreases (6-10%) in execution time. However, the changes are rather small for the other applications. On the real machine, there may be more adverse effects due to data and instruction conflicts in the second-level on-chip 96 Kbyte cache.

m88ksim provides one example of the usefulness of the facility for restarting procedure placement. This benchmark simulates the execution of two different programs, and we consider the second program, the Dhrystone benchmark, to be the main computation. When we restart the procedure placement just before simulating the Dhrystone benchmark, the run time goes down from 170.31 seconds to 156.36 seconds. The restart is improving performance because the mix of procedures used in simulating the first program is quite different from the mix used in simulating the Dhrystone benchmark. As we said above, a restart is likely to be beneficial when an application starts a new phase of computation with very different behavior.

	Static Placement	**Dynamic Placement**
LU	1.610s	1.562s
water	2.205s	2.188s
RT/teapot	1.581s	1.585s
RT/balls4	67.18s	65.13s
m88ksim	171.29s	170.31s
perl	69.93s	68.45s
gcc	10.40s	10.86s
go	150.48s	151.63s
Oracle/DSS-1	8.77s	8.24s
Oracle/DSS-2	4.91s	4.44s
Oracle/DSS-3	3.28s	2.99s
Oracle/DSS-4	2.48s	2.30s

Table 3: Application Run Times

5 Related Work

There has been a variety of work [2, 3, 4, 5, 6, 7, 8, 10, 12] in improving the instruction cache behavior of user applications, with most of it focusing on statically laying out the instructions of an application using profile information from a previous run. Samples and Hilfinger [10] did early work in using edge frequencies between basic blocks to lay out basic blocks, while McFarling [7] used basic block frequency counts. Pettis and Hansen [8] developed a simple algorithm for placing procedures based on procedure frequency counts that is frequently used. Several more complex but potentially more effective algorithms [4, 5, 6] have been developed recently, including some that make use of a full trace of procedure calls. Most of these systems have been run only on user-level simulators, and the authors report the changes in instruction miss rates, but not changes in execution time. Several systems [3, 12] have recently been developed to manage automatically the collection of profiles and the re-optimization of user executables.

Our system is most similar to the Just-In-Time Code Layout of Chen and Leupen [2]. However, as we describe in Section 2.4, their approach does not work for indirect procedure calls, which are used in almost all applications, at least in initialization and library code. Because they did not do an actual implementation, they did not necessarily address the detailed implementation issues that arise. In addition, their simulation results were obtained by instrumenting the user applications, and therefore probably do not model the interference of loader and kernel code with the application code in the i-caches. It is not clear if they modeled the behavior of the TLBs. Our simulations using SimOS-Alpha model the entire system, including the execution of the unmodified loader code, and we have reported performance results for real systems as well.

6 Conclusions

We have described a method called DPP (dynamic procedure placement) for laying out the procedures of an application at run time for better instruction cache behavior. Our method is fully implemented and transparently runs on a variety of applications, including the Oracle 7.3 database server. It handles a number of difficult problems that arise with real applications, including the use of indirect procedure calls. We have simulated the effects of DPP on a number of applications using SimOS-Alpha, a complete machine simulator, and have also measured actual performance results. Our method usually reduces instruction stall cycles significantly (by 8-50%) for the applications we have simulated, and typically reduces the execution time as well, but to a lesser degree. We also achieved notable improvements (6-10%) in the actual run time for the Oracle database executing a decision support workload. Most other recent work in this area has only reported simulation results, rather than changes in application run times on real systems.

By default, our system simply places procedures consecutively in a new text segment in the order in which they are called. However, we have developed an extension in which the layout is restarted during an application run, so that the procedures that are used in the main part of the application are more likely to be placed closed together. This extension significantly improved the performance of one of our applications that had two distinct phases of computation. We have also implemented a mode in which procedure calls are transparently monitored for a period of time, and then, during the same run, the profiling information is used to layout the most important procedures. In the future, we intend to investigate additional run-time layout heuristics. For instance, we could make use of cache size information to reduce conflicts even when important procedures cannot be placed near each other. Also, we may want to place very large procedures in a separate segment, since they are guaranteed to conflict with other procedures in the first-level caches. Finally, it may be useful to dynamically clone some small but important procedures at run time if it is otherwise not possible to place the procedures so as to avoid conflicts with some of their callers.

References

[1] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commerical Workloads. In *Proceedings of the 25th International Symposium on Computer Ar-*

chitecture, June 1998.

[2] J. B. Chen and B. D. D. Leupen. Improving Instruction Locality with Just-In-Time Code Layout. In *The 1997 USENIX Windows NT Workshop*, Aug. 1997.

[3] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An Optimizer for Alpha/NT Executables. In *The 1997 USENIX Windows NT Workshop*, Aug. 1997.

[4] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure Placement Using Temporal Ordering Information. In *30th International Symposium on Microarchitecture*, pages 303–313, Dec. 1997.

[5] A. Hashemi, D. Kaeli, and B. Calder. Efficient Procedure Mapping Using Cache Line Coloring. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[6] J. Kalamatianos and D. Kaeli. Temporal-Based Procedure Reordering for Improved Instruction Cache Performance. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 244–253, Feb. 1998.

[7] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[8] K. Pettis and R. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[9] M. Rosenblum, E. Bugnion, S. A. Herrod, and S. Devine. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[10] A. D. Samples and P. N. Hilfinger. Code Reorganization for Instruction Caches. Technical Report UCB/CSD 88/447, University of California, Berkeley, Oct. 1988.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[12] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System Support for Automatic Profiling and Optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

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.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’ **Steven J.E. Wilton and Norman P. Jouppi.** WRL Research Report 93/5, July 1994.

‘‘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.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’ **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.** WRL Research Report 93/8, December 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.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’ **Keith I. Farkas, Norman P. Jouppi.** WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’ **Jeffrey C. Mogul.** WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo, Herve Touati.** WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’ **J. Bradley Chen, David W. Wall, and Anita Borg.** WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’ **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.** WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’ **Keith I. Farkas, Norman P. Jouppi, and Paul Chow.** WRL Research Report 94/8, December 1994.

‘‘Drip: A Schematic Drawing Interpreter.’’ **Ramsey W. Haddad.** WRL Research Report 95/1, March 1995.

‘‘Recursive Layout Generation.’’ **Louis M. Monier, Jeremy Dion.** WRL Research Report 95/2, March 1995.

‘‘Contour: A Tile-based Gridless Router.’’ **Jeremy Dion, Louis M. Monier.** WRL Research Report 95/3, March 1995.

‘‘The Case for Persistent-Connection HTTP.’’ **Jeffrey C. Mogul.** WRL Research Report 95/4, May 1995.

‘‘Network Behavior of a Busy Web Server and its Clients.’’ **Jeffrey C. Mogul.** WRL Research Report 95/5, October 1995.

‘‘The Predictability of Branches in Libraries.’’ **Brad Calder, Dirk Grunwald, and Amitabh** Srivastava. WRL Research Report 95/6, October 1995.

‘‘Shared Memory Consistency Models: A Tutorial.’’ **Sarita V. Adve, Kourosh Gharachorloo.** WRL Research Report 95/7, September 1995.

‘‘Eliminating Receive Livelock in an Interrupt-driven Kernel.’’ **Jeffrey C. Mogul and K. K. Ramakrishnan.** WRL Research Report 95/8, December 1995.

‘‘Memory Consistency Models for Shared-Memory Multiprocessors.’’ **Kourosh Gharachorloo.** WRL Research Report 95/9, December 1995.

‘‘Register File Design Considerations in Dynamically Scheduled Processors.’’ **Keith I. Farkas, Norman P. Jouppi, Paul Chow.** WRL Research Report 95/10, November 1995.

‘‘Optimization in Permutation Spaces.’’ **Silvio** Turrini. WRL Research Report 96/1, November 1996.

‘‘Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.’’ **Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath.** WRL Research Report 96/2, November 1996.

‘‘Efficient Procedure Mapping using Cache Line Coloring.’’ **Amir H. Hashemi, David R. Kaeli, and Brad Calder.** WRL Research Report 96/3, October 1996.

‘‘Optimizations and Placement with the Genetic Workbench.’’ **Silvio Turrini.** WRL Research Report 96/4, November 1996.

‘‘Memory-system Design Considerations for Dynamically-scheduled Processors.’’ **Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic.** WRL Research Report 97/1, February 1997.

‘‘Performance of the Shasta Distributed Shared Memory Protocol.’’ **Daniel J. Scales and Kourosh Gharachorloo.** WRL Research Report 97/2, February 1997.

‘‘Fine-Grain Software Distributed Shared Memory on SMP Clusters.’’ **Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal.** WRL Research Report 97/3, February 1997.

‘‘Potential benefits of delta encoding and data compression for HTTP.’’ **Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy.** WRL Research Report 97/4, July 1997.

‘‘Neon: A (Big) (Fast) Single-Chip 3D Workstation Graphics Accelerator.’’ **Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll.** WRL Research Report 98/1, August 1998.

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.

‘‘Characterization of Organic Illumination Systems.’’ **Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.

‘‘Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.’’ **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.

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

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

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

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

‘‘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.

‘‘Piecewise Linear Models for Rsim.’’ **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.

‘‘Speculative Execution and Instruction-Level Parallelism.’’ **David W. Wall.** WRL Technical Note TN-42, March 1994.