DECEMBER 1990

WRL
Technical Note TN-17

MTOOL: A Method For Detecting Memory Bottlenecks

Aaron Goldberg
John Hennessy

digital **Western Research Laboratory** 100 Hamilton 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 technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

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

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton 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`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`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.

MTOOL: A Method For Detecting Memory Bottlenecks

Aaron Goldberg
Digital Equipment Corporation Western Research Laboratory

John Hennessy
Stanford University

December, 1990

Abstract

This paper presents a new method for detecting regions of a program where the memory hierarchy is performing poorly. By observing where actual measured execution time differs from the time predicted given a perfect memory system, we can isolate memory bottlenecks. MTOOL, an implementation of the approach aimed at Fortran programs running on MIPS-chip based workstations is described and results for some of the Perfect Club and SPEC benchmarks are reported.

This research was supported in part by an NSF Fellowship.

1 Introduction

Many modern computer architectures including cache-based uniprocessors and most shared memory multiprocessors present the programmer with a (deceptive) uniform access model of memory. RISC architectures for example provide simple load and store instructions to represent memory operations. In practice, however, the load instruction may involve accessing on-chip cache which is backed by a second level cache of static RAM which is backed by main memory. The time difference between a hit in the first level cache and a miss to main memory can be one to two orders of magnitude (see Table 1).

Programmers seeking to improve performance sometimes find it useful to optimize an algorithm with respect to a particular memory hierarchy. Studies like [1], [2], [7] and [8] have reported speed-ups of 100% and more owing to improved cache performance when nested loops are reordered and matrix algorithms are blocked. To make use of such techniques, the user must know where memory bottlenecks lie and when a transformation improves performance.

Two techniques are typically employed to isolate memory bottlenecks. The least time consuming approach is to statically analyze a program, using dependency analysis to identify how many items will be in cache after a certain number of iterations of a loop [3, 6]. Such techniques are important because they are potentially fast enough to incorporate into compilers to automatically manage transformations. However, the static techniques rely on the simple structure of both the loop and the memory system to perform their analyses. The approximate nature of analytic techniques and the in-

	Miss Penalty (cycles)		
Machine	Primary	Secondary	Remote
DEC 3100	6	–	–
DEC 5000	10	–	–
SGI (4 node)	14	40	–
MIPS 6280	2-4	50	–
Stanford DASH	14	28	103-136

Table 1: Memory Hierarchy Penalties

creasing complexity of the memory hierarchies they attempt to model make them inappropriate as a complete performance debugging solution.

At the opposite end of the spectrum, there are trace driven simulators which simulate the execution of every memory reference in a program. These simulations can model the entire memory hierarchy and produce access time estimates for every variable reference in a program. The obvious drawback of simulation is its cost; for large programs, simulation may be quite expensive. Also, it is non-trivial to correctly model a complicated memory hierarchy, particularly when multiprogramming or multiprocessing is involved.

Our technique strikes a balance between the expense of simulation and the inaccuracy of static analysis. Our key observation is that if we assume memory access time is uniform then, at least for simpler architectures, it is relatively cheap to correctly estimate the CPU execution time of a program. By comparing this uniform access model estimate with actual observed execution time, we can isolate regions in a program where the memory hierarchy performs poorly.

The next section discusses the method for estimating execution time assuming constant memory access time. Section 3 describes how to use the estimate to isolate memory bottlenecks. Section 4 presents a memory bottleneck tool implementation, MTOOL, which runs on the DECstation 3100 and 5000. Section 5 provides examples of MTOOL's user interface. Section 6 reports results when the tool is run on the Perfect Club benchmarks and scientific benchmarks in the SPEC set. Finally, section 7 gives some conclusions about the breadth of applicability of our technique and suggests directions for future research.

2 Estimating Execution Time

Consider a computer where all instruction scheduling is handled by software (i.e., no hardware interlocks) and where each instruction (including memory access instructions) has a known, fixed execution time. For such a computer, we can determine the execution time of a program given instruction execution counts using the formula,

$$\begin{aligned} \textit{execution time} = {} & \textit{(\# of times ith instruction executes)} \; * \\ & \textit{(time per execution of instruction i)} \end{aligned}$$

Let us try to apply a similar technique to a RISC architecture. First we divide the program into basic blocks. A basic block is a group of instructions with a unique entry point such that when the entry instruction executes, all other instructions in the basic block will execute. It is possible to identify all basic blocks in most executable programs by examining branch instruction destinations and indirect jump tables. After determining the basic blocks, we instrument the executable file by preceding each block with code to increment a counter. Running the instrumented program produces a table of basic block counts. A discussion of methods for instrumenting compiled code is provided in the Appendix.

Using the counts and our knowledge of when hardware interlocks are triggered, we can estimate how long each basic block executes. Our estimates will have two shortcomings:

1. Memory access instructions do not execute in constant time.

2. There may be hardware interlocks across basic block boundaries.

The first shortcoming is actually the feature on which our bottleneck detection technique is based. We assume all memory accesses take the minimum possible time (typically the time for a primary cache hit) and when our prediction disagrees with measured execution time we report a bottleneck.

The second weakness is not a problem for many RISC architectures because there are few hardware interlocks and these interlocks rarely cross basic block boundaries. On the MIPS processor where our experiments were performed, the inter-block interlocks were negligible in real code. If such interlocks occur with appreciable frequency, they can be estimated by instrumenting to collect branch frequencies as well as basic block counts. The branch frequencies tell us how often one basic block precedes another and we can improve our estimate by including interlocks between adjacent basic blocks.

Thus, we have a technique for estimating execution time of a whole program. Moreover, our method costs only one instrumented program execution rather than requiring a full, expensive machine simulation. The task now is to use this estimation technique to isolate bottlenecks.

3 Isolating Bottlenecks

In this section, we develop a framework for detecting bottlenecks by measuring divergence from predicted behavior. We begin by formalizing the notion of measuring actual time spent in a region of code. A measurable object or *m-object* is a set of instructions in which we can identify all entry and exit points. The object is measurable because we can place start timer and stop timer calls at these entry and exit points to measure the time spent in the object. For example, we can time a procedure by placing a *start_timer* call at the top of the procedure and *stop_timer* calls before every `return` statement. Similarly, we can time a loop by placing a *start_timer* above the top of the loop and a *stop_timer* below the bottom of the loop.

We say an m-object is *timable* if we can instrument the program to measure the time spent within the object. A timable object must satisfy two criteria:

1. The total time spent within the object substantially exceeds timer granularity.

2. The perturbation created by the timer is not significant.

The first aspect of timability is rarely a problem as most systems provide at least a 1/60th of a second timer, and m-objects of interest typically execute for seconds, minutes, or even hours. The perturbation issue is more difficult. To avoid changing memory performance, we require that the number of memory operations performed by the m-object substantially exceed the number performed in a clock timer call. In addition, to avoid appreciably slowing the program down, we require that the time spent in the m-object substantially exceed the time to make a clock call.

Using the above criteria, we can identify regions of the program whose actual execution times can be measured. These execution times include, however, not only the work done in an m-object proper, but also the work done on behalf of the object by any procedures that it calls. In contrast, the basic block counting estimation technique of the previous section calculates only the work done in a basic block; it ignores the time spent in procedure calls. Furthermore, while we can estimate the total time spent in a procedure q, we cannot necessarily determine the time spent in q on behalf of a particular caller. Thus, we cannot always estimate the time spent in and on behalf of an m-object that calls q. We will say that the m-objects whose execution time can be accurately predicted by basic-block counting techniques are *estimable*.

To identify estimable m-objects, we exploit information about the structure of a program's call graph. In a call graph, nodes represent procedures and there is a directed edge from node v to node w if procedure v calls w during the execution of the program. The call graph has a distinguished node, the root, which is the procedure where execution begins.

We say a node v dominates w if every path through the graph from the root to w passes through v. The useful aspect of the call graph is that a node is estimable if it dominates all of its descendants. Intuitively, if a node v dominates w then all the work in w is done on behalf of v. Hence, if a node dominates all of its children, then the estimated time for that node and its descendants is just the sum of each of their estimated times, ignoring procedure calls.

This observation serves as a working definition of estimability. The definition implies that both the root of the call graph, which corresponds to the execution of the whole program, and the leaves which correspond to call-free procedures, are estimable. Given this operational definition of estimability, the primary issue in isolating memory bottlenecks now becomes one of granularity of detection.

We could estimate the execution time of the full program and compare this number against actual run time, but this will only describe the magnitude of the memory effects, not localize them. Instead, our approach is to find a set of smaller, timable m-objects containing the majority of memory operations, and then to select members of this set that are estimable. The next section outlines our algorithm.

4 The Implementation

This section describes one implementation of the estimable, timable m-object approach to isolating memory bottlenecks, MTOOL. This specific implementation is for Fortran programs running on MIPS-chip based workstations. Fortran was chosen as a target language because large Fortran programs often have memory bottlenecked regions and much of the research on alleviating memory bottlenecks has concentrated on scientific code.

MTOOL seeks to isolate bottlenecks at the level of procedures and loops.

This decision reflects the fact that procedures and loops have natural meaning to the user, satisfy the definition of m-object, and typically run long enough to meet the timability criteria. The first step of the bottleneck isolation process is to instrument the program of interest to collect basic block counts and to run the instrumented code on a representative input. The basic block counts are useful not only for estimating execution time; they also provide MTOOL with precise knowledge of where memory operations are concentrated.

MTOOL sorts the procedures by the number of memory operations they execute and selects those that contain the first 95% of all memory operations. For each of these procedures, MTOOL makes a list of loops and tries to measure first the individual loops and then the whole procedure, subject to timability and estimability constraints. Timability constraints are strongly system dependent. DEC's ULTRIX provides a 1/60th of a second granularity clock, but a system call is required to read the clock.

The overhead of the system call perturbs execution undesirably. The cleanest solution would be to modify the operating system to create a clock directly accessible by user processes, but a simpler, more widely applicable option was to add an interval timer to create a clock in user memory. Start and stop clock calls access the clock in user memory which is updated by the interrupts of the interval timer.

Using the user memory clock, MTOOL's timer has granularity of 1/60th of a second and work per start/stop call of about 70 instructions. The results reported in Section 6 seem to indicate this granularity and overhead are acceptable. Also, we expect that interest in characterizing and improving performance will drive architects and operating systems programmers to provide better clocks in the future.

At this point, MTOOL has gathered a collection of timable loops and procedures. The next step is to determine which of these objects is estimable. Conceptually, MTOOL uses a simple depth first algorithm to label each procedure in the call graph with two parameters:

$$Tot(v) = 1 + \sum_{calls\ from\ v\ to\ w} Tot(w) * \frac{\#\ of\ calls\ from\ v\ to\ w}{total\ \#\ of\ calls\ to\ w} \quad \text{and}$$

$$TDES(w) = 1\ +\ number\ of\ descendants\ of\ w.$$

It is easy to show that for every node, $Tot(v) \leq TDES(v)$ and $Tot(v) =$

$TDES(v)$ exactly when v is estimable. This relation formalizes the observation of the previous section that a node is estimable when all the work of its descendants is done on its behalf.

Thus, we have a test for estimability. Extending the test to loops simply involves checking the analogous condition that:

$$\sum_{w\ called\ in\ loop} TDES(w) + 1 = \sum_{w\ called\ in\ loop} Tot(w) * \frac{\#\ of\ calls\ from\ loop\ to\ w}{total\ \#\ of\ calls\ to\ w}.$$

In most cases, the loops and procedures selected by MTOOL immediately satisfy the estimability condition. The condition is met frequently because the objects selected by MTOOL contain the majority of memory operations which typically means they involve the most frequently executed portions of code which are normally leaves or objects all of whose children are leaves.

When the test is not satisfied immediately, several options are available. One natural choice is to move up the call graph as we know that eventually we will encounter an estimable node, the root. This option is not ideal, however, because it reduces the precision with which we localize bottlenecks. A better choice is to check whether we can in fact accurately estimate the time spent in a procedure call. Below, we describe three cases where we can make an accurate estimate.

Many scientific library routines are simple, loop-free leaf procedures that always follow the same execution path. Their estimated execution times are consequently constant. We can often identify such procedures by checking that they meet two conditions:

1. The procedure is a loop-free and call-free.

2. The average time per call as determined by basic block counts is equal to the maximum or minimum possible time per call.

The first condition implies the procedure is estimable and that we can run shortest and longest path algorithms on the control flow graph of the procedure to bound its execution time. Using these bounds, we can check the second condition which implies that the execution time per call is constant because average equals extremum. This test finds that such common library calls as `SQRT()` and `EXP()` have constant execution times when called in the Perfect Club benchmarks.

1. Instrument executable program and collect basic block counts.
2. Select loops and procedures containing most memory operations.
3. Eliminate selected objects that fail to meet timability constraints.
4. Eliminate selected objects that fail to meet estimability constraints.
5. Instrument code to measure actual time spent in remaining selected objects.
6. Run instrumented code, correlate actual times with estimated times to isolate bottlenecks, and report bottlenecks to the user.

Figure 1: MTOOL's Bottleneck Isolation Algorithm

Two other simple heuristics suffice to eliminate many other problematic calls. Suppose object v calls procedure p. Then, we can normally assume average time per call from v to p is average time per call to v when either:

1. The vast majority (98%) of calls to p are made by v, or
2. (avg. time per call to p) * (# of calls from v to p) $\ll$ time spent in v, so any error in the approximation is negligible.

Both of these heuristics can be inaccurate under pathological conditions (when the variance of the execution time of p across calls is large), so MTOOL issues a warning whenever it invokes them. They have not caused problems with the benchmarks measured in this study.

The steps MTOOL uses to isolate memory bottlenecks are summarized in Figure 1. Step 6 is of course the most significant to the user.

5 User Interface

The user view of MTOOL is considerably less complex than the algorithms of the previous sections. The user types `MTOOL program-name input-files` and waits while MTOOL instruments the program to collect basic block

```
Predicted User Time: 95.0
Measured Time (compensated for counters): user 132.4    sys 0.7
Overhead estimates:         User (memory)         System (I/O)
                            39.3                    0.7
Proceed with memory bottleneck probe insertion (y/n)?
```

Figure 2: MTOOL's Initial Bottleneck Estimate

counts, runs and times the instrumented code, and displays a result like that shown in Figure 2. MTOOL generates the data in the figure by estimating the run-time of the program and comparing it with the measured execution time, appropriately adjusted for the effects of the counters. The information prevents a a user from proceeding when no significant memory bottlenecks are present.

Assuming the user asks MTOOL to produce an instrumented program, MTOOL executes steps 2 to 5 of Figure 1 producing a file of m-object descriptors and actual execution time measurements. This summary file is handed off to the front end.

The front end's top level window is shown in Figure 3 for the Perfect Club benchmark TFS.f, an air flow simulation. Overhead is defined as

$$(actual\,time - estimated\,time)/estimated\,time.$$

The histogram in the lower left of the window visually summarizes the data in the upper right. The line *"Measured 91% of memory operations"* reports the percentage of all executed memory operations that are included in measured objects. All times and overheads refer to user time only; time spent in the operating system on behalf of a program is ignored (though it was reported at an earlier stage–see Figure 2).

By pressing the histogram button, the user can obtain a histogram where bars represent total time spent on behalf of a procedure (Fig. 4). This time is split into estimated time and measured memory overhead. The bars are sorted in decreasing order of memory bottleneck magnitude. By clicking the mouse on the memory overhead portion of a bar, the user opens a text window (Fig. 5) displaying the procedure.

Measured bottlenecks within the procedure are displayed by highlighting the text. Bottlenecks are highlighted one at a time, with the overall overhead contribution and the extra cycles per memory operation associated with the bottleneck reported at the top of the text window. Pressing the INFO button opens a popup window that gives further information about the distribution of memory operations when the measured object includes loops or procedure calls. Figure 6 shows a simple INFO window for the top bottleneck in `dflux()`. Pressing PREV or NEXT displays the previous or next bottleneck; bottlenecks are sorted by magnitude. The text window may be scrolled by pressing the vertical arrows at the right side of the window, and multiple text windows may be open simultaneously.

Three aspects of the user interface deserve comment from an implementor's perspective. First, the interface is relatively portable because it is built using Interviews [4], an object oriented toolkit that runs on top of X. Second, the notion of clicking on a bar to obtain further information on the associated bottleneck is quite general. For example, it would be easy to support an option where clicking on a procedure's estimated time bar would provide information implicit in our time estimate like register usage, floating point stalls, MFLOPS, most often executed lines, etc. A third note about the interface is that is uses standard line number information available in the object file to relate basic block level information back to source code lines. In this sense, the user interface is language-independent.

6 Results

The final measure of MTOOL and its user interface is how well they isolate memory bottlenecks. Table 2 summarizes our results for a subset of the Perfect Club benchmarks and the scientific benchmarks in the SPEC suite. The column entitled "% of Memory Operations" indicates that our heuristics for selecting timable, estimable m-objects succeed in choosing good potential bottlenecks, as we measure over 90% of memory operations in all but two cases where we measure 85%. The column on unexplained overhead, which reports the difference between measured overhead for the whole program and the sum of measured overheads for individual objects confirms this conclusion, as our measured m-objects typically account for all but a few percent of overhead. The least positive result is the data on number of source lines

Program	% of mops	Over-head	Unex-plained Overhd	# of Source Lines In All m-obj.	Median/ m-obj	Max/ m-obj	In All Code
tfs	91%	40%	4%	252	8	53	2020
nas	97%	25%	0%	591	84	395	3976
sds	93%	6%	2%	167	12	120	7607
lws	93%	9%	0%	100	100	100	1237
lgs	98%	12%	0%	323	35	132	2327
fpppp	85%	50%	11%	940	274	666	2718
doduc	85%	29%	5%	2541	102	477	5334
spice2g6	95%	46%	2%	756	41	434	18411
dnasa7	97%	107%	1%	257	9	118	1105

Table 2: MTOOL Effectiveness

contained in the m-objects. On half of the programs, the median object contains fewer than 50 lines, but for some programs like fpppp, we are perhaps failing to localize bottlenecks adequately. This problem is discussed further in the Conclusions section below.

7 Conclusions

The results reported above support the general conclusion that our technique succeeds in detecting memory bottlenecks in scientific Fortran programs. One shortcoming of the current implementation is that it does not satisfactorily localize the bottlenecks in a few of the programs (see Table 2). This problem can certainly be addressed by modifying our algorithm to select smaller m-objects. Currently, MTOOL searches for timable, estimable m-objects starting with outer loops. Some benefit could be derived by trying inner loops first. Similarly, MTOOL will time a whole loop-free procedure, regardless of how many lines it contains. It would be a simple matter to add heuristics to partition large procedures into multiple m-objects. This new heuristic would benefit programs like fpppp in the SPEC set where a key bottleneck is a 700 line loop-free procedure. Finally, we could add a new option where MTOOL displays a procedure with its frequently executed memory references highlighted whenever the MTOOL selected m-object exceeds a certain line count threshold. The user could then manually create acceptably proportioned m-objects containing these highlighted memory operations, subject to MTOOL's checks for estimability and timability.

MTOOL could also be enhanced by broadening the class of programs for which it can detect bottlenecks. MTOOL is now restricted to non-recursive programs that do not use procedure variables. The restriction on recursion derives primarily from the fact that the simple start/stop clock timers will not work when an m-object can be re-entered (and the clock re-started) before it is exited (and the clock is stopped). The restriction could be removed by prohibiting the timing of recursive procedures or by using more complicated timers that keep track of the depth of the recursive call and only start and stop the clock on the first entry and last exit. Because MTOOL is language independent (except for the restriction on recursion) modifying the timers (and modifying the check for estimability of section 4 to account for loops in the call graph) would allow MTOOL to handle languages with recursion.

The restriction on procedure variables is also easy to remove. MTOOL's definition of estimability reqires a well-defined call graph and the use of procedure variables means that the full structure of the call graph is not determined until run-time. By modifying the basic block counting instrumentation to record a dynamic call graph, this restriction could be circumvented.

The final and perhaps most exciting application for MTOOL technology is porting it to a shared memory multiprocessor. Memory bottlenecks on shared memory machines can be severe and detecting them is difficult. Analytic techniques cannot handle the complexity of parallel systems and simulating multiple interacting processors is extremely complex and expensive. MTOOL should provide a viable alternative to these approaches.

Acknowledgements: I would like to thank David Wall of DEC WRL who sponsored part of this research during a summer internship and provided invaluable advice on the ins and outs of patching code.

8 Appendix: Instrumenting Code

MTOOL requires the ability to modify an executable file in a non-intrusive manner. One approach used by the Pixie [5] program from MIPS is to directly patch an executable. A problem arises because some jumps are indirect; they have the form `JR r1` where *r1* is a register containing an address. Pixie's solution is to include an indirect jump table which maps every address in the original executable to its corresponding address in the patched executable. Indirect jumps are always made through this table. The drawback of Pixie is that it can introduce non-negligible overhead.

A second approach is to patch the code at link time, modifying pc-relative jumps, text addresses in the data segment, and the relocation dictionary to correctly reflect changes in the executable file. The advantage of this approach is that all jump destination addresses are clearly identifiable and no overhead is added. See [9] for a a complete description of the technique.

MTOOL uses a third approach. MTOOL patches the executable directly, but it does not use an indirect jump table. Instead, MTOOL pattern matches to find instructions which load addresses in the text segment. The loaded text address is modified to represent the corresponding address in the instrumented code. Thus, a `JR r1` will succeed because the instructions to load *r1* with a value have been correctly updated. This technique suffers from the potential drawback that the pattern matcher could fail in highly optimized code, but we have encountered no problems to date.

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transanctions on Computing*, 30(5):341–56, 1981.

[2] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical Report CSRD 625, University of Illinois, 1987.

[3] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, 1988.

[4] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, 1989.

[5] Mips Computer Systems Inc., Sunnyvale, CA. *Language Programmer's Guide*, 1986.

[6] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[7] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations – exploiting the memory hierarchy. To appear *ACM Transactions on Mathematical Software.*

[8] E. Rothberg and A. Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. To appear *Proceedings of Supercomputing '90*, Nov. 1990.

[9] D. Wall. Link-time code generation. Technical Report 89/17, DEC Western Research Lab, 1989.

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.

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.

‘‘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.