OCTOBER 1989

WRL
Technical Note TN-12

Mostly-Copying Garbage Collection Picks Up Generations and C++

Joel F. Bartlett

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.

Mostly-Copying Garbage Collection Picks Up Generations and C++

Joel F. Bartlett

October, 1989

Abstract

The "mostly-copying" garbage collection algorithm provides a way to perform compacting garbage collection in spite of the presence of ambiguous pointers in the root set. As originally defined, each collection required almost all accessible objects to be moved. While adequate for many applications, programs that retained a large amount of storage spent a significant amount of time garbage collecting. To improve performance of these applications, a generational version of the algorithm has been designed. This note reports on this extension of the algorithm, and its application in collectors for Scheme and C++.

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

1. Introduction

The "mostly-copying" garbage collection algorithm [1] [2] was developed to allow compacting garbage collection in environments hostile to classical copying collection algorithms. Unlike classical copying collectors, it is able to tolerate ambiguous pointers in its root set. This has allowed it to be used to manage storage in a portable Scheme system, Scheme->C, that compiles Scheme to C [3]. In such an environment, the runtime system has no control over or knowledge of how the stack and registers are used. As originally defined, the mostly-copying collector's behavior is similar to that of the classical "stop-and-copy" collector. Both have a running time that is O(accessible storage) and reclaim memory by moving accessible storage to another "space" in memory.

While such a collector has adequate performance for many applications, it does have some limitations. In large programs, each time a garbage collection occurs there is a significant pause in computation as almost all accessible storage must be copied, and the time spent garbage collecting can be a significant portion of the total program execution time. For example, Steele [10] reports that large Lisp programs spend 10 to 40 percent of their time garbage collecting, and experience with Scheme->C suggests similar behavior. Before trying to improve this situation, one should remember that different programs require different garbage collector behavior. For example, an interactive graphics editor wants to minimize the program pauses during garbage collection, while a Scheme compiler wants to minimize the total time spent garbage collecting.

Generational garbage collection techniques appear to be able to satisfy the needs of both types of programs; the remainder of this note describes a "generational mostly-copying" garbage collector. Section 2 introduces generational collection and is followed by a review of the mostly-copying collection algorithm in section 3. These ideas are merged in section 4, which shows how the mostly-copying algorithm can be made generational. Section 5 describes the implementation in the Scheme->C system, and section 6 describes the implementation for C++. The note ends with the conclusion in section 7 that a "generational mostly-copying" collector is able to provide performance improvements for both interactive and non-interactive programs.

2. An Introduction to Generational Garbage Collection

One way to improve the efficiency of copying garbage collectors is to reduce the amount of storage copied on each garbage collection. Generational garbage collectors attempt to do this by treating objects differently depending upon their age. In his overview of garbage collection, McEntee [7] credits Lieberman and Hewitt with a proposal for generational garbage collection in 1980, and Moon with a later implementation of a generational garbage collector for the Symbolics 3600. Work more relevant to the current effort is that done on conventional processors by Ungar [12] for Smalltalk[1] and Shaw [9] for Common Lisp.

Generational garbage collectors rely on two observations about dynamic storage allocation: new objects are more likely to be freed than old objects, and new objects typically point to old

[1]Ungar also originated the generational garbage collection slogan: "young objects die young."

objects. These garbage collectors ignore long-lived objects and concentrate on reclaiming space occupied by short-lived objects. The algorithm can be illustrated by considering garbage collection with two generations (additional generations can be added in a straightforward manner).

When an object is initially allocated, it is considered to be a member of the first generation. Once an object has survived some number of collections, it is moved to the second generation (also known as the stable set). When storage is reclaimed, one ignores stable set objects and attempts to recover it from the inaccessible first generation objects. Only when one is unable to get enough space from first generation objects will one garbage collect objects belonging to the stable set.

Generational garbage collectors need a larger root set than either the mostly-copying or classical stop-and-copy collectors. Besides the usual register and stack contents, they must also know about all stable set objects that reference first generation objects. If this information (called the remembered set) were not known, first generation objects whose only references were from stable set objects would be deleted in error. The remembered set may be maintained by either software tests at appropriate stores in the program or by virtual memory hardware traps. With this introduction to generational collection complete, we will now turn our attention to a review of the mostly-copying collection algorithm.

3. A Review of Mostly-Copying Collection

The mostly-copying collector is an evolution of the classical stop-and-copy collector [5]. Both algorithms divide the storage area into two equal semispaces: "old space" and "new space". When all the storage is allocated in one semispace, the garbage collector is invoked. It recovers space by using a set of root pointers (also known as the base set) to move all accessible storage to the other semispace. Where the mostly-copying algorithm differs from stop-and-copy is in how it defines its root pointer set and semispaces.

The mostly-copying algorithm makes few restrictions on the root set. It simply requires that somewhere in the set of cells, *R*, there be sufficient "hints" to find all accessible storage. A typical *R* would be the current program state, i.e. the entire contents of the processor's stack and registers.

The heap used by the mostly-copying algorithm is a contiguous region of storage, divided into a number of equal-size pages[2]. Associated with each page is a space identifier, *space*, which identifies the "space" that objects on the page belong to. In the figures illustrating this algorithm, the space identifier associated with the page containing the cell is the number to the left of the cell, and each cell is assumed to be in its own page (see Figure 1).

Two space identifiers: *current_space* and *next_space*, identify important sets of pages. During normal allocation, *current_space* and *next_space* are equal, but during garbage collection, *next_space* is set to the "next" space identifier. When comparing this algorithm with the classical one, it is reasonable to think of pages with their *space* identifier equal to *next_space* as the

[2]The page size used by the algorithm is independent of any underlying hardware's page size.

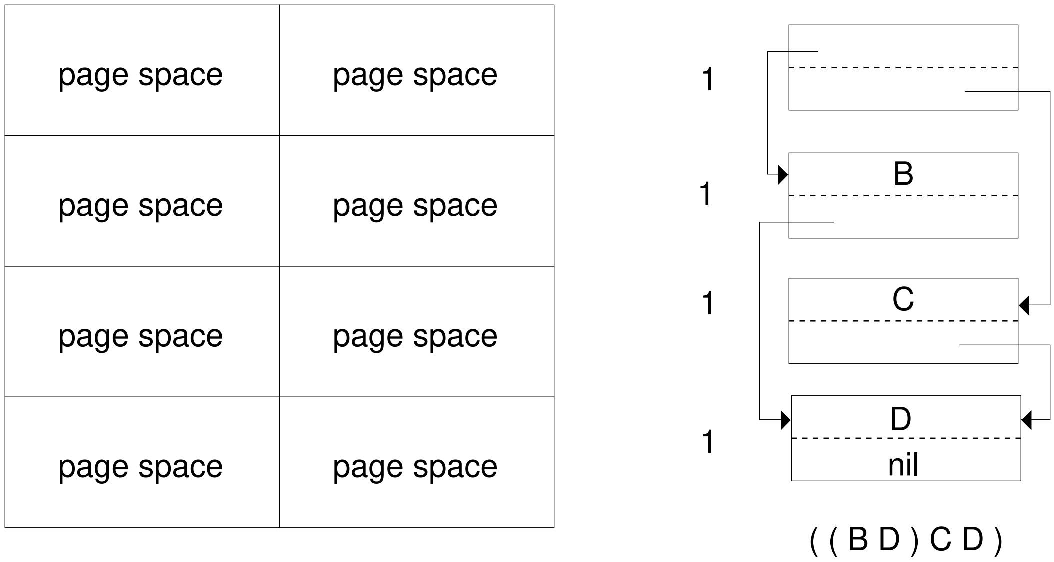

Figure 1: Mostly-copying memory organization and sample list

analogue of "new space", and those with their *space* identifier equal to *current_space* as the analogue of "old space". Like the classical one, this collector works by moving objects from "old space" to "new space". While this can be done by copying objects to newly allocated pages in *next_space*, it can also be done by changing the *space* identifier associated with the page holding the object to *next_space*. This later method is the key to mostly-copying collection as it leaves the object's address unchanged.

Memory allocation is a two part process: first, allocate a page of memory, and then allocate space from it. A page is free when its *space* field is not equal to *current_space* or *next_space*. When it is allocated, its *space* identifier is set to *next_space*.

The garbage collector is invoked when half the pages in the heap have been allocated. It starts by advancing *next_space* to the next space identifier. Next, it makes an educated guess as to what portions of the heap contain accessible items. This is done by examining each word in the stack and the registers and looking for "hints". If the word could be a pointer into a page of the heap allocated to the *current_space*, then that page is promoted to *next_space* by changing the page's *space* identifier (see Figure 2).

At the completion of this phase, all pages containing items which might be referenced by pointers in the stack or registers are in *next_space*. The items which they reference are now copied to *next_space* (see Figure 3). This is done by sweeping across all pages in the *next_space* and updating their pointers by a method similar to that used by the stop-and-copy collector. Like the stop-and-copy algorithm, the objects in the heap must be self-identifying at collection time, and their pointer fields must be valid. Once all pointers in pages in *next_space* have been updated, *current_space* is set to *next_space* and garbage collection is complete.

For more information on this algorithm and its variants, the reader is directed to the technical report [1]. With this review of mostly-copying garbage collection complete, it is now appropriate to turn our attention to making the algorithm generational.

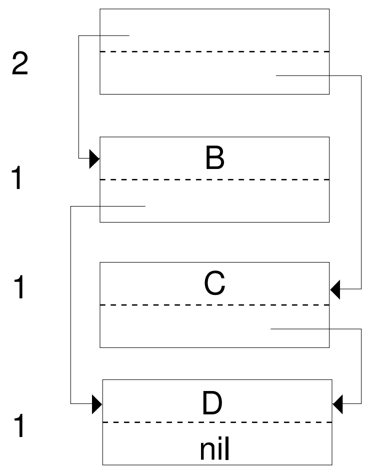

Figure 2: Promote possibly referenced pages to *next_space*

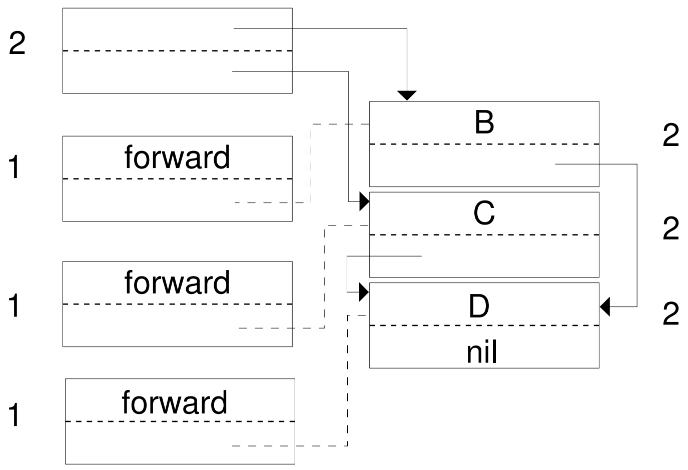

Figure 3: Copy the rest of the accessible objects

4. A Generational Mostly-Copying Collector

The most straightforward (and least efficient) way to keep track of the age of an object is to place a garbage collection counter in each object. Each time the object is garbage collected, the counter is incremented. When the counter gets above a certain value, the object is moved into the stable set. Since the mostly-copying collector divides memory into pages, an object's age may be approximated by associating it with the page holding the object. One way to store this is to encode it in the space identifier associated with each page.

A simple two-generation "generational mostly-copying" garbage collector can be constructed from the previous algorithm as follows[3]. During user program execution, objects are allocated

[3]Shaw [9] extended stop-and-copy collection in a similar manner and called it stop-and-copy with stable data.

with odd-numbered space identifiers. Objects that have survived one garbage collection are moved into the stable set that is stored in pages marked with an even-numbered space identifier.

During its execution, the user program assists the garbage collector by constructing the remembered set. This is done by keeping track of all pages with even-numbered space identifiers that have pointers stored into them.

Memory allocation is a two part process: first, allocate a page of memory, and then allocate space from it. A page is free when its *space* field is odd and not equal to *current_space*. When it is allocated, its *space* identifier is set to *next_space*.

The garbage collector is invoked when half the pages in the heap have been allocated. It starts by advancing *next_space* to the next space identifier (an even number). Next, it makes an educated guess as to what portions of the heap contain accessible items. This is done by examining each word in the stack and the registers and looking for "hints". If the word could be a pointer into a page of the heap allocated to the *current_space*, then that page is promoted to *next_space* by changing the page's *space* identifier.

At the completion of this phase, all pages containing items that might be referenced by pointers in the stack or registers are in *next_space*. The items which they or members of the remembered set point to are now verified to be in the stable set[4] or copied to *next_space*. This is done by sweeping across all pages in the *next_space* and the remembered set and updating their pointers by a method similar to that used by the stop-and-copy collector. Once all pointers in pages in *next_space* and the remembered set have been updated, *current_space* and *next_space* are set to *next_space*+1 (an odd number) and garbage collection is complete.

At some point in time, the garbage collector may wish to recover inaccessible objects in the stable set. This can be done by changing the space identifier of all pages in the stable set to *current_space* and then garbage collecting using the original mostly-copying algorithm.

Multiple generations can be accomodated by this algorithm with appropriate encodings of the space identifier. For example, space identifier values equal to 0 modulo 4 could represent the first generation, values equal to 1 modulo 4 could represent the second generation, etc. Appropriate remembered sets would be retained for references from older to newer generations. As objects continue to be retained, they would gradually move from one generation to the next, where higher numbered generations contain older objects that are less often garbage collected.

5. Implementation for Scheme->C

The generational mostly-copying collector added to Scheme->C is similar to the one previously described. It has two generations and the remembered set is maintained by software. When an object survives one garbage collection, it is moved to the stable set. When the amount allocated in the heap following a garbage collection of the first generation is greater than a user defined level, both generations are garbage collected.

[4]Pointers to objects in pages with even space identifier values need not be followed as those objects are in the stable set.

When compiling the Lisp dialect Scheme [8], the compiler need only generate code to maintain the remembered set for constructs that destructively change lists: SET-CAR! and SET-CDR!, store into a vectors: VECTOR-SET!, and make assignments: SET!.

By comparing the performance of programs running with the old and new garbage collectors one can observe the costs and benefits of generational mostly-copying collection. The test programs are four benchmarks from Gabriel's benchmark suite [6] and the Scheme->C compiler compiling a large Scheme program. The benchmarks chosen perform a significant number of modifications to existing data structures. As they are all fairly short, no garbage collection is done during their execution. The increase in the time spent in the benchmark programs gives one an estimate of the cost of managing the remembered set via software. As one might expect, the worst increase was in the program Destructive which constructs a tree as a list of lists and then destructively modifies its elements. Programs that make extensive use of vectors, such as FFT, Puzzle, and Triangle, showed smaller increases in running time. The performance decrease observed here is in agreement with measurements reported in [9].

For larger programs that require garbage collection, the time spent managing the remembered set is more than made up for by the reduced garbage collection time. In the sample Scheme->C compile, the program time was increased by 8%, but the total running time was reduced by 20%.

Program name	Program time change	GC time change	Total time change
Destructive	+12%	-	+12%
FFT	+4%	-	+4%
Puzzle	+4%	-	+4%
Triangle	+9%	-	+9%
Scheme->C compiler	+8%	-82%	-20%

Similar improvements have been observed for interactive programs. A graphical multiprocessor performance monitor written by Bert Halstead that used to take approximately 500 ms to garbage collect is now able to collect in less than 30 ms. Similarly, garbage collection pauses for a visual programming language system written by John Danskin are now 50-400ms rather than 2-5 seconds.

Those familiar with earlier work in generational garbage collection may wonder if this collector is really too simple. That is, by using only two generations of storage, and moving objects into the stable set when they survive one collection, do we retain too many items[5]? Experience to date with Scheme->C suggests that this is not a problem, and a comparison with Ungar's work [12] shows why. The typical Scheme->C workstation is a DECstation3100 that has approximately one order of magnitude greater performance than the SUN's used by Ungar. First, this allows Scheme->C programs to allocate 2MB or more of heap between collections and still have good interactive performance. In contrast, Ungar's volatile objects occupied only 200KB. Second, in Scheme->C, reclamation of the stable set takes a few seconds and thus can be

[5]Ungar calls this the "the tenuring problem".

tolerated after some number of minutes of computation. For example, a 12MB heap with 3MB accessible data can be totally reclaimed in less than 7 seconds. Ungar on the other hand suggests that collecting the stable set will take 5 minutes and must be done every 3 to 8 hours.

Having shown that the generational extension to the mostly-copying collection algorithm is worthwhile for Scheme->C, our attention now turns towards implementations for other programming languages, such as C++.

6. Implementation for C++

C++ is an object oriented superset (except for a couple of minor details) of the C programming language [11]. The garbage collector for it evolved from that published in the appendix of [1][6]. In order to simplify its use, the heap size need not be configured (it expands automatically as needed) and all static data is assumed to contain hints. Rather than requiring pointers to be located at the beginning of each object, the user provides the garbage collector with a method to locate the pointers. For example, a class that holds a variable length string, a reference count, and pointers to strings that are greater or lesser than it can be defined as follows:

```
struct  word  {
        word* lesser;
        word* greater;
        int  count;
        char  symbol[ 4 ];
        word( char* chars );
        void  pointers();
};

void  word::pointers( )  {
        gcpointer( &lesser );
        gcpointer( &greater );
}

GCCALLBACK( word_callback, word::pointers );

word::word( char* chars )
{
        this = (word*)gcalloc( sizeof( word )+
                               strlen( chars )-3,
                               word_callback );
        lesser = NULL;
        greater = NULL;
        count = 1;
        strcpy( symbol, chars );
}
```

The constructor, `word::word`, allocates space in the heap and then initializes it. Space allocation is done by the call to `gcalloc` that takes the size in bytes and a "callback object" as its arguments. The callback object, `word_callback`, is declared by `GCCALLBACK`. It denotes

[6]Astute readers of that document will note that processing of objects larger than one page is incorrect in the procedure `promote_page`, as it does not correctly identify the first page of the object.

the method, `word::pointers`, that the garbage collector calls to identify all pointers in the object. This is done by calling `gcpointer` with the address of each pointer in the object that could point to a garbage collected object.

Once the object has been defined, storage is allocated using the normal C++ mechanism:

```
sp = new word( "dictionary" );
```

Maintaining the remembered set is more of a problem in C++ than in Scheme. Since Scheme->C was implemented at WRL, it was easy to modify the compiler to add software checks in the small number of areas where data in the stable set could be modified. In C++ though, any store via a pointer has the possibiliy of modifying stable set data. Monitoring all such stores by software would require modification of a foreign compiler (not desired) and would have a significant impact on program performance. An alternative way to manage the remembered set is to use the underlying processor's virtual memory system, as was done by Shaw [9]. This was not done here as it was desired to keep the garbage collector as architecture independent as possible. Instead, the entire stable set is considered to be the remembered set. While this is not as good as knowing the actual remembered set, it is better than not using a generational garbage collector. Given that one expects much of the stable set to be retained, it is cheaper to scan the entire stable set for pointers to first generation objects, than to both move and scan most of the stable set. Shaw's work with Common Lisp supports this view.

One extension that an implementer might wish to make is to allow C++ destructor methods to explicitly return storage. Given that one trusts the programmer to know when to return data, there may be some benefit in this. A second extension that one may wish to make is to have the garbage collector call a destructor method when it detects that an object is no longer needed. In order to do this without having to scan all recovered storage (and thus losing one of the advantages of a copying collector), one could register those objects needing cleanup action with the garbage collector. One should recognize though that rapid activation of cleanup procedures is not something that generational collection does well. It gains performance by ignoring long lived objects; the longer an object lives, the longer it will take to recognize that it is no longer needed. One way to get around this would be to keep all objects needing cleanup actions in the first generation. However, if this represents a significant portion of the accessible data, the benefits of generational collection will be lost.

An alternative type of garbage collector that one might try with C++ is a conservative collector [4]. Unlike the mostly-copying collector, such collectors do not put any restrictions on pointer use nor do they compact memory. With such collectors, objects in the heap need not have valid pointers nor have runtime discernible structure. While they might place fewer restrictions on programs, the benefits of memory compaction provided by the mostly-copying collector are worth the restrictions.

7. Conclusions

This note has discussed an improvement to the mostly-copying garbage collection algorithm that allows it to provide generational collection for a variety of programming languages. While performance data is not available for C++, its performance in the Scheme->C system shows that it provides significant performance improvement over the previous non-generational version of the collector. While small benchmarks that don't require any garbage collection run a bit slower, large batch programs such as compiles run faster and interactive programs have shorter pauses.

8. Acknowledgements

John Danskin's desire to construct a large interactive Scheme program encouraged the search for a generational mostly-copying collector. Bert Halstead provided a benchmark program. Joel McCormack and Mike Newman spoke strongly in favor of explicit storage deletion in C++. Mary Jo Doherty, David Wall, and Jeremy Dion offered insightful comments that improved the presentation of the material. I thank you all.

9. References

[1] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots*. Technical Report WRL Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, February, 1988.

[2] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. *LISP Pointers* 1(6):3-12, April-May-June, 1988.

[3] Joel F. Bartlett. *Scheme->C a Portable Scheme-to-C Compiler*. Technical Report WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, January, 1989.

[4] Hans-Juergen Boehm, Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience* 18(9):807-820, September, 1988.

[5] Jacques Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys* 13(3):341-367, September, 1981.

[6] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* The MIT Press, 1985.

[7] Timothy J. McEntee. Overview of Garbage Collection in Symbolic Computing. *LISP Pointers* 1(3):8-16, August-September, 1987.

[8] Jonathan Rees, William Clinger (Editors). Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* 21(12):37-79, December, 1986.

[9] Robert A. Shaw. *Empirical Analysis of A Lisp System*. Technical Report CSL-TR-88-351, Stanford University, February, 1988.

[10] Guy L. Steele Jr. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM* 18(9), September, 1975.

[11] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[12] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157-167. April, 1984.

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.

‘‘Leaf: A Netlist to Layout Converter for ECL Gates.’’
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, 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.

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 Bartlett.
WRL Technical Note TN-12, October 1989.

MOSTLY-COPYING GARBAGE COLLECTION PICKS UP

Table of Contents

List of Figures