CMU-CS-84-122

Error Detection with Memory Tags

Richard H. Gumpertz

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

December, 1981

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-84-122

Error Detection with Memory Tags

Richard H. Gumpertz

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

December, 1981

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the United States Government, or Carnegie-Mellon University.

Abstract

The ability to achieve and maintain system reliability is an important problem that has become more critical as the use of computers has become more common. Fundamental to improved reliability is the ability to detect errors promptly, before their effects can be propagated.

This dissertation proposes methods for using storage tags to detect a broad class of hardware and software errors that might otherwise go undetected. Moreover, the suggested schemes require minimal extensions to the hardware of typical computers. In fact, it is shown that in many situations tags can be added to words of storage without using any extra bits at all.

Although tagging is central to the discussion, the methods used differ radically from those used in traditional tagged architectures. Most notably, no attempt is made to use the tags to control what computations are performed. Instead, the tags are used only to check the consistency of those operations that would be performed anyway in the absence of tagging. By so doing, redundancy already present in typical programs can be harnessed for detecting errors. Furthermore, it becomes possible to check an arbitrary number of assertions using only a small tag of fixed size.

The dissertation examines various strategies for exploiting the proposed tagging mechanisms; both the positive and negative aspects of each application are considered. Finally, an example is described, showing how tagging might be implemented in a real machine.

Acknowledgments

Acknowledging those who have contributed to any major piece of research is always awkward. One does not want to omit anyone, but on the other hand one does not want to dilute the expression of gratitude by listing everyone who was even peripherally involved. Although my advisor, William Wulf, was my primary technical sparring partner, I also thank Joseph Newcomer, the rest of my committee, and the many other members of the CMU Computer Science community who helped me resolve vague notions into concrete proposals. I know of no other environment that would have been as cooperative and helpful.

To my family and friends who saw me through, I am deeply indebted. Most important of all, however, was the support of my wife Linda who, once again, endured.

Table of Contents

Chapter 1: Introduction 1
1-1. Overview 5
1-2. Coding 5
1-2.1. Some terminology 5
1-2.2. Unpatterned errors 8
1-2.3. External redundancy 9
1-2.4. Error correction 10
1-3. Encryption 12
Chapter 2: Mechanisms 17
2-1. Traditional coding and the addition of tags (C-tagging) 18
2-1.1. The parity code 18
2-1.2. Adding a tag to the parity code 19
2-1.3. The Hamming code 20
2-1.4. The extended Hamming code 22
2-1.5. Adding a tag to the extended Hamming code 23
2-1.6. Hsiao's modified Hamming code 25
2-1.7. Codes with greater minimum distance 26
2-1.8. Previous work 27
2-2. Tagging using encryption (E-tagging) 28
2-2.1. Previous work 28
2-2.2. Exaggeration of undetected errors 29
2-3. Combining explicit tagging with encryption 30
2-4. Hashing of tags 31
2-4.1. Incremental hashing 33
Chapter 3: Applications 35
3-1. Addressing checking 35
3-1.1. Choice of addresses for tags 36
3-1.1.1. Physical address tagging 37
3-1.1.2. Virtual address tagging 37
3-1.1.3. Object-name tagging 38
3-1.2. Instance tagging 39
3-1.3. Multiplexer checking 41
3-2. Type tagging 42
3-2.1. Run-time type checking 42
3-2.2. User-defined types 43
3-2.3. Implementation of type-tagging 46
3-2.3.1. Type-tagging at subroutine entry and exit 46
3-2.3.2. In-line type-tagging 47
3-2.3.3. Other models of type unsealing 48
3-2.3.4. An example 50

3-2.4. Type comparison without type inquiry 52
3-3. Ownership tagging 52
3-4. Other applications 54
3-4.1. Bounds checking 54
3-4.2. Random jump detection 55
3-4.3. Uninitialized variables 55
3-4.4. Variant records 56
3-4.5. Extended tagging 56
3-5. Summary of the applications 57
Chapter 4: Implementation details 59
4-1. Tag checking when storing 59
4-2. Allocated but uninitialized *vs.* initialized storage 61
4-3. Tagging of variant records 62
4-4. Slicing 63
4-5. Packed data structures 64
4-6. Multi-cell values 65
4-7. Debugging tools 65
4-8. I/O and other "blind" accesses 66
4-9. Garbage collection 67
4-10. Stacks 68
4-11. Resonance 70
Chapter 5: An example 73
5-1. A summary of Nebula 74
5-2. Tags used 77
5-3. Tagged-cell size 78
5-4. Tag size 79
5-5. Hashing 79
5-6. Registers 80
5-7. In-line literals 81
5-8. Stack allocation 82
5-9. Operand specifier changes 83
5-10. Instruction changes 84
5-11. Summary of the changes 85
5-12. Omissions 86
Chapter 6: Conclusion 87
6-1. Summary 88
6-1.1. Hashed tags 88
6-1.2. Implementation 89
6-1.3. Applications 89
6-2. Evaluation 91
6-3. Future research 93
6-4. Parting words 94
Appendix A: An observation concerning DES 95
Appendix B: A fast parity encoder 97
Glossary 99
References 103

Chapter 1
Introduction

> When you are reciting poetry, which is a thing we never do, you find sometimes, just as you are beginning, that Uncle John is still telling Aunt Rose that if he can't find his spectacles he won't be able to hear properly, and does she know where they are; and by the time everybody has stopped looking for them, you are at the last verse, and in another minute they will be saying, "Thank-you, thank-you," without really knowing what it was all about. So, next time, you are more careful; and, just before you begin you say, *"Er-h'r'm!"* very loudly, which means, "Now then, here we are"; and everybody stops talking and looks at you: which is what you want. So then you get in the way of saying it whenever you are asked to recite . . , and sometimes it is just as well, and sometimes it isn't. . . . And by and by you find yourself saying it without thinking. Well, this bit which I am writing now, called Introduction, is really the *er-h'r'm* of the book, and I have put it in, partly so as not to take you by surprise, and partly because I can't do without it now. There are some very clever writers who say that it is quite easy not to have an *er-h'r'm*, but I don't agree with them. I think its is much easier not to have all the rest of the book.
>
> A. A. Milne [58 (pp. ix–x)]

The ability to achieve and maintain system reliability is an important problem that has become more critical as the use of computers has become more common. Fundamental to improved reliability is the ability to detect errors promptly, before their effects can be propagated. This dissertation proposes methods for using storage tags to detect a broad class of hardware and software errors that might otherwise go undetected. Moreover, the suggested schemes require minimal extensions to the hardware of typical computers. In fact, it is shown that in many situations tags can be used without adding any extra bits to words of storage.

Even when the hardware of a computing system performs properly, the software might not. Standard hardware approaches to reliability, such as coding, replication, etc. provide no help in such situations. Nevertheless, there is quite a bit of redundancy in the typical program that might be used to detect software errors. This takes the form of simple semantic content, such as the fact that an *integer-add* operation would normally reference integers. The proposed technique allows one to

harness at least some of this semantic content that would otherwise be ignored. In particular, it allows the hardware to verify certain kinds of simple assertions, many of which are specific to the program being executed. While this in no way guarantees program correctness, it nonetheless does detect certain classes of software (and hardware) failures that currently might go undetected.

Of course, it would be impractical to add arbitrary amounts of information to each item in storage. To do so would consume excessive storage and execution time. Therefore, the proposed mechanism uses a fixed-size tag. By appropriate specification of the size of the tag, a system designer can choose any desired level of confidence in the thoroughness of the comparison. As the tag-size is increased, the error-detection capability is also increased.

In the early years of computing, the devices used to implement machines were individually so unreliable that redundant elements were often necessary to provide reasonable performance. Completely duplicated hardware, self-checking circuits, and error detecting codes were common. With the advent of more reliable devices, primarily the transistor, much of the concern with component failure disappeared from commercial designs. Fault detection was included in only a few applications: telephony, weaponry, space exploration, etc. In most other equipment one often found no more than parity-checks of primary (core) storage and more complex checks of secondary (magnetic) storage. The central processor of a computing system was considered "reliable enough" without any extra circuitry. No special provisions were made for detecting errors. In many cases, those failures that did occur were discovered only through inability to get programs to run "properly;" subtle failures could go undetected for long periods.

As usage of computers increased, higher demands were made for both speed and correctness. Unfortunately, these two properties tend to oppose each other; often the one can be improved only at the expense of the other. Meanwhile, the cost of error detection circuitry dropped drastically relative to overall system cost. Advances such as integrated circuit memories both necessitated and allowed redundancy. Furthermore, the relative cost of labor-intensive maintenance and debugging increased. At least one manufacturer is now reported to devote about 30% of the hardware in a "typical" processor to error detection/correction/diagnosis circuitry. Nevertheless, more error control is needed. This is especially true at higher abstraction levels which until now have not received much attention.

Dynamic ("run-time") checking of software has not been as common as dynamic checking of hardware. This is largely due to the common assumption that software cannot "break;" if it works today then it should also work tomorrow. Those defects that are present when a program is first

written ought to be detected during the debugging and testing process. Although this premise may be valid for very simple software, experience has shown that it is often incorrect for even moderately large programs. The following are all examples of problems that might not appear until well after initial debugging:

- defects that affect only a few instances of possible input data (*e.g.*, the bug in evaluating $e^{ln\ 2.02}$ on the original HP-35 scientific calculator [26]);
- defects that affect behavior only under peculiar ordering or timing of requests;
- defects that show up only under usage patterns not previously anticipated or tested;
- defects that were previously masked by the compensating behavior of other software or hardware.

All of the above examples are cases in which software can at least *seem* to break. Since no reprogramming of the faulty program occurs in any of them, one could not in general assume that defective module will be specifically tested. Rather, only those checks made during *normal* execution would be available to detect the failure. Thus, to control the effects of software "breaking," one must provide checks that execute during normal "production" runs.

Many programmers have learned, often through bitter experience, to employ dynamic checking. Almost every standard of programming style exhorts one to check explicitly that a program's parameters meet any assumptions made by that program. A few may even suggest checking of values generated within a module. Some programming languages can help by automatically inserting well known tests, such as for subscript-range errors. Unfortunately such checks are often omitted to speed execution. Furthermore, the implementation of some checks, such as for detecting *dangling references* or improper *flow of control*, are by no means straightforward. Their implementation can slow down not only execution of the program but also the programmer himself (by distracting him from the principal problem). Were these checks implemented in hardware, they might be used more often. As pointed out by Hoare [27],

> . . . it is absurd to make elaborate [error] checks on debugging runs, when no trust is put in the results, and then to remove them in production runs, when an erroneous result could be expensive or disastrous.

Myers [62] cites several instances of such fiascos.

The error-detection mechanisms proposed in this dissertation are designed to help detect a number of hardware and software failures that would otherwise go undetected. These include:

- **Failures in addressing mechanisms:** Far more attention has been paid in the past to the failure of data storage elements than to failure of the mechanisms used to address those elements.

- **Incorrect demultiplexing:** Various other forms of selecting the wrong data, such as improperly timed "strobing" of a time-multiplexed bus, are not often detected by current hardware.

- **Use of illegal pointer values:** Software too can cause improper accesses. Typical cases are the use of uninitialized pointer variables or obsolete (dangling) pointer values.

- **Type and modularity violations:** It has been observed that a wide range of errors can be detected by limiting a program's ability to access storage. Visibility restrictions based on *modules* and *types* are already common in modern programming languages. Just as the military reinforces security by granting access to information on a "need-to-know" basis, so can a programming system detect improper code by enforcing the boundaries defined by module and type abstractions.

Note that all of the above can be considered instances of accessing the wrong value, accessing it in the wrong manner, or accessing it at the wrong time. In each case the fetched data are likely to be internally consistent. That is, a parity or Hamming code would not signal an error. Despite the appearance of validity, the value read is just not the one really wanted.

Traditionally, not as much attention has been paid to detecting the errors listed above as has been given to detecting other errors, such as memory failures. Nevertheless, the effects of one of these errors occurs, when it occurs, can be at least as catastrophic as those that would result from those errors for which checking is currently performed. Furthermore, experience seems to indicate that the higher-level errors occur fairly frequently. Therefore, if it can be done at reasonable cost, adding checks for these errors seems well worthwhile.

Normally, adding error checks requires adding extra hardware. If nothing else, there must be bits that can be checked for consistency with each other. In this case, however, much of the redundancy necessary for appropriate error checks is already present in today's computers. The instruction stream, for instance, contains information about the data it accesses. From the program counter one can often derive information as to which module is executing and hence the data that it might legally reference. Furthermore, addressing arithmetic is often quite limited—one almost never uses a pointer to an object to compute the address of another object that is not strongly related to the first. In all of these cases, comparison of the implicit knowledge so derived with that stored in tags can be used for verification of assertions.

Although the checks could be implemented in software, it will be shown that it is possible to provide them in hardware at minimal added cost. By exploiting existing redundancy and adding small amounts of new information, a large gain in error detection is achieved. Because the checks are inexpensive, they can be performed with great frequency and so it is likely that detection will occur

soon after the error condition manifests itself. *Prompt* detection, even without details, is the most important factor in achieving reliable computation. Errors must not be allowed to spread far from the original failure. Otherwise, not only can determination of the source of the error become difficult but also location and correction of the damage becomes much harder.

1-1. Overview

The body of this dissertation consists of three principal parts: mechanisms for tagging words in storage, potential applications in which such tags might prove useful for detecting errors, and an example to make the ideas more concrete. There is little direct precedent for the tagging mechanism proposed and so a traditional discussion of previous work would be strained. Of course this does not mean that I do not draw on existing knowledge. In particular, two areas are critical to the background of this dissertation: coding and encryption. Because many in the intended audience may not be well versed in these subjects, brief introductions to each are included below. Other references to previous work are integrated with the descriptions of the relevant mechanisms.

1-2. Coding

Rather than give a detailed survey of error-detecting and error-correcting codes, I refer the reader to the survey article by Tang and Chien [85]. A good introductory textbook was recently published by Hamming [25]; for more detail there are two outstanding references works, one by Peterson and Weldon [70] and the other by MacWilliams and Sloane [54]. The latter book includes a bibliography which must be seen to be appreciated. The reader need not be familiar with the detailed operation of any particular codes. The two codes that are used in this dissertation, parity and Hamming, are explained as needed in Section 2-1.

1-2.1. Some terminology

Before going on, it is important to define some terminology. In addition, a glossary at the end of the dissertation contains definitions of a few terms that may be unfamiliar to particular readers.

It is assumed that Boolean algebra needs no explanation. Because symbology sometimes varies, however, it should be noted that the symbol "$\wedge$" is used to denote the *intersection (and)* operator and the symbol "$\oplus$" the *sum/difference (exclusive-or)* operator.

All error detection schemes depend on redundancy—without some distinction between legal and illegal conditions there would be no way to realize that an error has occurred. Checking for an error

involves examining a set of signals, to see whether the value it denotes is valid or invalid. If invalid, an error is detected; if valid either no error has occurred or one has gone undetected. The redundancy provides the information necessary to distinguish valid values from invalid ones. Without it, all values would be valid.

The term *codeblock*[1] is used in this dissertation to denote the set of signals examined by such an error check. In memory systems, the codeblock is usually obvious—the bits fetched from memory. In other systems the codeblock may be less obvious—an example might be the combination of an array descriptor and a subscript. Those codeblock values that are designated as valid are usually referred to as *codewords*; there is no standard name for the remaining (invalid) values. The selection of a codeblock and which of its values are codewords are known collectively as a *code*. The individual signals in the codeblock are often referred to as *characters*. Because I will normally restrict my discussion to binary codes, however, the term *bit* will often be used instead.

In the case that a codeblock's value is changed by an error, MacWilliams and Sloane [54] and Wakerly [92] use the term *error vector* to denote the Boolean difference between the intended codeword and the codeblock value actually received. I use the term *symptom*[2] to express a similar but more general (and less formal) concept. Even when one cannot be as mathematically specific about an error as the term *error vector* would require, one can still use *symptom* for an intuitive connotation. It can be considered to denote those things which are visibly changed because of an error.

There is a convention often used for describing codes. The notation *(n,k)* is used to describe codes that have n bits in each codeword, k of which are data bits. That is, the codeblock can take on 2^n values of which 2^k are codewords. The remaining $2^n - 2^k$ codeblock values are invalid (and so can be encountered only as the result of an error).

Occasionally, this *(n,k)* notation is extended to *(n,k,d)*, in which case d denotes the minimum *distance* between codewords. For binary codes the metric most often used is *Hamming distance* which is the weight of (*i.e.*, the count of ones in) the Boolean difference of the words being measured. The minimum distance of a code is thus the minimum number of bits that must be altered to distort

---

[1] In traditional coding theory, the term *message* is sometimes used to denote a similar concept. For the purpose of this discussion, however, the implication of a communication system would be misleading. Hence, using an alternate term seems appropriate.

[2] The term *symptom* should not be confused with another term often used when discussing error-correcting codes: *syndrome*. The latter is the matrix-product of a linear code's parity check matrix and (the transpose of) a codeblock. If non-zero, the syndrome indicates that an error has occurred. For a binary code, it is the sum of the bit-numbers where errors have occurred.

at least one codeword into another. To simplify later discussions, a fourth letter, r, is used for the difference $n-k$. This quantity can be considered a measure of a code's redundancy. All four letters, n, k, d, and r are used in this dissertation for no other purpose than that defined above.

The designer of error-detection circuitry must specify the extra signals to be included in the codeblock. because the choice can have a significant impact on system cost, one generally will try to design the redundancy so as to maximize the probability of detecting the errors that are most likely to occur. The ability to do this is clearly dependent upon the pattern of errors to be encountered. If all error symptoms, including the "null" (*i.e.*, no error) one, were equally likely, then the probability of detecting an error would be directly proportional to the number of non-codewords and inversely proportional to the overall number of codeblock values. The probability of detecting an error would be

$$\frac{(\#\text{ of invalid codeblock values})}{(\#\text{ of codeblock values})}.$$

Of course all symptoms are not equally probable. If they were, then the codeblock would be useless—it would be equivalent to a random number generator! That is, there would be no statistical correlation between the original codeword and any codeblock value examined later.

A more reasonable assumption might be that the null symptom is very probable while the other symptoms are less probable. In this case, one would be interested in the conditional probability of detecting an error *when one occurs.* Assuming that all errors will either be reflected in the codeblock as a non-null symptom (or may be safely ignored), this probability is

$$\frac{(\#\text{ of invalid codeblock values})}{(\#\text{ of codeblock values}) - 1}.$$

In general, one is interested in the complementary probability, that of an error going undetected. I call this probability ψ; it is the value

$$\psi = \frac{(\#\text{ of codewords}) - 1}{(\#\text{ of codeblock values}) - 1}.$$

If one can isolate "check" bits from "data" bits (*i.e.*, the coding is *systematic* or *separable*), then this quantity can be expressed for an (n,k) code as

$$\psi = \frac{2^k - 1}{2^n - 1}.$$

For large values of k, one can simplify this expression with the approximation

$$\psi = 2^{-r}$$

(remembering that r is defined as $n-k$).

Traditional codes have attempted to reduce the probability of not detecting an error to less than ψ. While this is impossible for the uniformly distributed, unpatterned, error described above, it can be done if there are useful patterns to the errors that are most likely to occur. For example, in the case where $r = 1$, one can do a variety of things with the redundant bit. Two straightforward possibilities are to set it to zero or to set it to the "parity" of the data bits. For the totally unpatterned error (see Section 1-2.2), either method will miss errors with probability ψ (in this case 0.5). On the other hand, for independent bit failures (where a single-bit error is far more probable than a multi-bit error) the parity code will do substantially better than ψ while the constant-zero code will do worse. The former code would miss only double-bit, quadruple-bit, etc. failures, which would be relatively rare. The latter code, however, would miss any error which did not affect the check-bit itself.

Most codes capable of detecting multiple-errors achieve similar improvements over ψ through similar concentration on the most likely errors. Leung-Yan-Cheong and Hellman [47] and later Leung-Yan-Cheong, Barnes, and Friedman [48] discuss the attainability of ψ for various codes and error rates. They show that, under some circumstances, a few well-known codes will not keep the probability of an undetected error below ψ.

1-2.2. Unpatterned errors

The above discussion of ψ and uniformly distributed symptoms may at first seem naive. After all, most of the work in coding theory has sought to utilize patterns in the symptoms. By tailoring codes to detect the most likely errors, people have been able to reduce the probability of an undetected error substantially below ψ. Perhaps this emphasis has resulted not only from the existence of such patterns but also from the lack of interesting mathematics when such patterns do not exist. Any code of a given size, even one chosen at random, that has a unique codeword for each value to be transmitted will perform equally well in the face of the unpatterned error, missing detection with probability ψ. Little benefit can be provided by coding theory in such cases—hence the interest in codes for patterned errors.

The concept of an unpatterned error is specific to a given codeblock size. If one steps back to look at a bigger codeblock, treating the original codeblocks as characters in the larger codeblock, then the larger error pattern might resemble what is known as the *random* error distribution. The incidence of errors in any particular character under a random error distribution is statistically independent of that for the other characters. In this case one-character failures would be more common than two-character failures and so on. If the probability for any particular character being received erroneously is p, then the probability of an m-character failure is p^m. Thus, methods that

distinguish short-distance errors from longer ones, such as Hamming codes, can become useful. Even if the error symptoms in individual small codeblocks may seem uniformly distributed, the assumption that the null symptom (*i.e.*, lack of an error) is more probable than any other symptom implies that there is a pattern in the symptoms affecting the large codeblocks.

One assumption of this dissertation is that the unpatterned error is a reasonable model for many of the errors that actually arise in computer systems. While the causes of these errors may be patterned, the visible symptoms are not. Were a component common to all the bits of a codeblock to fail, for instance, one might induce errors in all the bits of the codeword at once. An example that comes to mind is a power supply "flicker." Several other failures might yield seemingly unpatterned results:

- fetching the wrong word from a storage module due to faulty address-decoding hardware;
- fetching an uninitialized variable;
- fetching data through a "dangling pointer;"
- strobing a bus-receiver register at the wrong time.

These are all examples of using the wrong word. In each case (except, perhaps, the last) the value fetched will roughly reflect the distribution of values in the system in general. If one assumes that all values are equally likely to be found in storage[3] and that an erroneous storage reference will access any one of these words, then clearly the value fetched will also appear random, thus behaving like an unpatterned error distribution.

1-2.3. External redundancy

Most study of coding has employed what I call *internal redundancy*. That is, all of the bits of the codeblock are transmitted or stored together. While this is reasonable for certain situations, there are also cases in which one can accomplish almost nothing using such a code. Consider trying to detect addressing failures in storage. No matter what code is used to store the data, fetches of the wrong word will go undetected unless one utilizes *external redundancy*. That is, without some redundant information that is not subject to the same addressing failure, there would be no way to spot an error that just selected a different (but properly encoded) word. This is because, in the absence of other errors, the bit-pattern found in an incorrect cell would be an acceptable codeword. The obvious

---

[3]Admittedly this is not actually the case; certain values such as zero seem to predominate. As will be seen in Section 2-2.2, however, techniques such as encryption can be used to make this distribution much more uniform.

solution to this problem would be to keep part of the codeblock elsewhere, so that an addressing failure would not affect all data. In this particular case, one could use bits that are present anyway: those indicating the address of the word. For instance, suppose that all locations whose address has odd parity were coded with odd parity and all locations whose address has even parity were coded with even parity. Then an addressing failure that errs by an odd number of bits would be detectable because the value fetched would have the wrong parity (assuming the storage system had worked reliably).

Another way to implement external redundancy might be to split the codeblock across two different storage modules that are not likely to fail simultaneously in a similar manner. Even if some of the wrong bits are fetched, the fact that not all of them come from the same wrong location allows traditional error-detecting codes (especially *burst-error* detecting codes) to function. The failure of one module would then look just like a multiple-bit storage failure.

Given that various forms of external redundancy have been sporadically used in various computer systems, it is surprising that this idea has often been overlooked in storage and communication systems. The oversight is probably due to an assumption that most errors will result from failures that affect the bits independently rather than from failures that affect all bits of a codeblock in common. While the assumption may be appropriate for the first attempt to control errors, one must consider common failures when trying to control second-order problems.

Interestingly enough, a single code can be considered as both internally and externally redundant, depending on the context. For instance, a residue-three code (in which the check bits of codewords equal the modulo-three residue of the data bits) might well act externally redundant with respect to checking for proper functioning of an adder yet act internally redundant with respect to checking that the correct operand was selected for addition. In the former case, the check bits pass through separate circuitry from the main adder and so can be used to check the operation of that adder. In the latter case, however, selection of the wrong operand would probably also imply selection of the wrong check bits and so no error could be detected.

1-2.4. Error correction

The next step after error detection is error correction. In communications systems a distinction is generally made between *forward* and *reverse* error-correction. Forward error-correction refers to situations in which sufficient redundancy is included in the original codeblock to permit recovery of the intended data. Each *correctable* invalid codeblock value is associated with the codeword most

likely to have been transmitted when that invalid value is received. In most cases the codeword assigned is the one "closest" to the erroneous value. If there is no such clear choice, receipt of that value indicates than an uncorrectable error has occurred. A typical example would be the extended Hamming code: those codeblock values that differ by exactly one bit from a codeword are corrected to that codeword; all others are just detected. The original choice of codewords is such that each no invalid value is "close" to more than one codeword.

Reverse error-correction, on the other hand, does not require such guessing as to the intended codeword. Instead, correction is performed by requesting retransmission. The sender is, therefore, obligated to retain a copy of each message until it is properly received. The name *reverse* is derived from the fact such correction requires a reverse communication channel from the recipient to the sender on which success or failure of the primary transmission can be indicated.

For computer storage applications, forward error correction has usually been considered the only kind possible. Reverse correction is normally impossible because, by the time a reader determines that a value in storage is questionable, the writer of that location has long since discarded the proper value—no other copies are normally retained when writing storage. It turns out, however, that situations exist in which this is not quite the case—duplicate copies (or sufficient data to reconstruct them) may be available. For example, there might be a copy of a storage page on backing storage. For another example, pointer-values in doubly-linked lists can be recovered by following the pointer chain for the opposite direction. While all of these copies (including the original) could be considered to collectively form a forward correction code, the actual reference pattern more closely resembles reverse correction. That is, the normal codeblock consists of just one copy which is checked for errors without any attempt at correction. Only upon detection of an error is "retransmission" of the other copies requested. The distinction is that the coding of the primary codeblock is used for detection purposes only.

Using (forward) error-correction can be hazardous. Many errors that exceed the correction ability of the code will slip by although they would have been caught if just error-detection were used. No code can detect all possible errors. If a particular failure changes the codeblock from one codeword to another, there will be no indication that the error has occurred. If forward correction is being attempted, things become even worse. Undetected failures will arise from two sources: symptoms that reach other codewords and symptoms that reach correctable codeblock values associated with those other codewords. That is, in addition to errors that result in codewords, correction causes one to miss errors that come close enough to other codewords to be corrected. One can imagine an n-dimensional sphere around each codeword. This sphere, whose radius is the

correction distance of the code, contains those codeblock values that correct to the codeword at the center. As the radius of this sphere of correctability increases, its n-dimensional volume increases very fast! Thus, the probability of any "large" error being detected is markedly diminished. For an untruncated Hamming code, it actually goes to zero—any error affecting more than one bit is missed. For the more common extended Hamming code, all errors that affect an odd number of bits and even a few that affect an even number of bits will go undetected. That is, once the error distance exceeds the guaranteed capability of the code, a Hamming code becomes completely impotent while an extended Hamming code becomes slightly worse than a simple parity code. Only if one can show that the probability of such errors is very low can one justify so increasing the exposure to miscorrection. MacWilliams [53 (p. 12)] mentions a test of data transmission over the telephone network in which:

> The bit error rate went up when the [error-correcting] code was used. The code detected the errors all right, but it corrected them all wrong.

1-3. Encryption

Most previous applications of encryption have been for communicating information via media that would otherwise be insecure. Because of its ability to seem to randomize data, however, encryption can also be useful for tagging. While secrecy is not important to my usage of encryption, the techniques used to implement it are essentially the same as those used for communication.

An extensive historical discussion of cryptography may be found in *The Codebreakers: The Story of Secret Writing* by Kahn [39]. For a more technical discussion, especially oriented toward modern techniques (which have changed dramatically with electronic computation), see the survey articles by Diffie and Hellman [13] and Lempel [46].

Two basic encryption methods are *substitution* and *transposition.* In the former, characters (or blocks of characters) of the *plaintext* or *cleartext* are replaced by characters of the *ciphertext.* For instance, the word *tagging* might be replaced by *ubhhjoh.*[4] The list of substitutions to be made is known as the *key*; it is the part of the encryption/decryption process that must be kept secret. Due to the redundancy typically present in data being encrypted,[5] single character substitution ciphers can usually be broken. An attacker can employ frequency analysis of the characters in a message, as well as similar analysis of adjacent pairs and so on. In English, for instance, *e* is by far the most common

[4] Solution of this trivial cipher is left to the reader.

[5] This is quite noticeable in natural language communications, where information-theoretic redundancy normally exceeds a factor of three. Military communications, which employ highly stereotypical words and phrases, are often much worse!

character and *th* is the most common digraph. A character or digraph with corresponding frequency in the ciphertext very probably can be so translated. Transposition ciphers rearrange the characters of a message according to some secret pattern. For example, *encryption* might become *nercpyitno.* Again the redundancy normally present in the text of messages often allows transposition ciphers to be broken. Thus, for a message written in English one might try those permutations that tend to bring the characters *t* and *h* together to form the digraph *th.*

Despite their individual weaknesses, combinations of substitution and transposition ciphers can be significantly stronger than the individual elements from which they are formed. DES, which is discussed below, is an example of such a *product* cipher.

Mathematically, the most secure encryption algorithm for use with digital data is known as a *one-time-pad.* In such a system, a different substitution cipher is used for each digit to be transmitted. Furthermore, there is no correlation between these ciphers—they are chosen in a statistically independent manner. The important property of a one-time-pad is that even a brute-force attack, examining all possible keys, will give the attacker no information whatsoever. If an n-bit binary message is intercepted, then trying all 2^n possible keys will just yield all 2^n possible cleartext messages. This obviously impedes the cryptanalyst, who, to make any progress, must find ways to eliminate some of the potential interpretations. In fact, if the key itself is not compromised then a one-time-pad provides absolute security. That is, no matter how much ciphertext or other information becomes available, nothing (other than an upper bound on the message length) can be learned from an encrypted message without knowing the corresponding key.

For transmitting binary messages, the choice of ciphers may be be represented as a bit-string where each bit denotes the cipher to be used for encrypting one bit of the message. The actual encryption process consists of nothing more than computing the *exclusive-or* of the key and the plaintext. Thus the hardware required can be both simple and fast. The main problem with one-time-pad systems is that they require as many bits of key as there are data to be encrypted. Furthermore, this key cannot be reused for any other messages—any reuse can significantly reduce security of the cipher. Clearly one cannot use such a scheme to store large amounts of data in a computer unless one can solve the equally difficult problem of storing the key! While acceptable for use in communications systems where one is only trying to improve real-time response (by shipping the key in advance by slow, secure means and the data by a faster medium), the large key of one-time-pads makes them impractical for most applications.

Any encryption algorithm that uses fewer bits of key than data to be transmitted is theoretically

vulnerable to attack. A thorough evaluation of this vulnerability from the point of view of information theory may be found in a classic article by Shannon [80]. On the other hand, large keys tend to increase the probability that the key itself will be compromised—big things can be harder to protect than small. To get around this problem, people have attempted to find encryption algorithms that, despite being theoretically breakable, are in fact reasonably secure. This is done by making the work necessary for an attacker computationally impractical.

None of the methods used before this century are resistant to attack by computer when enough text is available for cryptanalysis. Two systems that were used during World War II offered some resistance to machinery of the day, but eventually each proved vulnerable. One, the Hagelin device, resembles the one-time-pad described above (in this case using radix 26) except that a pseudo-random key generator was substituted for the truly random key essential to one-time-pads. It turns out that bias in the selection of individual digits of the key along with recognizable periodicities in the generated sequences allow one to recover the parameters controlling the key generator. The other World War II encryption method, referred to as rotor machines, provided higher levels of security but they too have been broken [95, 71, 76].

Two algorithms designed specifically for use with computers are Lucifer [83, 16] and its derivative, the federal Data Encryption Standard (DES) [65]. These are based on an early proposal by Shannon [80] for a product cipher composed of several layers of alternating substitution and transposition. While the net effect is equivalent to a substitution cipher, the size of the tables necessary to perform direct substitution make such an implementation impractical. Unlike a simple alphabetic substitution cipher which requires 26 5-bit entries, implementation of DES by table-lookup would require a table of 2^{64} 64-bit entries for each key that might be used—more storage than will ever be available. The compound implementation, however, can be produced quite efficiently in digital circuitry. While some controversy has surrounded the security of DES [12], the criticism has centered not on weakness of the general design but rather on the choice of one important parameter—the size of the key. Trivial changes to the definition could circumvent all the criticism to date.

Although not critical to this dissertation, some mention should be made of a more recent development, the concept of *public-key* algorithms [11, 77, 55], which has changed the way people think about encryption. Unlike previous schemes, in which a single key was used, separate keys are used for the encryption and decryption processes; it is assumed that knowledge of one does not make practical computation of the other. Using such systems, security no longer depends on the security of two parties and their mechanism for transmitting keys. Instead, one only need guarantee security

within one party and *reliability* of key communication. In fact, the encryption key can even be published openly without compromising the security of communications that use it! At this point in time, the throughput of these algorithms is not competitive with that of single-key systems such as DES. Even Rivest's special purpose chip [78] provides throughput more than four orders of magnitude less than that of a comparable DES implementation. For high bandwidth applications a hybrid seems most appropriate in which a public-key method is used only to exchange a *session-key*. A traditional single-key method employing this key would then be used for the main transmission. This effectively combines the speed of traditional algorithm with the advantages of the public-key algorithm.

For use with tagging, the most important property of encryption is its sensitivity to changes. If a value is encrypted with one key and then decrypted with another, then the resulting value will usually be quite different from the original. Some encryption functions are all sensitive to changes in the cleartext or ciphertext—a small change in one can be expected to show up as a large change in the other.

Chapter 2
Mechanisms

> "Oh, no, Eeyore," said Pooh. "Balloons are much too big to go into Pots. What you do with a balloon is, you hold the balloon——"
> "Not mine," said Eeyore proudly.
>
> A. A. Milne [57 (p. 86)]

The research reported here consists of two parts: the development of tagging methods and the examination of areas in which such tagging might be utilized. This chapter describes two different methods for including tags in memory cells. The first method merges the tag with error-checking bits already present in many computer words; the second utilizes encryption. For lack of better names, I call the two methods C-tagging and E-tagging, respectively. Both allow one to efficiently store tags with data that can later be compared against tags presented when fetching those data from storage.

Unlike previous uses of tagging, the proposed methods allow only for setting tag values when storing into memory and comparison when fetching. There is no way to inquire what tag is associated with a particular cell of storage. At best, one can compare it with a known value and determine either that it is different or that it *might* be the same. While equal values are always reported to be equal, so may a few unequal values.

In return for giving up the ability to read tags, one gains some advantage. Perhaps of most importance is the "cheap" storage of the tags. In fact, when merged with other codes, some bits of the tag require no extra bits of storage. In other cases, fewer bits would have to be stored than are checked in any particular comparison. Although information-theoretic arguments show that this must allow some incorrect values to be erroneously accepted, such occurrences should be rare because they are a second-order phenomenon.

A related benefit is the ability to handle arbitrary amounts of information in a single tag. Conventional tagging systems have difficulty adding extra bits as needed. If, after initial implementation, one wants to add new information to the tags, then major modifications to the

hardware may be necessary. Because of tag compaction, however, this is no problem for my compact tagging. New fields can be added at will. More detail appears in Section 2-4.1.

2-1. Traditional coding and the addition of tags (C-tagging)

The following discussion will show how a tag can be merged with error coding by a method that does not require the addition of extra bits. That is, the bits already present for memory-error coding will also be used to store the tag. Furthermore, the error-handling ability of the code will be minimally affected. As mentioned above, the tag is not of the sort that can be fetched. Instead, it is *checkable* by comparison with a tag value presented on each access to a particular memory cell. The term *tag-mismatch* will denote an error in which the tag presented during an access to memory does not match the one previously stored with the word.

Tags usable for such checking can be implemented nearly for "free" in computers that already include codes for detecting storage failures. The mechanism for so doing is best explained in stages. Since some readers may not be familiar with coding techniques, I will also briefly explain two traditional codes: parity and Hamming.

To make the discussion follow a reasonable progression, let us start with a non-redundant memory word which contains just data bits. Because all possible codeblock values are codewords, no errors are detectable. An eight-bit memory word will be represented as

D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8

where D_i denotes the i^{th} data bit. Although the examples in this section will provide for eight data bits, all of the techniques to be discussed can be extended to an arbitrary number of bits. Eight bits make the methodology apparent without overly cluttering the examples. The rest of this section alternates between explaining codes without tags and describing how tags can be added to those codes.

2-1.1. The parity code

The simplest useful redundant code is that formed by the addition of a parity bit, P, to the memory word. In this case the word would look like

P D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8.

The parity bit is generated when storing information into memory and checked when fetching it from

memory using the equation[6]

$$S = P \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8.$$

When storing, the equation is solved for ***P*** using the appropriate values for the data bits and zero for ***S***. When fetching, the equation is instead solved for ***S***, using the values fetched from storage for the other variables. If the resulting ***S***, which is called the *syndrome*, is non-zero, then an error has been detected. If, on the other hand, ***S*** is zero then it is assumed that no error occurred. There is no way to distinguish a situation in which sufficient errors have occurred to produce a zero syndrome from one in which no errors have occurred (but the original data bits had different values). Using this code, a change in the value of any single bit in storage will be detected; a change of several bits will only sometimes be detected. In particular, any change of an odd number of bits will be detected while a change of an even number of bits will not. If one imagines the bit-failures as happening sequentially, then each failure will toggle the syndrome between zero and non-zero; an odd number of failures will leave it non-zero, thereby allowing detection.

The probability of bits in memory failing is generally considered to be fairly low. Furthermore, failures are assumed to occur independently of each other. A multiple-bit failure is, therefore, assumed to be much less probable than a single-bit failure. Hence, under these assumptions, a parity code will detect the majority of errors that occur in such a memory. It will not be particularly effective, however, against the various unpatterned errors discussed in Section 1-2.2—half of them can be expected to result in a zero syndrome.

2-1.2. Adding a tag to the parity code

Let us now add a one-bit tag to each memory word. The value of this tag might be something determinable from the program-context on every memory access. For example, it could specify whether the word contains instructions or data.

The obvious way to add such a tag is to add an extra bit, ***T***, to the storage word and extend the syndrome equation appropriately:

$$S = P \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8 \oplus T.$$

Such a scheme allows the detection of two different sorts of errors: single-bit memory failures and tag-mismatches. The former is detected as before by computing the syndrome ***S*** (an internal

---

[6]Coding theorists typically avoid Boolean constants in such equations. Any of the equations in this section could have "⊕ 1" (or equivalent) appended to either side without significant effect on its meaning. Such a suffix will turn an "even" parity code into an "odd" one and vice versa. For abstract studies of systems with symmetric bit-failure probabilities this distinction is pointless.

redundancy check); the latter is detected by comparing T with a corresponding bit presented when fetching (an external redundancy check).

Suppose one chose not to store the tag bit explicitly with the data (but still include it in the parity generating/checking equation). To detect memory errors when the word is fetched, one clearly will need to know the value of T in order to solve the syndrome equation for S. This creates no problem because a copy of T is presented at fetch-time (for tag checking). Assuming that the tag value presented is the same as the original value of T, it does not matter that T was not stored along with the rest of word—memory errors can still be detected. On the other hand, should the wrong tag T be presented (*i.e.*, a tag-mismatch occurs), the situation would be just as if a stored bit had changed. In the absence of other errors, the syndrome would become non-zero. Admittedly, one cannot distinguish a storage error from a tag-mismatch but it is unlikely that this would be a critical problem. Higher-level recovery procedures, cognizant of the context, might be able to diagnose the most probable cause. Even if this is not possible in a particular situation (and so the two possible explanations for a given failure cannot be distinguished), the higher-level procedures will at least know that "something is wrong" and can back up or abort the computation appropriately. Preventing the spread of the effects of an error is critical.

If both a single-bit memory-failure and a tag-mismatch appear simultaneously, then the syndrome computed will be zero and so the errors will not be detected. Note, however, that this is really a double error and so is beyond the the error detection ability of a parity code. More is said about this below in Section 2-1.5 in the discussion of the corresponding situation for Hamming codes.

2-1.3. The Hamming code

Sometimes detection of failures is not sufficient—one would like also to "correct" the error. To do this requires a slightly more complicated code. The one most commonly used is the Hamming code, first published in 1950 [24]. With this code (but without tagging), a storage word might look like

$$H_1\ H_2\ D_3\ H_4\ D_5\ D_6\ D_7\ H_8\ D_9\ D_{10}\ D_{11}\ D_{12}$$

where each H_i denotes a check bit. Although there are still eight data bits, they have been slightly renumbered; this allows a single numbering system to be used for both data and check bits. The equations used to generate and check the word are as follows:

$$S_1 = H_1 \oplus D_3 \oplus D_5 \oplus D_7 \oplus D_9 \oplus D_{11}$$

$$S_2 = H_2 \oplus D_3 \oplus D_6 \oplus D_7 \oplus D_{10} \oplus D_{11}$$

$$S_4 = H_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_{12}$$

$$S_8 = H_8 \oplus D_9 \oplus D_{10} \oplus D_{11} \oplus D_{12}.$$

Again, the four syndrome bits, S_1 through S_8, are set to zero so that one can solve for H_1 through H_8 for storage in memory. During a fetch from memory, the syndrome bits are determined from the bits fetched using the same equations. If any of the resulting syndrome values are non-zero, then a detectable error has occurred. Now, however, there is enough information to determine *which* bit is wrong, assuming that only one bit has changed. In particular, if one treats the syndrome as a four-bit binary number $S_8S_4S_2S_1$ then that number will be the bit number of the erroneous bit. This happens because each of the equations was carefully chosen so that it included only those bits whose position included a particular power of two in their representations. That is, each of the position numbers in the first equation is odd and so has a one in the "units" bit of its binary representation. Similarly, the "twos" bit is set in each of the position numbers of the second equation, the "fours" bit in the third equation, and the "eights" bit in the final equation. Any change in one memory bit will affect exactly those equations that are used to determine syndrome bits corresponding to its position number. For example, if bit D_{12} were to change then S_4 and S_8 would become non-zero. This would yield a syndrome value

$$S_8S_4S_2S_1 = 1100_2 = 12_{10}$$

thereby indicating that bit twelve is incorrect and should be complemented.

Similar to parity codes, Hamming codes leave multiple-bit failures undetected. Worse, however, most multiple-bit errors are improperly "corrected" by a Hamming code. If more than one bit changes, then the syndrome indicates the (exclusive-or) sum of all the corresponding bit-numbers. The syndrome might indicate that one particular bit is wrong when in fact several others are. For example, a failure of D_5 and D_6 would produce a syndrome indicating that D_3 had changed. In this case the correction mechanism would increase the number of erroneous bits, not decrease it. Since this can happen only when a multiple-bit failure has occurred, the assumption that bit failures occur independently implies that it should not be a frequent phenomenon. The ability to correct the more common single-bit error often justifies this sacrifice. Besides, for practical purposes, a word with $m+1$ errors is rarely any worse than one containing only m errors. For most programming uses, a word is either correct or incorrect; there are usually no useful gradations of incorrectness in digital systems. Even if there are some gradations, it is unlikely that Hamming distance would be an appropriate way to measure them.

The problem with error-correction is that high level processes are not notified whether the data really are valid. No bit failures, other than the single-bit ones, will be handled properly. It is a property of codes that one must sacrifice a significant portion of their error-detection ability in order to gain any error-correction ability. If the same code were used for detection rather than correction, it

would detect *all* single, *all* double, and *most*[7] other failures!

It should be noted that the example above shows a *truncated* Hamming code. It is derived from the *full* code which has eleven data bits and four redundant bits by ignoring three of the data bits. For every positive integer r there is a full Hamming code with $2^r - 1$ bits. Of these, $2^r - r - 1$ are data bits and r are check bits. Since the number of data bits in a full Hamming code rarely coincides with the word-length of modern computers, most practical systems use truncated Hamming codes. This truncation actually has one slight benefit: some double-bit errors will not cause miscorrection. This is because they produce syndromes that indicate bit positions that are not being used. Note, however, that only *some* double-bit errors fall into this category; most will cause improper "correction."

2-1.4. The extended Hamming code

It is possible to achieve a compromise between correction and detection by making the Hamming code one bit longer. Any odd-distance linear code (such as the Hamming codes) can be extended to the next higher even-distance code by adding an overall parity bit. For the example, the computations can be kept convenient by first adding 16 to each bit number:

$$P_{16}\ H_{17}\ H_{18}\ D_{19}\ H_{20}\ D_{21}\ D_{22}\ D_{23}\ H_{24}\ D_{25}\ D_{26}\ D_{27}\ D_{28}.$$

One then gets the storage word

$$P_{16}\ H_{17}\ H_{18}\ D_{19}\ H_{20}\ D_{21}\ D_{22}\ D_{23}\ H_{24}\ D_{25}\ D_{26}\ D_{27}\ D_{28}$$

and the new syndrome equation

$$S_{16} = P_{16} \oplus H_{17} \oplus H_{18} \oplus D_{19} \oplus H_{20} \oplus D_{21} \oplus D_{22} \oplus D_{23} \oplus H_{24} \oplus D_{25} \oplus D_{26} \oplus D_{27} \oplus D_{28}$$

where P_{16} is the overall parity bit.[8]

As before, if any of the bits of S are computed to be non-zero when fetching, then a memory failure has occurred. If S_{16} is zero then the error must be in an even number of bits (because all memory bits contribute to the computation of S_{16}). Therefore, all double-bit errors will produce syndromes between 00001_2 and 01111_2. Because the valid bit-positions are numbered 16 through 31, these values do not indicate any of the bits in the word stored in memory and so the code can detect

---

[7] Very roughly, $1 - \psi$

[8] By replacing this equation with the sum of it and the four preceding equations, one could obtain the slightly shorter equivalent equation

$$S_{16} = S_1 \oplus S_2 \oplus S_4 \oplus S_8 \oplus P_{16} \oplus D_{19} \oplus D_{21} \oplus D_{22} \oplus D_{25} \oplus D_{26} \oplus D_{28}.$$

This optimization is particularly useful when writing a word in storage because one then sets all the bits of S to zero, thereby reducing the number of terms further:

$$P_{16} = D_{19} \oplus D_{21} \oplus D_{22} \oplus D_{25} \oplus D_{26} \oplus D_{28}.$$

double-bit errors. Single-bit errors will produce syndromes ranging from 10000_2 through 11111_2 and so will be correctable as before.

Since such an *extended Hamming code* is capable of correcting all single-bit failures and detecting all double-bit failures, it is often described as "Single Error Correcting, Double Error Detecting" (SEC-DED). While most higher order odd multiple failures are miscorrected, most higher order even multiple failures are detected. Because an extended Hamming codes requires only one more bit to implement than the non-extended Hamming code but is "fooled" far less often, one almost never sees the latter actually used.

2-1.5. Adding a tag to the extended Hamming code

Just as one can add a tag to a parity code (see Section 2-1.2), so can one add a tag to an extended Hamming code. This addition is best described in two steps: first I will add a sixteen-bit tag which itself is encoded using a one-of-sixteen code; later I will show that the corresponding four-bit binary tag can be used directly. This two-step explanation has generally made the extension easier to comprehend.

Let us change the first five syndrome equations to include a sixteen-bit tag, represented by T_{16} through T_{31}:

$$\begin{aligned}
S_1 &= H_{17} \oplus T_{17} \oplus D_{19} \oplus T_{19} \oplus D_{21} \oplus T_{21} \oplus D_{23} \oplus T_{23} \oplus D_{25} \oplus T_{25} \\
&\quad \oplus D_{27} \oplus T_{27} \oplus T_{29} \oplus T_{31} \\
S_2 &= H_{18} \oplus T_{18} \oplus D_{19} \oplus T_{19} \oplus D_{22} \oplus T_{22} \oplus D_{23} \oplus T_{23} \oplus D_{26} \oplus T_{26} \\
&\quad \oplus D_{27} \oplus T_{27} \oplus T_{30} \oplus T_{31} \\
S_4 &= H_{20} \oplus T_{20} \oplus D_{21} \oplus T_{21} \oplus D_{22} \oplus T_{22} \oplus D_{23} \oplus T_{23} \oplus D_{28} \oplus T_{28} \\
&\quad \oplus T_{29} \oplus T_{30} \oplus T_{31} \\
S_8 &= H_{24} \oplus T_{24} \oplus D_{25} \oplus T_{25} \oplus D_{26} \oplus T_{26} \oplus D_{27} \oplus T_{27} \oplus D_{28} \oplus T_{28} \\
&\quad \oplus T_{29} \oplus T_{30} \oplus T_{31} \\
S_{16} &= P_{16} \oplus T_{16} \oplus H_{17} \oplus T_{17} \oplus H_{18} \oplus T_{18} \oplus D_{19} \oplus T_{19} \oplus H_{20} \oplus T_{20} \\
&\quad \oplus D_{21} \oplus T_{21} \oplus D_{22} \oplus T_{22} \oplus D_{23} \oplus T_{23} \oplus H_{24} \oplus T_{24} \oplus D_{25} \\
&\quad \oplus T_{25} \oplus D_{26} \oplus T_{26} \oplus D_{27} \oplus T_{26} \oplus T_{28} \oplus T_{28} \oplus T_{29} \oplus T_{30} \\
&\quad \oplus T_{31}.
\end{aligned}$$

As before, one will not explicitly store the tag bits; they will be presented on each access.

Since the tag is defined to be a one-of-sixteen code, the parity of the whole tag will always be odd unless an error occurs in the tag. If one further assumes that such errors occur only while words are stored in memory, then not storing the tag will obviously prevent such errors. Therefore the sum of all the T terms in the equation defining S_{16} may be replaced by the constant 1. Since this is a

constant, it may be dropped from the computation without effect. Thus the last equation can be simplified to its original form:

$$S_{16} = P_{16} \oplus H_{17} \oplus H_{18} \oplus D_{19} \oplus H_{20} \oplus D_{21} \oplus D_{22} \oplus D_{23} \oplus H_{24} \oplus D_{25} \oplus D_{26} \oplus D_{27} \oplus D_{28}.$$

How do the modifications above of the extended Hamming code behave in practice? If the same tag value is presented when fetching a word from storage as was presented when storing that word, then clearly the changes will have no effect—only changes between the bit values used when encoding and those used when decoding can affect the syndrome. Should a memory failure occur, the same detection/correction would be provided as when no tagging is used: single errors would be correctable; double errors would be detectable; and higher order errors would be beyond the capacity of the code. On the other hand, if an incorrect tag were presented it would masquerade as a memory failure. In particular, each tag bit in error would yield symptoms equivalent to those of the correspondingly numbered bit in the base code. In isolation, there would be no way to tell the difference between an incorrect value being presented for tag bit T_{23} and an incorrect value being fetched from memory for data bit D_{23}. Since the code is defined to be a one-of-sixteen code, however, it is guaranteed that any change in the tag will concern exactly two bits (one cleared and the other set). Therefore, in the absence of true memory failures, a tag-mismatch will be detected as if a double-bit memory failure had changed those two bits.

This resemblance of tag-mismatches to double-bit memory failures carries through to the case in which a tag-mismatch and a memory failure occur together. A single-bit memory failure combined with a tag error will behave like a triple-bit memory failure. Since these are beyond the scope of extended Hamming codes, the error will not be detected and improper correction may be done. Similarly, a double-bit memory failure together with a tag-mismatch will act like a quadruple-bit memory failure which will probably (but not always) be detected. Such combinations of a tag-mismatch and a memory failure should be rare; perhaps it is not worth worrying about detecting that situation. Remember that any error-detection scheme has its limitations. Adding detection of tag-mismatches, a first-order phenomenon, surely compensates for sacrificing detectability of certain second-order phenomena. Assuming that the memory failure rate is not unusually high, this trade-off will yield a net gain. If the memory failure rate is high, one probably should have been using a code better than SEC-DED in the first place.

As stated at the beginning of this section, one need not encode the tag as a one-of-sixteen code. Instead one can use a normal four-bit binary code. For the extended Hamming code this substitution of $T_8'T_4'T_2'T_1'$ is straightforward:

$$S_1 = H_{17} \oplus D_{19} \oplus D_{21} \oplus D_{23} \oplus D_{25} \oplus D_{27} \oplus T'_1$$

$$S_2 = H_{18} \oplus D_{19} \oplus D_{22} \oplus D_{23} \oplus D_{26} \oplus D_{27} \oplus T'_2$$

$$S_4 = H_{20} \oplus D_{21} \oplus D_{22} \oplus D_{23} \oplus D_{28} \oplus T'_4$$

$$S_8 = H_{24} \oplus D_{25} \oplus D_{26} \oplus D_{27} \oplus D_{28} \oplus T'_8$$

$$S_{16} = P_{16} \oplus H_{17} \oplus H_{18} \oplus D_{19} \oplus H_{20} \oplus D_{21} \oplus D_{22} \oplus D_{23} \oplus H_{24} \oplus D_{25} \oplus D_{26} \oplus D_{27} \oplus D_{28}.$$

For example, replacing the one-of-sixteen tag that had only T_5 non-zero with the corresponding four-bit binary tag $T'_8T'_4T'_2T'_1 = 0101_2$ would not affect the functionality of the mechanism at all but would somewhat simplify the equations used.

2-1.6. Hsiao's modified Hamming code

There are actually two distinct methods for deriving the extended Hamming code (without tagging). The traditional method is that shown above: adding an overall parity bit to a distance three Hamming code. An alternative derivation starts with the next longer Hamming code and truncates it to yield a code with a minimum distance of four. For instance, let us start with a Hamming code whose bits are numbered 1 through 31 (twenty-six data bits and five check bits). If one eliminates bits 1 through 15, one will be left with the bits of the extended Hamming code with eleven data bits. Any double-bit error will be detectable because the Boolean (exclusive-or) sum of two bit position numbers must have a zero in the "sixteens" bit. The extended Hamming code described in Section 2-1.4 is exactly equivalent to this except for the further truncation to store only eight data bits.

Hsiao [32] noticed that one could do the first truncation in a slightly different manner which has a few advantages (which are not relevant to this discussion). Since many implementations of SEC-DED in actual hardware have used his techniques, it is important that I demonstrate compatibility with tagging.

Instead of discarding positions 1 through 15, Hsiao discards positions 3, 5, 6, 9, 10, 12, 15, 17, 18, 20, 23, 24, 27, 29, and 30. He retains only those bits whose position numbers (represented in binary) have odd parity. For eleven data bits, his code would use the equations

$$S_1 = H_1 \oplus D_7 \oplus D_{11} \oplus D_{13} \oplus D_{19} \oplus D_{21} \oplus D_{25} \oplus D_{31}$$

$$S_2 = H_2 \oplus D_7 \oplus D_{11} \oplus D_{14} \oplus D_{19} \oplus D_{22} \oplus D_{26} \oplus D_{31}$$

$$S_4 = H_4 \oplus D_7 \oplus D_{13} \oplus D_{14} \oplus D_{21} \oplus D_{22} \oplus D_{28} \oplus D_{31}$$

$$S_8 = H_8 \oplus D_{11} \oplus D_{13} \oplus D_{14} \oplus D_{25} \oplus D_{26} \oplus D_{28} \oplus D_{31}$$

$$S_{16} = H_{16} \oplus D_{19} \oplus D_{21} \oplus D_{22} \oplus D_{25} \oplus D_{26} \oplus D_{28} \oplus D_{31}.$$

This still allows detection of double-bit errors, but in a slightly more subtle manner. If the binary numbers X and Y have weights $W(X)$ and $W(Y)$ respectively, then the weight of their Boolean sum will be

$$W(X \oplus Y) = W(X) + W(Y) - 2W(X \wedge Y).$$

If X and Y each have odd parity then $W(X \oplus Y)$ must be even (because *odd + odd − even* is *even*). Therefore any double error cannot produce a syndrome that will be confused with that of a single-bit error. This makes the code usable for correcting single-bit errors and detecting double-bit errors.

Tagging can be added to Hsiao's modified code without much difficulty. In particular, by using a five-bit tag with even parity in place of a four-bit tag, one can still detect tag-mismatches as if they were double-bit memory failures. Alternatively, one could accomplish the equivalent by continuing to use four-bit tags but changing some of the syndrome equations to include two tag bits each. Thus, for storing eight data bits, one might use the equations

$$S_1 = H_1 \oplus D_{11} \oplus D_{13} \oplus D_{19} \oplus D_{21} \oplus D_{25} \oplus T'_1$$

$$S_2 = H_2 \oplus D_{11} \oplus D_{14} \oplus D_{19} \oplus D_{22} \oplus D_{26} \oplus T'_1 \oplus T'_2$$

$$S_4 = H_4 \oplus D_{13} \oplus D_{14} \oplus D_{21} \oplus D_{22} \oplus T'_2 \oplus T'_4$$

$$S_8 = H_8 \oplus D_{11} \oplus D_{13} \oplus D_{14} \oplus D_{25} \oplus D_{26} \oplus T'_4 \oplus T'_8$$

$$S_{16} = H_{16} \oplus D_{19} \oplus D_{21} \oplus D_{22} \oplus D_{25} \oplus D_{26} \oplus T'_8.$$

The underlying mathematics are briefly discussed in Appendix B.

2-1.7. Codes with greater minimum distance

It should be obvious that similar methods might be applicable to codes with minimum distance greater than the Hamming codes. An (n,k,d) code would normally be able to detect up to $d-1$ independent bit failures. If used for correction of up to c errors (where $2c < d$), this detection limit drops to $d-c-1$. To add a tag to such a code, something is needed that, if changed, causes one to exceed the correction limit c but not exceed the detection limit $d-c-1$. Therefore the tag itself must be precoded so that all possible changes in it have distance between these limits. For detecting memory bit failures in combination with tag mismatches these bounds are even tighter. While there are no simple rules for generating such subcodes with minimal and maximal distances, appropriate codes for specific applications should be findable. It is beyond the scope of this dissertation to construct these.

2-1.8. Previous work

In retrospect, one can see that some *ad hoc* effort toward combining tags with other error-detecting codes was included in an early Univac machine [79] and some Bell Telephone Electronic Switching System (ESS) computers [89]. The Univac III stored the parity of a word's address along with the data bits (and residue-three check bits) for that word. Thus, if a single bit failed in an address, the hardware would be able to detect the error. In the Number 1 ESS, the "program store" (read-only memory) used an overall parity check that covered both the address and the data bits. A Hamming code check was included for the data bits. For the "call store" (read-write memory) two parity bits were employed; one covered the bits of the address and the other both the address and data together. All of these uses are instances of physical address tagging, as proposed below in Section 3-1.

In the Number 2 ESS, many of the checks used in the earlier machine were dropped because it had been found that comparison of dual processors running identical programs and diagnostics provided sufficient detection of failures for the application. For this reason, error checking in the "program store" is reduced to a simple parity check. The machine apparently does not do any explicit checking of addressing logic as had been done in the Univac III and the Number 1 ESS. One unusual coding feature, however, is that the words in this store are divided into two classes: *instructions* and *translation data.* The former are stored with odd parity and the latter with even. That way, attempts to access a word in the wrong class for a given context are detected and trapped. Note that this can be viewed as an instance of C-tagging. In this particular case, each tag indicates whether the corresponding word is used as an instruction or as translation data. Accessing a word in the wrong class produces a tag-mismatch. This is a degenerate form of the type-tagging proposed later in Section 3-2.

Finally, it should be noted that the original implementation of WATFOR [81] took advantage of an unusual feature of the IBM 7040/44: the ability to set the parity bit of a word intentionally to the wrong value for the data in the rest of the word. Using this capability, the load-and-go system could "tag" each word of data storage with one bit. At the start of execution, all of these tags were set to indicate *uninitialized* (improper parity). Because subsequent memory fetches and stores asserted that the tag should indicate *initialized* (proper parity), a tag-mismatch would result upon an attempt to read a variable that had not been set by the program. As for the C-tagging proposed here, higher-level analysis (in this case by the user, presumably) was required to distinguish such mismatches from genuine memory failures.

2-2. Tagging using encryption (E-tagging)

Thus far in this chapter, I have described a mechanism that allows the storage of a tag with memory words. It is important to note that there might be several functions other than C-tagging that can accomplish the goal of merging tags with data in an efficient manner. The requirements for such a function include:

- Given tag and data values, one can compute a "combined value" to store in memory.
- Given a tag and a "combined value," one can recover the original data value.
- It can (probably) be detected if the wrong tag is presented when attempting to recover the original data value.

Another tagging method that can meet these requirements uses encryption.

If one chooses a reasonably secure encryption function, one can assume that decryption with the wrong key will produce a value quite different from the original plaintext. Therefore, if one uses the tag as an encryption key and the data to be stored (plus some redundancy) as the plaintext, then after decryption with the wrong key a check of this redundancy should reveal the error. Note that this redundancy can be very simple if an encryption function is used that sufficiently intermingles the data bits. For example, padding the data with some "must-be-zero" bits will suffice. For other ciphers, especially those that process bits independently of each other (such as simple one-time-pads), one must add data-dependent values (such as a checksum) that will allow detection of minor changes to the ciphertext. Unlike C-tags, E-tags are *implicit* and are not stored with the data. Therefore they can be substantially longer without mandating extra storage. As for explicit tags, however, the probability of not detecting a tag mismatch is ψ. That is, the error-detection ability still depends primarily on the number of redundant bits added to the word, not the number of tag bits being checked.

2-2.1. Previous work

The idea of using encryption for error detection is not new. Cryptographers have long recognized that many codes and ciphers can be used not only to keep information secret but also to authenticate it. Forged and corrupted messages ought to decrypt to "garbage" unless the code has been broken. Gilbert, MacWilliams, and Sloane [21] looked at codes specifically designed to detect forgery. Later, Gligor and Lindsay [22], Chaum and Fabry [9], and Needham [67] pointed out that

cryptography could be used in implementing capability[9] systems.

2-2.2. Exaggeration of undetected errors

Using encryption for error detection has an additional benefit which might be helpful in many situations. Even if a tag mismatch is not detected, which can be expected to happen with probability ψ, the data decrypted using an incorrect key will be quite different from those originally stored. Many encryption methods, such as DES, are also quite sensitive to changes in the the text being processed. That is, even a small change in the cleartext or ciphertext can cause a large change in the resulting ciphertext or cleartext, respectively. In other words, the value of each output bit is strongly dependent upon the value of all the input bits—ideally any change in an input bit would complement each of the output bits with probability 0.5. These two sensitivities—to changes in the key and to changes in the data—can be used to obtain yet another level of error detection.

The net effect of these sensitivities is that errors will often be exaggerated by E-tagging. Even when the primary check on the tag fails to discover an error, it is possible that higher-level redundancies, already present in many programs, will allow detection. In particular, data structures are often coded less than optimally. For instance, a variable is often stored using a full word, even if the number of values that that variable can take on is more limited than the word size allows. In a one-bit Boolean value, all but one of the bits of the word are redundant. It would be fairly easy to check them for consistency. Similarly, the high-order bits of an integer with restricted range would always match its sign. Whenever a *case* or *subscript* operation is performed, these bits can be checked by verifying that the value is within the intended range. Range checks by explicit code, compiler-inserted checks, or even tagging (see Section 3-4.1) can often catch errors that sneak past the primary tag check.

Pointers provide an extra level of tag-checking: if a pointer-value is scrambled for some reason, the tag-checking performed when fetching the word indicated by the pointer (as opposed to fetching the pointer itself) will probably catch the error. In fact, if the program uses only a small portion of the address space then randomization of a pointer value will often yield an illegal address which cannot be fetched at all.

---

[9] Capabilities, which were first mentioned in the literature by Dennis and Van Horn [10], are generalized pointers. The principal distinguishing characteristic is that a capability includes access-control information. These extra data can be used to restrict the operations that may be performed using a given capability. Furthermore, to ensure that these restrictions are obeyed, capabilities must be implemented in an unforgeable (and tamperproof) manner.

Taylor, Morgan, and Black [86, 87] discuss some other instances of software redundancy. In fact, they propose various data structures which are specifically designed to allow detection and correction of errors. It appears that a combination of their robust data structures with my tagging might be quite effective.

2-3. Combining explicit tagging with encryption

The two different methods of tagging described above, C-tagging and E-tagging, are not incompatible. There is no reason that a word cannot have an explicit tag added and then be encrypted with an implicit tag. The explicit tag could even take the place of the "must-be-zero" bits mentioned above in Section 2-2. Alternatively, one could encrypt a word and then add the explicit tag.

Both of these methods—adding the tag either before or after encryption—have limitations. In the former case, use of an extended Hamming code for correcting single-bit errors would no longer be practical. This is because any memory failures that occurred in the encrypted word would probably be exaggerated by the decryption function to multiple-bit failures in the cleartext. While still useful for detecting errors (subject to the ψ limitation), correction would only be possible for errors occurring before encryption or after decryption (*i.e.*, in the cleartext, not the ciphertext).

In the latter case, in which one encrypts first and then adds the tag, there would be no possibility of tag-mismatch detection attributable to the encryption. This is because one must add extra bits *before* encryption to gain this capability. Without some redundancy, error detection is impossible. On the other hand, the tag-mismatch detection capability of the explicit tag and the error exaggeration property of encryption would still be present. When an error is not properly detected or corrected by the explicit tag, one would expect that the word presented for decryption would differ slightly from the correct ciphertext. Therefore the decrypted word would probably differ greatly from the correct plaintext, which is the desired effect.

If one employs an encryption algorithm that allows a Hamming tag to be carried alongside the data during the encryption process (such as is described in Appendix A for DES), then a reasonable combination of encryption with explicit tagging might first add the tag and then encrypt. In the case, however, only the original data and not the tag would be encrypted. Tag mismatch checking could be done for either the original or the encrypted word. Furthermore, if incremental changes to the tag (as will be described in Section 3-1.1.1) are necessary, they too can be done on either side of the encryption fence. While this flexibility may at first seem unnecessary, it can be important for

maintaining reasonable performance—at times it may be desirable to store unencrypted words rather than the corresponding ciphertext.

2-4. Hashing of tags

Traditional tagged architectures have used a tag that is big enough to contain all the information to be checked. For example, there might be a unique tag for each hardware type. One of the principal contentions of this dissertation is that one can store fewer bits of information with a tagged cell than are being checked. By so doing, one can check many different assertions at minimal cost.

An important concept that one must accept before continuing is that no design has to work all of the time. In fact, no error-detecting scheme that uses a finite number of redundant bits can cover all errors. At best, it can detect failures that produce a certain class of symptoms. If an error occurs that transforms one codeword into another, then that error cannot be detected. (Any system that might claim to do so must be using other information and so has a bigger codeblock than stated.) There will always be a non-zero probability that some failure will produce an undetectable symptom. Of course, one tries to orient error coverage toward the most common errors, missing only the less common ones.

With this in mind, it should should be clear that neither extreme, using too little or to much redundancy, is desirable. In the former case, one will miss errors that could have been caught; in the latter case, the extra error detection provided by excess check bits will be marginal. Choice of a reasonable middle-ground is an engineering decision that must be tailored to particular circumstances. One can, however, look at some typical costs involved and thereby get an intuitive feeling for the situation. The following chart shows the incremental cost (in bits of storage) required for various combinations of word-sizes and detection probabilities:

		Added overhead for various word-sizes			
r	ψ	(k=8)	(k=16)	(k=32)	(k=64)
1	0.5	13%	6%	3%	2%
2	0.25	25%	13%	6%	3%
4	0.063	50%	25%	13%	6%
6	0.016	75%	38%	19%	9%
8	0.0039	100%	50%	25%	13%
12	0.00024	150%	75%	38%	19%
16	0.000015	200%	100%	50%	25%

If one assumes that a SEC-DED code, such as the extended Hamming code, is already used with each storage word, then the incremental overheads can be decreased significantly. Part of this is due to there being more bits per word initially (and so the denominator of the fraction is larger). More significant, however, is the ability to merge some of the tag bits with the Hamming check-bits, as

explained in Section 2-1.5. The resulting chart looks quite different from the preceding one:

		Added overhead for various word-sizes			
r	ψ	(k=8)	(k=16)	(k=32)	(k=64)
1	0.5	-	-	-	-
2	0.25	-	-	-	-
4	0.063	-	-	-	-
6	0.016	15%	5%	-	-
8	0.0039	31%	14%	5%	1%
12	0.00024	62%	32%	15%	7%
16	0.000015	92%	50%	26%	13%

Those entries indicated by "–" are especially attractive because no new bits at all need be added for tagging. The architecture described in Chapter 5, for example, uses a 6-bit tag with no extra storage requirements even though ψ is less than 0.02.

The error-detecting methods proposed in this dissertation will detect "most" occurrences of the targeted errors. Although a few (as indicated by the fraction ψ) errors will slip by, this need not be of major concern—no method will detect all errors anyway. More significant is that a large fraction of the errors can be detected at reasonable cost.

In many instances a particular failure might be detected by any of several different checks. One instance of such compound checking is the use of an addressing value (*e.g.*, a pointer or subscript) that itself has been tagged. Not only will a check be made on this value but also on the item referenced using the value. In general this will decrease the probability of missing an error by a factor of ψ for each level of indirection. Multiple executions of the same program can also be a form of compound checking. As long as the tags can be made to differ quasi-randomly on successive runs, the probability of missing an error will decrease geometrically. Even small tags can be effective when this probability is reduced to a power of ψ. See also the discussion of "resonance" in Section 4-11.

One will often want to include a lot of information, perhaps several different tag components, in the tags to be checked. In order to store only a small tag with the data, some form of reduction is necessary. This compaction should use information from all of the components so that any change in a component will usually cause a corresponding change in the derived tag. Programmers already have a method for doing this: *hashing*. A *hash* function is one that maps from a large domain onto a relatively small range. A "good" hash function "spreads" the results uniformly throughout its range when presented with a "typical" set of inputs.

In the case of tagging, one would like to detect a difference in the hashed result even if only one of the input tag components is changed. Therefore, for this application, it is desirable that the result depend strongly on *each* of the tag components. This is in contrast to some situations, such as

symbol-table management, in which the hash function can often depend only upon a subset of the bits in its input without greatly reducing performance.

2-4.1. Incremental hashing

Various applications for tagging are examined in Chapter 3. To use more than one of these, one must combine several tag components into a single compound tag. In fact, some of the individual tag components may require more bits than the chosen value for r. Therefore, hashing is mandatory. One might even have an arbitrary number of tags rather than some fixed number. For example, there is no fixed upper bound on the number of abstract types that can be associated with an object. Associated with each of these types would be a distinct tag value. Therefore, one must design the hash function so that it can be applied to an arbitrary number of inputs.

Dynamically varying the number of inputs to the hash function might be difficult if they must all be presented at once. In addition, each potential accessor of an object would have to accumulate a list of all the type codes to be used when accessing an object. Such lists could consume far more storage space than would be saved by the compact final tags. This problem can avoided, however, by making special choice of the hash function. In particular, let the function be one that can be computed incrementally with short intermediate results. For instance, if the hash function is the sum (with some modulus or moduli to limit the size of the results) of all the type codes of an object then one can merge one tag at a time into the hashed tag. At each level of abstraction one need add only one part of the tag, the type code for the corresponding type. The number of levels need not be given a fixed bound; such an accumulation can be iterated an arbitrary number of times. This cumulative computation has the additional advantage that at no point does one have to maintain large amounts of information. At worst, one would require registers for the current tag and the old and new sums.

A typical incremental hash could be implemented as an expression like

$$f(T_0 \star T_1 \star T_2 \star \ldots \star T_n)$$

where $\star$ is a binary operator that is evaluated from left to right and f is a function that maps the accumulated result just before use.

In addition to the normal requirements for a hash function, such as uniform distribution of the results, there are two properties that the $\star$ operator should have to ease use with tagging.

One such useful property is the ability to combine several tag components in advance of adding them to the incremental hash value (see Section 3-2.3.2). To do so, the definition of the $\star$ operator

must allow combination of tag components in other than strict left-to-right order without changing the value produced. Although this might seem to imply that ★ must be associative, it turns out that a weaker property, which I call *quasi-associativity*, is sufficient. Whereas the definition of associativity is

$$(\forall a,b,c)\ (a \bigstar b) \bigstar c = a \bigstar (b \bigstar c),$$

the definition of quasi-associativity[10,11] is the existence of another operator, ▮, such that

$$(\forall a,b,c)\ (a \bigstar b) \bigstar c = a \bigstar (b \blacksquare c).$$

Clearly any associative operator is also quasi-associative because the ★ operator itself would suffice for ▮. Two examples of commonly used operators that are not associative but are quasi-associative are traditional arithmetic subtraction and exponentiation. In the former case ▮ would be addition and in the latter it would be multiplication. Note that an operator can be quasi-associative even if it would be meaningless to just shuffle the parentheses. For example, multiplication of a vector by a scalar is quasi-associative (using scalar multiplication for ▮) even though the regrouping necessary for associativity,

$$v \bigstar (s_1 \bigstar s_2),$$

would be meaningless.

The other useful property for ★ is invertibility. In fact, several different forms of invertibility are relevant, but it is not worth describing them in detail. The primary uses are in type-sealing and -unsealing (see Section 3-2.3) and retagging during storage relocation (see Section 3-1.1.1).

Despite the wide range of potential functions, for the applications proposed here a complex ★ operator is not needed. A simple and probably sufficient hash function for use with binary hardware is the Boolean sum of the inputs. Any inputs wider than the sum being accumulated can be split into two smaller parts, each of which is added separately. While more complicated functions are certainly possible, this one will suffice for most purposes. This is particularly true if the various tag-component inputs to the hash function are statistically independent of one another. Note also that this choice also allows easy partial accumulation (it is fully associative) and easy cancellation (it is commutative and is its own inverse).

---

[10] Actually, *right quasi-associativity* might be a more appropriate name given that there is a corresponding property, *left quasi-associativity*, which is the existence of a ▮ such that

$$(\forall a,b,c)\ (a \blacksquare b) \bigstar c = a \bigstar (b \bigstar c).$$

Nevertheless, the latter property is not relevant to the current discussion and so the distinction can be ignored.

[11] Kogge and Stone [40, 41, 42] would say that ★ is *semi-associative* with respect to its *companion* operator, ▮, but the prefix *semi* does not seem appropriate in this context.

Chapter 3
Applications

I went into a house, and it wasn't a house,
It has big steps and a great big hall;
But it hasn't got a garden
A garden,
A garden,
It isn't like a house at all.

A. A. Milne [56 (p. 63)]

This chapter considers several applications for the mechanisms described in the preceding chapter. It also describes the tags required for such applications. To avoid distractions, the examination of low-level details is delayed until Chapter 4.

For the remainder of this discussion it is often easier to act as if ψ were zero. Given that ψ can be made arbitrarily close to zero by using additional tag bits, it is reasonable to take this liberty. In fact, not taking it would make the description of the problems far more awkward and would often obscure the point being made—the text would be peppered with caveats such as "with probability $1-\psi$" and "except for ψ of the time." Therefore, I make the distinction only when important to the discussion; in all other cases it should be fairly straightforward for the reader to see how things differ when ψ is non-zero but small.

3-1. Addressing checking

With a few exceptions, such as the Univac and ESS machines mentioned in Section 2-1.8, computers have not included checking of addressing and multiplexing mechanisms. In most machines there is no direct way in which one would detect an incorrect word being accessed in memory (due to a hardware failure). Even if the transmission of addresses from the processor to the memory controller is checked, addressing faults within the storage module are missed. Traditional memory coding does nothing to help—the improper word will generally have been encoded in the same manner as the proper word and so will appear valid.

Suppose each word in memory were coded using a different error-detecting code from the others. Further suppose each code were chosen to have codewords that are completely disjoint from those of each of the other codes. Finally, assume that the correct code is used for each memory access. Under such a scheme all addressing failures during memory fetches would be detected (in the absence of a memory data failure) because a fetch from the wrong location would yield a value guaranteed to not be a codeword. Addressing failures during store operations would not be immediately detected, but this is a much more difficult problem. To detect that a word other than the intended one is written would require checking every such word in storage.

This use of different, non-overlapping codes for each memory element would be expensive—it would add to each memory word at least as many bits as are needed to uniquely indicate the address of that word. It is not really necessary, however, to detect *all* possible addressing failures. Detecting *most* of them will generally suffice. Thus, by not insisting that the codes be completely disjoint and by letting a few addressing errors go undetected, one can significantly reduce the number of bits used for each memory word. Instead of storing the equivalent of the full address with each word, one can store just a few bits which indicate the result of a many-to-one function of that address. That is, one can include the address as one of the components used in generating the tag for a word. If addressing failures are completely random, one will still be able to detect all but ψ of the errors; if they have particular biases one may be able to tailor the function so as to do even better.

3-1.1. Choice of addresses for tags

There are different levels at which one can talk about the address of an object. Distinctions are often made between the *physical address space* (which is defined by the hardware), the *virtual address space* (which is defined by the memory management system), and the *name space* (which is defined by the programming language). In addition, some of these can be broken up into several parts. For instance, the name space of most programming languages allows for the name of an object plus a selector (such as a subscript) for a sub-object. Similarly, the virtual address space of several machines has both a *segment number* and a *word number.* In designing a system that will do address checking, one must decide which of these various sorts of address to include. Each seems to have both advantages and disadvantages that deserve individual examination, as follows in the next three subsections.

3-1.1.1. Physical address tagging

The primary advantage of using the physical address in the tag is that it is already available on all memory accesses—no special new mechanism is needed to provide the information. Furthermore, there is no problem of *aliasing* at this level—a given memory location has exactly one physical address.[12]

On the other hand, in many systems the physical address of an object can change rather often, most notably for paging. Each time an object is moved from one place in storage to another (or to another level in the storage hierarchy) its physical address also changes, thereby forcing one to retag each word as it is moved. Depending on the exact manner in which words are tagged, this may be difficult to do unless all other tags associated with each word are known to the mover. Because facilities such as paging are usually implemented at a level well below that for other anticipated tag components, they are unlikely to have such knowledge.

In a few cases one might be able to do this retagging automatically when moving the data without knowing the other tag components. In particular, assume that the tag bits are explicit and accessible (which might not be the case if encryption is being used) and that an appropriate incremental hashing algorithm is used. In this case, for each word moved one could "remove" the tag corresponding to the old physical address and "add" that corresponding to the new. Remember, however, that these are very special circumstances which depend upon the implementation of the tagging mechanism.

3-1.1.2. Virtual address tagging

The virtual address of an object tends to be a little more stable. Within a given virtual address space the address of objects do not normally change. Therefore one may not have the same problems with retagging objects that arise for physical address tagging. Aliasing, however, may now be a problem: two page-table entries (or other memory-mapping registers) might indicate the same block of storage. A single object might, therefore, have different addresses in different address spaces. In fact, in some systems an object might even be assigned two different locations in a single space. Although aliasing within a given virtual address space could reasonably be prohibited,[13] it would be a

---

[12]While one might conceive of a machine that intentionally ties two physical addresses to the same location, such a feature would probably offer little to recommend it. Note also that features such as relocation of *workspace-registers* in the TI 990 [88] and *prefixing* in the IBM 370 [38], which at first might seem to allow two different addresses for a given word, are actually memory mapping mechanisms; the physical addresses are their outputs rather than their inputs.

[13]Even though enforcement of this prohibition might be difficult, it is not strictly necessary. Just as has been done for undesirable aliasing in programming languages, one could specify only that the penalty for violations would be undefined results. In other words, ignore the enforcement problem.

much more serious imposition on current designs to disallow aliasing across different spaces. As very large virtual address spaces with associative or hashed (as opposed to directly indexed) memory management become more common, however, even this restriction may become less important—a given object could have a single virtual address in all spaces.

3-1.1.3. Object-name tagging

Although it is rarely represented explicitly at run-time, another space exists in which one can denote objects. The name of each object would correspond to the name used by the programmer at the time that the object is created. Except for *heap*-allocated objects, which might be considered anonymous, this would normally be the name which appears in the declaration that allocates the object. For compound objects, there might also be selectors, such as ***.B*** and ***[3]***, to distinguish the various parts of the object. To guarantee uniqueness, further qualification of these names may be necessary, using the names of any containing procedures and modules (and other scopes). Furthermore, if any of these qualifying names can have several incarnations (such as for objects local to a recursive procedure or a generic package), then the path-name must also distinguish between the various incarnations.

There are some objects that do not follow scope rules, such as those allocated from an Algol 68 *heap*. These objects are normally created explicitly (by invoking a storage-allocator from the program) rather than implicitly (by entry to the containing scope[14]) and have no names that can be directly associated with them. Therefore, special provision would have to be made to provide unique names for such objects.

By definition, the name of a particular object will never change. This means that, once again, one need not worry about retagging previously written values due to relocation. Aliasing, on the other hand, might become a serious problem, especially if one considers Fortran's *EQUIVALENCE* and *COMMON* declarations and similar facilities in other languages for overlaying variables. Certain features of some programming languages, such as the parameter passing mechanism, might also hide the "official" name of an object from a program segment that accesses it—only the name of the formal parameter might be known. In general, most current machines maintain insufficient information at run-time to reasonably support name-tagging. It is not clear how this will be affected in the future with the development of higher-level architectures.

---

[14] Actually, the term *extent* would be more appropriate here than *scope*, because the lifetime of the object is what matters, not how long it can be named. In Algol, for example, the extent of an *own* variable often exceeds its scope.

3-1.2. Instance tagging

In most systems, portions of the various address spaces are repeatedly reused, each time to represent a different object. For example, most dynamic storage allocation packages reuse those words that have been previously "freed" in preference to "new" locations. If this allows *dangling pointer* values, which point to areas of storage no longer used for the object originally designated, to be generated, then errors can result that are difficult to locate. What is needed is a way to ensure detection of any attempt to reference through an obsolete pointer value. Tagging that utilizes the programming language name space might accomplish this, but, as seen above, such factors as aliasing can still cause difficulties.

Various restrictions that prevent generation of dangling pointers have been tried in programming languages but none has proved totally satisfactory. A typical shortcoming is that otherwise valid code is prohibited because compilers would be unable to determine whether a dangling pointer can result from that code. Except in a few capability-based systems, run-time checking of pointer validity has been ignored. In principle, one could solve this problem by never reusing any address and invalidating obsolete ones. In fact, such a system could be implemented as part of the memory mapping mechanism of a computer. The expense, however, in logic complexity and pointer size might be excessive.

Tagging can provide a slight variation on this technique fairly cheaply. Instead of generating truly unique addresses, let us occasionally reuse a particular address. Furthermore, let each pointer value be split into two parts. One part is used in the traditional manner to address storage, while the other, called the *instance tag*, is the part that changes between successive allocations of the designated storage. By checking that this instance tag is still current upon each access through a pointer, attempts to use an obsolete pointer can be detected.

Some storage is reused at high rates. A stack location, for instance, might be reused as quickly as procedure or lexical-block entries and exits occur. Providing a unique instance tag value for each reallocation of stack storage would therefore imply the use of a large instance tag. It might be more practical to use only r bits. By so doing, one decreases the probability of detecting usage of a dangling pointer from certainty to $1-\psi$ (assuming random allocation of instance tag values), but this sacrifice should not be significant.

The introduction of instance tags has some interesting implications with respect to implementation. Most notably, current architectures do not make provision for determining an

appropriate instance tag for each reference to storage. When full (unabbreviated) addresses are used, as would normally be the case for *pointer* variables, the tag would just be stored along with the address.

It should be noted that tagged pointer values that appear in storage will actually have two different tags associated with them—one is the implicit tag that labels the pointer value itself while the other is the explicit tag that describes the object pointed to. The former is used to access the pointer value; the latter is used when referencing through the pointer. Perhaps this distinction is best expressed by analogy to a key contained in a locked box. The keying of the key *in* the box is quite independent of the keying of the key *to* the box. Similarly, the instance tag *in* the pointer value is distinct from the instance tag for the word *containing* the pointer value.

In a typical machine, not all references to memory include full pointers in the formation of the address. To make programs more compact, one or more methods are usually provided for expanding a "short address" in the instruction stream into a full address. Requiring a tag in each of these would substantially increase the size of the code.

Luckily, many such references involve a base register and an offset. In such cases, the tag portion of the value in the base register can act as a tag to be used when referencing the addressed storage. Therefore, this situation is fundamentally similar to any other indirection through a pointer. One problem with this assumption about based addressing is that typically only one pointer value is used to address all storage of a given type. For instance, a single such register might be used to reach all *own* storage belonging to a particular module while another might be used to reference all variables in the invocation record (*i.e.*, stack frame). In either case, all of the variables of a given class would be allocated together as a block and would receive the same instance tag. References intended to touch one but actually touching another would not be detectable unless some other tag components differed. Note, however, that the primary purpose of instance tags is to detect temporal errors such as dangling references. In this case, it does not matter that two different locations share a single tag value. Furthermore, if the variables have differing types (either "abstract" or "representation"), then that component of the final tag will suffice to detect the error. Finally, if it is deemed essential that a particular sub-block of storage be tagged uniquely, one can always allocate an extra base register for this purpose (or use an *operand modifier*, as proposed in section 3-2.3.2).

Another common form of short address selects a register. In this case, there is no obvious way to obtain an instance tag except by including it in the instruction stream. This, however, would probably cancel the principal point of register addressing, very compact addresses. A solution would

be to not tag registers at all. This could cause problems in architectures in which registers can also be addressed as memory locations because some references to the word would include tags while others would not. If, as in the PDP-10, references to storage locations that are also addressable as registers can be distinguished by the hardware, then the tag could just be ignored for those locations. In other cases, some other solution will have to be found. Note, however, that other problems, such as performance degradation, are rapidly causing the demise of such aliasing in new architectures anyway.

3-1.3. Multiplexer checking

Memory addressing actually involves two more fundamental processes, *multiplexing* and *demultiplexing*. These concepts are fundamental to almost all parts of modern computers. Other than memory elements, most of the active circuitry is devoted to selecting and routing data—only a small fraction of the hardware performs any computation other than copying bits. The term *multiplexer* will be used here to denote a device that selects a signal from a set of inputs. It propagates the selected value as the output value of the device. The choice of which input to propagate is determined by another, independent, set of inputs called the *address*. To be useful, this address obviously must be able to change with time, thereby selecting different input signals. *Demultiplexing* involves the opposite function. A single input is routed to any of several possible outputs. The value of the addressed output signal is determined by the input datum; the other (unselected) outputs take on values that do not depend on that input signal. Tagging can be used to check such multiplexing and demultiplexing.

As in the special case of fetching from random access storage, failures of multiplexers can be detected if the input values are each coded uniquely. With appropriate choice of codes (such as Hamming-like codes), partial failures, where only some of the bits are mis-selected, can also be detected. *De*multiplexing, on the other hand, is usually hard to check. To do so would require that verification not only that the appropriate signals are propagated to the addressed output but also that they are not propagated along unselected routes. Such checking would be impractical for demultiplexers with many outputs. Nevertheless, many errors can be detected (but not necessarily isolated) by employing a *read-after-write* scheme that remultiplexes the outputs of the demultiplexer so that one can verify at least that the addressed output was correctly set. Furthermore, when the addressing mechanism is shared between a multiplexer and a demultiplexer it may be useful to perform a *read-before-write* operation to check that mechanism. An example of this is the Alto file system [45], which reads the header of a disk block just before writing that block. By so doing, it is able to verify that the proper block is located under the head and that the block is a part of the file

being written. If there is a failure in the hardware addressing mechanism (in this case the disk and head positions), it can be detected *before* the wrong block is overwritten.

There is no reason that tags could not be used along the internal data paths of a machine in a manner similar to that proposed for storage. For the typical execution unit one might consider tagging each value with a number indicating from which register it is received or to which it is destined. One might also tag it with the route it is supposed to follow through the data paths. Of course, any decision whether to include such low-level tagging would be highly dependent upon performance considerations. Even the few gate-delays needed to check a tag might be excessive for some high-speed implementations.

3-2. Type tagging

The preceding sections examine various forms of address tagging. This sections examine another form of tagging: *type*-tagging. Most recent programming languages have included some notion of *typing*. That is, each object that can be manipulated by a program has a *type* which determines the attributes of that object and the operations that may be performed on it. One obvious thing to include in a tag is the *type* of the value stored in each word. If an attempt is made to access such a word without properly naming its type, then that access can be prohibited. Any legitimate accessor should know its type.[15] A correctly written program should never violate the constraints implied by type-checking. If it does, then an alarm should be raised.

3-2.1. Run-time type checking

Some systems, most notably programming languages such as PAL [15], have provided flexibility by specifying that types are associated only with values; the type of a variable can change each time it is assigned a new value. Any type-checking that is performed must be done at run-time. Nevertheless, the general tendency has been toward *strong typing* under which the type of a variable is fixed at compile-time.[16] Only objects of a specific type may be assigned to a given variable. In such a system, one can usually verify the type-correctness of a given assignment at compile-time. Why, then, is further type-checking useful? Many previous works [17, 18, 19, 34, 35, 62, 63, 64, 69]

---

[15]This might not be strictly true for some optimized implementations of *generic* procedures. The problem is that these implementations depend upon *not* performing type-checking at run-time. Remember, however, that the compilation of generic procedures is still an active research area.

[16]The term *strong typing* has been used to denote many related, but different, concepts. Compile-time typing, however, seems to be included in most people's definitions.

have cited arguments for run-time checking; several are relevant to this discussion:

- Faults in the hardware, run-time system or even the compiler itself may allow errors to be introduced *after* compile-time checking. By delaying the checks as long as possible one increases the probability of detecting errors.[17]

- Some checks are difficult or impossible to implement at compile-time. Consider, for example, assignment to *constrained* variables in Ada [33]. Whether a particular assignment is valid can depend upon the value being assigned. These same checks may be trivial if delayed until execution time.

- Forcing some of the more difficult cases to be checked at compile-time may necessitate adding restrictions to the language (such as the ALGOL 68 [94, 5, 49] rules for avoiding dangling references). Such restrictions may be stronger than really necessary, thereby prohibiting certain programs that would otherwise be safe.

- In other cases, difficulties are often resolved in the other direction—being too permissive. Allowing potentially erroneous programs to get past the checks is sometimes considered preferable to prohibiting otherwise correct programs. For instance, in Ada [30] the type of a variant record does not have to be re-checked on each access even though it might be changed by another parallel process. The burden of avoiding such a situation is left to the programmer.

It should also be noted that in some systems objects may be long-lived. In fact, an object or variable might even outlive the program that created it. In such cases, type-checking cannot be done completely at compile time; some of it must be left for run-time. In general, the longer one delays checking, the closer one can come to catching all errors without adding arbitrary restrictions.

In addition to the points made above, which argue for using run-time type-checking in addition to compile-time checking, one must consider that run-time checking may sometimes be the only form of checking available. For example, much as we might like to get rid of Fortran, its use will probably continue for some time to come. Should the users of an old language be denied access to type-checking just because it is not included in the language? If anything, they probably need it more than users of newer languages, not less!

3-2.2. User-defined types

A given object may in fact have several different types. What is seen by the low-level hardware as bits in storage may be seen by the instruction set as a bit-string, by the compiler as a set, by the programmer as a hand of playing cards, and by the programmer's boss as a waste of time. At another

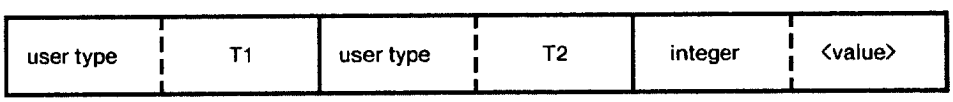
---

[17] Of course one can employ *both* compile-time and run-time checking, thereby gaining the advantages of each. The point being made is that the former, by itself, is insufficient.

moment the same bits in storage may be seen by the processor as an integer, by the low-level programmer as an index into an array, and by the next higher level programmer as a channel identifier. Recent programming languages have allowed the programmer to define new types in terms of base types "built in" to the language. Such *user-defined*, *extended*, or *abstract* types usually are not reflected at any level lower than the language system. The target machine sees only the *representation* types, which it is able to process. Compiled code manipulates all values as *integers*, *floating-point numbers*, *Boolean values*, or maybe *character strings*. Nothing, other than symbol tables, reflects the fact that the user used abstract types rather than these machine-defined types in his program.

The Burroughs B5700 and B6700 (previously called B5500 and B6500) family [69] provided only three data types: pointers and single-precision or double-precision floating point numbers.[18] BLM [34, 35] and the Rice Research Computer R-2 [17] provided several more primitive types. Myers' SWARD machine [63, 64] provided not only the basic data types commonly included in programming languages but also several special prefix tags for constructing arrays, records, etc. While these were very useful, they still only allowed one to describe the representation type of an object, not its abstract type.

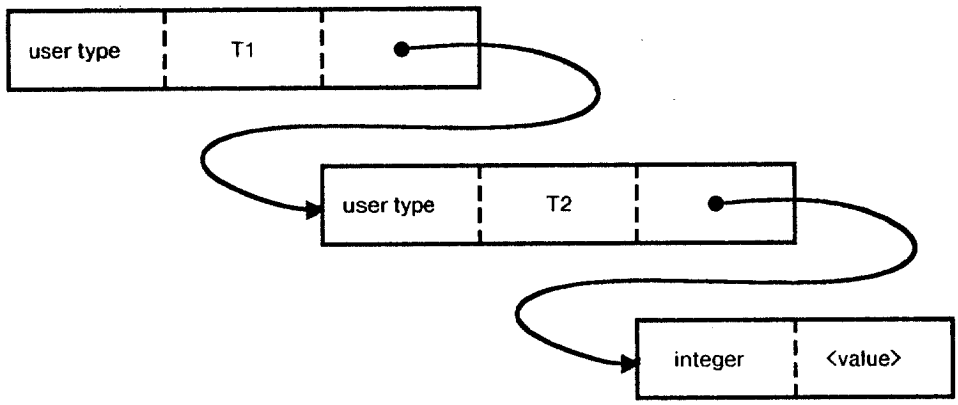
Gehringer [19] provided for objects with user-defined types. He did this using a scheme similar to that of Myers but with the addition of another special prefix for specifying an extended type. Under his scheme, an object of type *T1* which is represented as type *T2* which in turn is represented as an *integer* would appear in storage as a series of tags preceding the actual storage word. As more abstract types are used, more storage would be need for the object:



Hydra [68, 96, 97] provided (in operating system software) extended types in a slightly different manner. It imposed a fixed structure on all objects: a type-name, an array of binary data words, and an array of pointers (*capabilities*). Redell [75], Luniewski [52], and Intel [36, 91] employ a simpler model for user-defined objects. In their designs, an object of extended type is simply a type-name combined with a capability pointing to the another object which is the representation. In these cases, the above mentioned object would appear as a linked list where each link is a capability:

---

[18] The various "special control words" were part of the implementation and generally not available for manipulation by user-level programs. Even character strings were represented as arrays of floating-point numbers



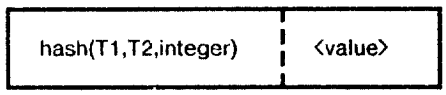
It is not clear whether any of these schemes for extended types would be practical if carried out to the lowest level. Hydra was successful, despite being object-oriented, largely because it only imposed its notion of typing, objects, and capabilities at a high level, corresponding to that used for files in more traditional operating systems. Due to the relative infrequency and large granularity of operations at this level, the added cost was not significant. Had Hydra's objects been used at the same level as objects in typical programming languages, this overhead probably would have been much more significant. In fact, even if these mechanisms were implemented completely in hardware, one might eventually find that just the overhead of all the extra memory cells for storing type-names would be intolerable.

Extended types in programming languages, on the other hand, have typically not involved extra storage at run-time. An object requires no more storage than that needed for its representation. Since the type-name need not be maintained for dynamic checking, one can also dispense with the extra level of indirection implied by the pointer. If a good optimizing compiler is employed, hiding information with an extra level of type abstraction will not involve any penalty for storing or referencing the object.

This optimization would seem to prevent one from doing run-time type checking, even as a precaution. To do so would require storing an arbitrary number of tags[19] together with each object, each being checkable by the corresponding type-managing module. Accesses performed without checking all of these tags should be trapped as errors. It is in just this situation that the compact tags proposed in this dissertation can be very useful. Because of hashing to a single fixed-size tag, only a small amount of storage is required to store an arbitrary number of abstract types for an object. Adding an extra level of abstract type just increases the number of inputs to the hash function. No

[19] or *seals*, such as those defined by Morris [61] and Redell [75]

matter how many abstract types are used, the size of a tagged object remains constant:



3-2.3. Implementation of type-tagging

The implementation of type-tagging would be quite straightforward if it were not for one complication: a given object can have more than one (user defined) type. If one had to support only the representation type, things would be quite simple. The type-tag for each access would be determined from the machine-level instructions. For instance, the operands of an *integer-add* instruction would be labelled with the type-tag corresponding to the type *integer*. Similarly, the operand of a *branch* instruction would be tagged with the type-tag for the type *instruction*. Implementation of abstract types, however, is a completely different problem. The type of a variable is dependent upon the context in which it is named. To the end-user, a variable might be of type ***T1***. To the module that defines that type, however, the variable would be represented and manipulated as type ***T2***. This renaming might be arbitrarily nested. Because such types are not defined as part of the architecture, the corresponding tag(s) cannot be derived automatically by the hardware. There must be a way to determine an appropriate tag value to use when setting and checking tags in storage. The required tag must be explicitly provided by the software. The remainder of this section examines one possible implementation of a facility for systematically so doing.

3-2.3.1. Type-tagging at subroutine entry and exit

Assume for the moment that the operations that manipulate a given abstract type are implemented as subroutines in a module defining that type. In such a case an object of that type would be accessed only from within one of those subroutines. Outside of the module the storage associated with an object would never actually be touched; at most a pointer to it could be manipulated. Therefore, a reasonable implementation might provide the required abstract type information on entry to code in the defining module. That is, for the example above, the knowledge that a particular word is a of type ***T1*** could be provided on entry to procedures in the module that manages that type. Such a scheme would resemble the model provided by CLU [50, 51] when its *cvt* feature is used: each parameter to which *cvt* applies is converted from its abstract type to its representation type on entry (and, perhaps, back on exit). I refer to the process of converting how one looks at a variable from its abstract type to its next lower representation type as *unsealing* and the reverse as *sealing*. The two terms correspond to the terms *down* and *up* that are used in CLU.

If parameters are passed by value, then at some point in the calling sequence the parameter *actuals* are probably copied to the locations associated with the parameter *formals*. If these formals are tagged only with the representation type (and not the abstract type) then the abstract type tag will be needed only during the fetch half of that one transfer. Similarly, if parameters are passed by value/result, then a corresponding transfer at subroutine exit can restore the value to a cell tagged with the abstract type's tag. Nested abstract types are handled without difficulty because at most one level of abstraction is removed at each level of subroutine nesting.

If parameters are passed by reference, then the mechanism must be a little different. Inside the type-specific subroutines there may be many references to the parameter, each of which will have to utilize appropriate type-tag information. Recall from Section 3-1.2, however, that one can put information into a pointer describing the tag of the object referenced by that pointer. While this feature was previously used to store the instance tag of the denoted object, it could also be used to store the type of that same object. If the various tag components are combined using an incremental hashing function (see Section 2-4.1), then upon entry to a type-specific subroutine the appropriate tag for the abstract type can be merged into the tag field of the pointer that denotes the parameter. Outside of the type-defining module one would indicate the address of an an object of that type with pointers containing just the instance-tag of the variable. Inside that module, however, one would address the same variable with a pointer containing both the instance-tag and the type-tag. The types of the pointer variables themselves would be *ref ⟨abstract type⟩* and *ref ⟨representation type⟩* respectively. In summary, instead of copying the value back and forth between variables of the abstract and representation types one just changes the way of looking at a single variable.

3-2.3.2. In-line type-tagging

Neither of the above mechanisms for implementing abstract type tagging need be restricted to use during subroutine invocation; they could also be supported by *unseal* and *seal* machine operations. This would allow subroutines to easily gain access to objects other than as explicit parameters. For example, objects in "global" storage or linked lists could be unsealed as needed. Although one could force the module to invoke itself to gain access to the object, such convoluted code would be awkward, unnecessary, and inefficient.

There are two ways in which one could implement these in-line operations. The most obvious manner would utilize *unseal* and *seal* instructions. Such instructions would have two input operands: a pointer (consisting of an address and a tag) and another tag. They would yield a pointer value indicating the same address as the original pointer but a tag that combines the two tag values. If tag

accumulation provides for inverses, such as would be the case for a Boolean sum, then the same instruction could be used for both purposes.

If the sealing operations are used often enough, it might be desirable to follow the analogy of replacing *add* instructions with *indexing*. Instead of using an instruction to achieve the desired effect, one could use the addressing mechanism available to each instruction to do unsealing. That is, for each data operand, an *operand modifier* could be made an optional part of the address specification. When used, it would indicate a tag component that would be merged into the tag that results from the normal addressing mechanism. Note that if a quasi-associative (see Section 2-4.1) hashing function is used for tag accumulation, then only one such modifier would be needed per operand. If more than one unsealing or sealing operation is needed for the same operand, then all the necessary tag components could be pre-merged into one operand modifier at compile-time.

Operand modifiers also allow the in-line expansion of code sequences that are too short to reasonably implement as out-of-line procedures. For instance, the assignment operator for a simple abstract type would probably be best compiled as an in-line *move* instruction rather than as a call upon a trivial type-specific subroutine. Otherwise, the procedure-call overhead would dominate any "real" work being done.

Few (or, perhaps, no) complete systems (*i.e.*, including the operating system, memory management and input/output facilities) have been implemented without bypassing type-checking at some point. Recent languages have often provided a standard "escape hatch" for this purpose: PL/I [3] has *unspec*; Euclid [44] has *<<=*; Mesa [60] has *loophole*; and Ada [30] has *unchecked_conversion*. Normally such functions do not result in the generation of any extra instructions—the distinction between various types is discarded in the run-time representation. When the proposed tagging mechanism is implemented at a low level, such escapes must also be implemented at this same level. Operand modifiers provide a simple mechanism for doing this remapping between types.

3-2.3.3. Other models of type unsealing

Although the model presented above for type-unsealing upon subroutine entry and exit is plausible, it does not seem to match that provided by many programming languages that support typing. The more common scheme seems to be to unseal objects only when operations dealing with the representation are invoked. That is, unsealing happens not on entry to the module managing the abstract type but rather on calls out from that module. It can be shown, however, that the distinction is not critical.

Consider a simple scaled arithmetic package, written in CLU [50, 51]:

```
scaled = cluster [scale_factor: int] is create,
                 negate, add, subtract, multiply, divide
         rep = int
```

The *multiply* routine could be implemented either using unsealing on entry and sealing on exit

```
multiply = proc (a: cvt, b: cvt) returns (cvt)
           c: int
           c := (a * b) / scale_factor
           return(c)
           end multiply
```

or by using unsealing at each reference to the representation type

```
multiply = proc (a: scaled, b: scaled) returns (scaled)
           c: scaled
           c := up((down(a) * down(b)) / scale_factor)
           return(c)
           end multiply
```

Note that assignment in CLU is implicitly defined for all types, including user-defined ones. If this were not the case, then the latter example would have to be rewritten to unseal *c* rather than seal the value assigned to it. Although the following is not quite a legal program, it better reflects the actual implementation:

```
multiply = proc (a: scaled, b: scaled) returns (scaled)
           c: scaled
           down(c) := (down(a) * down(b)) / scale_factor
           return(c)
           end multiply
```

This last example corresponds to the the type-unsealing model provided by many other languages, such as Ada [30]. Nevertheless, because the examples are equivalent, a compiler for a language like Ada could legally generate code that resembles the first example.

Some routines, even though part of the type-definition, might not ever reference the representation of a parameter. For example, the *subtract* operation might be implemented in terms of the *add* and *negate* operations:

```
subtract = proc (a: scaled, b: scaled) returns (scaled)
           c: scaled
           c := scaled$add(a, scaled$negate(b))
           return(c)
           end multiply
```

In this case, a compiler would not unseal the parameter at all.

A few routines might reference a parameter both as the abstract type and as the representation type. A number of options exist for the compiler in this case:

- The procedure could explicitly unseal its parameters when needed.
- If the majority of references within the procedure are to the representation, then one could still unseal at entry but explicitly reseal when needed.

- The parameter-passing mechanism could provide both the sealed (abstract) and unsealed (representation) versions of the appropriate references.

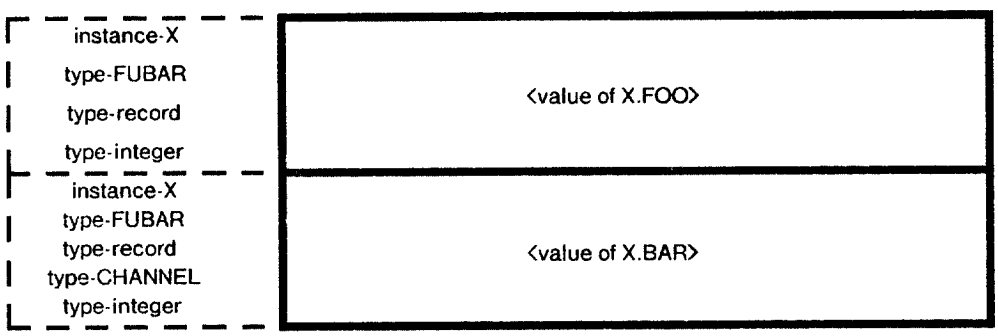
Note that a compiler could choose to use any of these approaches for each parameter of each procedure; it would not be constrained to choosing a single method for all code that it generates.

3-2.3.4. An example

By now the reader may be thoroughly confused about the values of various tag fields. Consider, therefore, a simple example based upon the following declarations written in Pidgin 81 [7]:

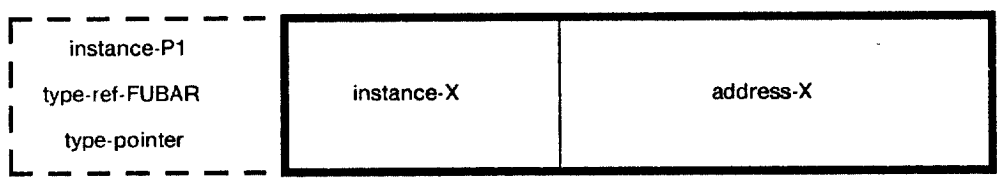
```
type CHANNEL = integer;
type FUBAR = record(FOO: integer, BAR: CHANNEL);
variable X: FUBAR;
```

If just type-tagging and instance-tagging are used, then one might picture the storage allocated for *X* as follows:



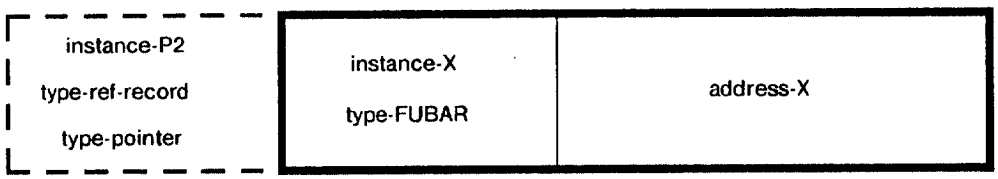
The portions bounded by solid lines represent the values in the storage allocated for *X* and the portions bounded by broken lines represent the tags attached to those words. The tag components shown would be combined by the hashing algorithm into a single composite tag.

A pointer, ***P1***, that refers to *X* might be represented in a similar manner:



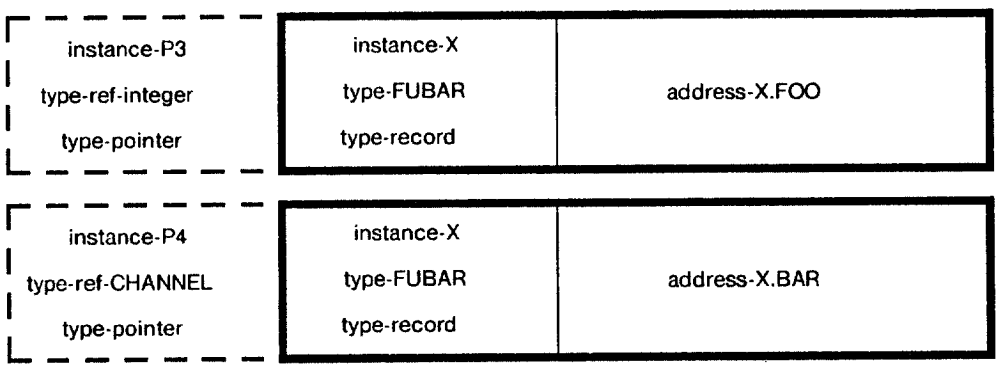
The new field, which is part of the value of the pointer, provides the instance tag information that was generated upon allocation of storage for *X*. Note that the tag of ***P1*** itself is based upon the variable's abstract type being *ref FUBAR* and its representation type being *pointer*. This is not strictly necessary; ***P1*** could also be declared to be a primitive pointer. This would eliminate the need to unseal it when used but would also eliminate some of the double tag-checking referred to in Section 2-4.

If one passed *X* to the module that manipulates objects of type *FUBAR*, it might unseal the reference to the parameter, yielding another pointer value in ***P2***:



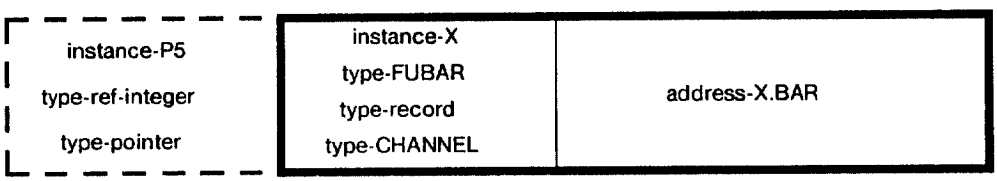
The unsealing operation just adds the type-tag corresponding to *FUBAR* to the pointer value.

Selecting each of the components of this parameter and putting their addresses into pointers ***P3*** and ***P4*** would in turn unseal the *record* type:



Including *record* as an explicitly tagged, run-time, type is arbitrary (on my part), based on the notion that it is a special sort of generic type. It could be omitted without problem, in which case ***P2*** would be a primitive pointer instead of a typed pointer. Alternatively, the unsealing of ***X*** and the selection of its components could be combined into a single operation, thus yielding ***P3*** and ***P4*** without the intermediate value shown in ***P2***.

The final unsealing operation, performed in the module that manipulates *channels*, would yield ***P5***:



Below this level the hardware would implicitly unseal the type *integer* when performing instructions that operate on integers.

3-2.4. Type comparison without type inquiry

The most significant difference between the architecture proposed in this dissertation and previous tagged designs is that inquiries about the type of a word are prohibited. Such information must be provided as an *input* to each memory access rather than be produced as a *result*. This is because the tagging mechanism is designed only to allow comparison of tags, not the fetching thereof. Furthermore, because it sometimes (with probability ψ) gives the "incorrect" answer, it is possible that two logically different tags may be reported as matching. When an incorrect answer is given, however, it will always be a "safe" one—no correct program will encounter spurious error traps. The constraint used is that no properly matching tag will be rejected as erroneous (in the absence of memory failures) but that *most* non-matching ones will be discovered. This allowance of occasional improper acceptance is critical to the compact storage of tags, as explained in Section 2-4.

Some tagged designs reduce the number of bits in the instruction stream by making many of the machine-code level operations *generic*. For instance, an *add* operation might be capable of doing integer, fixed-point, or floating-point addition, with the appropriate selection made on the basis of the type of each of the operands. Such a system is often referred to as having *overloaded* its operators. Since my proposed scheme does not allow one to inquire about the type of a word, it clearly cannot directly support such run-time overloading of the machine instructions. Just as for untagged machines, addition of such generic operations would require addition of explicit tags to the data. Thus one does not get one of the benefits of a typical tagged architecture. In fact, because every memory operation must present the tag of the word being accessed, one may even require *extra* bits in the instruction stream! While some traditional machines might use the same *clear* instruction for single-word integers, floating-point numbers or even bit-strings, those cases must somehow be distinguished in the proposed architecture. Of course any extra bits required for this purpose are not wasted: errors may be caught that otherwise would have been missed. Indeed, this example supports one of the main contentions of this dissertation, that the redundancy present in most instruction streams can be used to detect errors.

3-3. Ownership tagging

Both instance tagging and the abstract-type tagging proposed above depend upon the inclusion of a tag part in pointer values. Because in some situations the attendant expense might be excessive, it is worthwhile to at least consider other schemes which would avoid this requirement. One such tagging system does provide some of the benefits of instance and abstract-type tagging, although the scope of detection is narrower.

Two tag components could be provided to the executing process without major overhead: one component corresponds to the primitive type of the object being referenced and another corresponds to the module from which the access takes place (the *owner*). For objects that have an abstract type, the owner would be the module that implements the lowest level programmed abstraction; for objects implemented directly as a primitive type, it would be the module in which the object is declared.

This definition prohibits any sharing of data between modules. Therefore, for those cases where sharing is needed, one would have to define an intermediate module and declare the shared objects to belong to it. Each time one of the modules wished to access a shared object, it would call a procedure in the intermediate module. Furthermore, parameter passing during procedure invocation would be specially implemented to sit on the fence between the two modules, able to read from the actuals using the caller's tag and then write to the formals using the the callee's tag. Upon procedure return, the inverse would take place for any returned values. They would be read using the callee's tag and written using the caller's tag. In both of these cases, a mechanism for optionally specifying a tag for each operand of an instruction should prove useful. The operand modifiers proposed in Section 3-2.3.2 will work for this purpose. Again, it would allow in-line expansion of procedures that are logically part of another module and thereby avoid subroutine-invocation overhead for trivial operations.

The primary disadvantage of ownership tagging is that only the tags associated with the primitive representation type and the owner are stored with each word. No checking of higher-level abstractions can be included. Furthermore, there is no way to distinguish an object of an abstract type from any other object of that same type. Many more errors, such as those resulting from dangling pointers, might slip by undetected.

The only ways to regain a little control seem to require extra storage. For instance, abstract types might be implemented as pointers to the representation (as opposed to renamings of the representation). Note, however, that the extra memory access required to "walk the path" to an object might cause significant performance degradation. Any attempt to reduce these storage requirements or memory accesses will have the negative side-effect of also reducing the probability of detecting an erroneous situation.

3-4. Other applications

The tagging proposed so far in this chapter (other, perhaps, than ownership tagging) can also provide some benefits other than those already described. The net effect of the mechanisms is to allow separation of storage words into equivalence classes. Any error-check which depends upon dynamically verifying assertions as to class-membership can be implemented using tags. Most of the errors so detected can be described as accessing the wrong word, accessing the right word but in the wrong manner, or accessing it at the wrong time. There are a number of commonly encountered error-checks that could be included in this group; the rest of this chapter examines several of them.

3-4.1. Bounds checking

If each variable is uniquely tagged then attempts to access an array element using an index that exceeds the bounds of the array will be automatically detected. This is because the tag of words outside the array will differ from that of words inside the array. Unlike traditional subscript range checking this does not require time-consuming comparisons with each of the limits. Instead it is included in the one tag comparison.

Of course there is a limitation: tagging will only check against exceeding the extent of the entire array rather than testing each dimension individually. For instance, a 2×2 array called A, stored contiguously in row-major order, might be accessed using ***A[1,3]*** which would actually refer to ***A[2,1]***. Because this is still part of the array A, it would be tagged with the same instance tag and so the error would go undetected. There are two ways around this problem:

- If, as mentioned in Section 3-1.1.3, subscripts were included in the tag computation for each element of the array then such checks become possible. This, however, requires that subscripts be handled as a special case—simple arithmetic operations for combining subscripts would no longer suffice.

- Alternatively, one could represent multi-dimensional arrays using Iliffe vectors, which are vectors of pointers to vectors. By distributing the the tags so that one dimension is checked at each level of indirection and one at the final fetch, one can avoid special handling of subscripts. On the other hand, one still may have troubles with certain forms of array *slicing*, as described in Section 4-4.

Although subscript-checking using tagging is not well suited to checking individual subscripts of a multidimensional array, it is quite appropriate for use with oddly shaped arrays. Consider, for instance, a vector that is indexed by a subrange not of the integers but rather of the prime numbers. Such sparseness would normally make verification of subscripts rather expensive—limit checks are not sufficient. On the other hand, tagging of the data cells works wonderfully. No matter how widely

the elements of the vector might be scattered in storage, only they will be tagged appropriately to allow access. Even if other variables are allocated to the "gaps" between them, one can still detect bad subscript values because those gaps would be tagged differently from legitimate cells.

3-4.2. Random jump detection

If instance tags are used for all segments of storage, including the instruction stream, then even errors in the flow of control will be detectable. A particularly nasty error to isolate is the "random jump" in which a processor apparently inexplicably reaches a particular execution point. Possible causes include a hardware failure in the program counter, an uninitialized label variable, an out-of-bounds *case* index, etc. With instance tagging, almost all of these will be detectable because each code segment can be tagged differently. The most notable exception is a branch that ends up in the "right" segment but the wrong location within it. Fortunately, this gap can be minimized by fragmenting the code into many different segments, perhaps even as small as the *basic blocks* referred to in the literature on compilers [23, 2]. Transfers between these segments would, of course, specify the new instance tag.

Except when the value of the program counter is destroyed by a hardware error, it is also possible to determine the location of the errant jump. If, as in the Honeywell processors used to support Multics [29 (pp. 8–9)], the instruction at the destination of the transfer is fetched as an operand *before* changing the program counter, then the tag-mismatch can be caught while still at the offending instruction.[20] Treating a transfer-of-control as a memory-referencing instruction need not negatively affect performance—the target instruction must be fetched anyway prior to its execution.

3-4.3. Uninitialized variables

Section 3-1.2 discusses the use of tags for detecting references to variables whose storage has been freed and then reallocated. It should be obvious that the same mechanisms will also detect references to uninitialized storage locations. This depends on the fact that such locations will be tagged according to their previous usage. Not only might the type of the old value differ from that of the new usage, but also the ownership would probably vary. Even if both of these happen to match,

---

[20]Similar effect can sometimes be achieved in machines that record in another register the previous value of the program counter. In the IBM 1401 with indexing installed [37], for instance, every branching instruction copied the *I-register* to the *B-register*. Programmers used this feature, along with one-character *halt* instructions, to track down the source of erroneous jumps. The paging box used with KA-10 versions of ITS [28] provided a special register, designed especially to aid debugging, that was set by each user-mode transfer instruction. Some later machines have added special *history* registers to keep track of similar information. Nevertheless, checking before changing the program counter seems to be a simpler yet effective method.

the instance tag ought to differ. Thus, except for the always-present miss-rate ψ, one can expect that fetches of uninitialized will be detected by tagging.

3-4.4. Variant records

There is a general problem with *variant records*, *union* types, etc. that some accesses may be legal only when an appropriate value is contained in the object. Normally, a *discriminant* is included in the representation to allow one to make the appropriate checks. Many languages will not let a program access any part of the representation of an object that depends on this discriminant without first explicitly checking its value (as with a *discrimination* in Mesa [60]). Although designed to restore type-safety to a situation where the normal rules of strong typing have been relaxed, such dispatches do leave loopholes. Most notably, shared variables and aliasing can often invalidate the result of such tests. For example, the type discriminant might change between the time that it is checked and the time that the dependent field is accessed. The typical response by a language designer is to specify that such behavior is illegal and will yield undefined results. Furthermore, automatic detection of an erroneous situation is not required—*caveat magister!*[21]

When every cell is tagged, appropriate checking is done automatically. This is particularly true if one uses name-space tagging (or any other scheme that includes the discriminant's value in the tag). At last one can enforce *constrained* objects in Ada at reasonable cost.

3-4.5. Extended tagging

There are other applications in which the basic tagging mechanisms proposed in Chapter 2 might be useful. Furthermore, it is usually good design practice to provide the "hooks" necessary for others to implement such extensions. In the case of tagging there is little problem because the inclusion of tags in pointers (see Section 3-1.2) and the operand modification mechanism (see Section 3-2.3.2) should suffice. In no way are new applications of tagging precluded by those already chosen. While detailed examples of such extended tagging are inappropriate here, two possibilities are at least worth mentioning:

- Access to some objects could be restricted to certain phases of program execution. Temporarily changing the "key" used to "unlock" those objects would make then unreachable.

- Multi-word objects whose components are supposed to "track" each other could be

[21] The Romans, of course, had no Latin word for *programmer*. Traupman [90] defines *magister* as a "chief, master, director; teacher; adviser; . . . author; captain; pilot; . . ." which seems a suitable approximation.

checked for consistency. If each word is tagged with a version number, then obsolete values would be caught because they would be tagged with the wrong version number.

The important point to be noticed is that, due to the arbitrary fan-in of an incremental hash function (see Section 2-4.1), these facilities are not usurped by the built-in applications and so are available for other uses as well.

3-5. Summary of the applications

In the preceding few sections are listed a number of possible applications for tagging. Reviewing, one finds the following tag components mentioned:

- physical address tags
- virtual address tags
- object-name tags
- instance tags
- representation type tags
- abstract type tags
- ownership tags

It is unlikely that any one implementation would choose to use all of them. Instead, one would expect to choose from among the items on this list those that would be most practical for the circumstances at hand and would tend to catch many errors at low cost. It would be impossible to make blanket recommendations here as to which would be "best" without knowing more about the application. Nevertheless, Chapter 5 gives one scenario that was designed to be simple yet effective.

It should be noted in passing that some of the tags mentioned above are "visible" at different levels than the others. For example, there is no reason that even the assembly-language programmer would have to know what sort of (or even whether) physical address tags are used. In fact, this might vary widely between different implementations of the same architecture. Similarly, although knowledge of representation type tagging would be needed by an assembly-language programmer or a compiler, it could be hidden from higher levels. Recognizing the level at which a given tag becomes invisible might allow the designer some flexibility in his decisions. With some care, later implementations could be changed without requiring major retrofits to previous software.

Chapter 4
Implementation details

> . . . another failure was when I tried to run my 6 volt electric motor by connecting it up to the electric light switch. Nothing happened. Even now, looking back on the event with greater electrical knowledge, I still can't understand why *nothing* happened.
>
> Christopher Milne [59 (p. 40)]

As mentioned at the beginning of Chapter 3, there are a number of implementation details that could get in the way of implementing my proposals. None of them, however, seems to be insurmountable. Some of the apparent problems vanish once one fully appreciates the implications of run-time tagging. In particular, by implementing low-level restrictions that previously were higher-level, one will clearly affect those parts of a system that intentionally circumvented such restrictions. For example, debugging-aids would have to be given information (*e.g.*, in a symbol-table) about the abstract types of the objects that it can examine.

Of course, not all problems disappear just by changing one's thinking. Therefore this chapter examines some of the problems and proposes possible solutions. It is quite likely that there are other (and perhaps better) ways to handle many of these issues, but in configuration-specific factors may well dominate in determining one's choice between them. For the purpose of this dissertation, however, demonstrating solvability will suffice.

4-1. Tag checking when storing

An important factor in error detection is detecting (and acting upon) errors as soon as possible. It is best to detect the error before it is propagated, while its effects can still be contained. Error detection schemes generally do not detect the actual failure; instead they detect inconsistencies that result from a failure. Therefore, it is important for one to do this consistency checking as often as possible. Only by constantly looking for such symptoms can one hope to detect errors before their effects become too widespread.

One common (and puzzling) circumstance encountered by programmers is finding a "garbage" value in a variable. Often this is the result of an assignment statement that has stored into a location other than the one intended. The cause of this improper store operation is usually a program bug but occasionally is a hardware problem. Unfortunately, debugging such problems with current machines involves difficult backtracking to determine at what point the bogus value was stored. Neither traditional error checking nor the tagging methods proposed so far will detect such mistakes until the erroneously overwritten location is fetched. It would be better to detect the error at the time of the erroneous store operation, *before* it is performed.

Suppose, for instance, that all stores to memory are preceded by fetches using the same address. Only if the fetch succeeds will the store be allowed. By itself this gains almost nothing but when combined with the proposed tagging mechanism it can become quite effective. If an *integer* is to be stored in a variable, then one should be able to fetch (and discard) an *integer* from that variable first. If this is not possible, then the operation will be aborted before spreading the damage. Such a check will be able to detect *in advance* many software and hardware failures that might otherwise result in randomly overwriting data. The most notable exception would be hardware addressing failures that did not affect the preceding fetch. For many storage technologies, however, such problems tend to be rare because the addressing mechanism can be shared between the fetch and the store phases.

One might, for even more reliability, consider combining the above-mentioned read-before-write sequence with a read-after-write sequence. It is not possible to evaluate the benefit versus cost of such a read-write-read sequence, however, without considering the specific storage technology being used. In some cases, the final fetch might even decrease overall reliability! For example, if a *destructive read* is involved, then the last read (and its implicit rewrite) might introduce more errors than it catches.

A note on efficiency: many memory devices can be read and written nearly as quickly as they can be just written. The cycle time of many MOS chips, for instance, is dominated by the time needed for address-decoding and line-precharging. Therefore, with careful implementation, one might add this tag-checking without much incremental expense. See also the example of the Alto file system in Section 3-1.3 in which a header is read before writing the corresponding sector.

Of course, if one reads before writing then there must be some provision for initializing variables after they are allocated. Otherwise, the first attempt to store into a word would fail. Initialization could be done explicitly by a special operation that stores without first fetching, but this creates another problem. This special initializing-write operation might be used on the wrong memory cell!

A better choice would be to define the special initialization operation such that it only allows writing into cells that are marked *empty*.[22] In effect one can shift one's thinking from doing initialization to doing finalization. Instead of explicitly initializing a cell when one allocates it, one must be sure to explicitly mark it empty when freeing it. The biggest advantage of this scheme is that it eliminates the window of vulnerability that would be introduced by an initialization operation. *Every* operation that changes storage would be checked by a preceding fetch.

It turns out that this paradigm may have another advantage: improved performance. Finalization, unlike initialization, does not have to be done on demand. Instead it can be done by an asynchronous process (such as a garbage collector[23]) that returns no-longer-used memory to a free storage pool. This potential parallelism might be exploited to speed up program execution.

4-2. Allocated but uninitialized *vs.* initialized storage

One problem with eliminating explicit initialization upon allocation is that it leaves no way to distinguish storage that has been allocated but not yet initialized from storage that is in the free storage pool: in both cases the storage will be marked empty. Any attempt to solve this would require rewriting storage whenever it is allocated, which is equivalent to initialization. While the use of initialization and finalization in combination does not introduce any conceptual problems, bringing back the former also brings back the performance degradation I was so glad to eliminate above. Nevertheless, the problem is not at all serious. If, for a particular situation, the cost of initialization is not excessive then using it will allow the detection of a few more errors—attempts to write into unallocated storage. On the other hand, if no distinction is made and initiation is not performed, then one would still not allow data to be accidentally overwritten. That is, no information will be lost because one would be taking this freedom only with cells marked empty anyway. The worst consequence might be that when the storage is eventually allocated it will not be accessible due to being non-empty.[24]

---

[22] Alternatively, one could eliminate the initialization operation in favor of modifying the normal store operation to allow writes either when the cell being written contains a proper value or when it is marked empty.

[23] See Baker's description [4] of a practical implementation of parallel garbage collection.

[24] If the error that caused the errant store is transient, then it may also be impossible to later fetch the newly computed data, but that is another problem.

4-3. Tagging of variant records

Many languages allow some flexibility in the run-time representation of objects. That is, they allow the representation of a particular instance of an object to vary in a manner depending on the value of the object. An example of such would be an object used to store bibliographic information in a text processing program. The record for an article might consist of an *author*, *title*, *journal*, *volume*, *pages*, and *date*. A doctoral dissertation, on the other hand, would be represented as an *author*, *title*, *department*, *school*, and *date*. In both cases, the representation would also include a *document-type* field indicating which variant is being represented. Such objects are usually referred to as *variant records*; the common type indication is referred to as the *type discriminant*.

Variant records present two problems. The first is that the type discriminant often can take on one of only a small number of values. Therefore, it might be stored in only a few bits. This means that it may well take up less than a machine word and so some other part of the structure may share the same word. If tagging is done on a per-word basis, there may be no way to fetch the discriminant without knowing the format of the rest of the word. Such knowledge may in turn be available only by examining the type discriminant. To avoid this circularity, compilers for this machine must be careful about how they pack data. While this may not be very difficult to do, it is, nevertheless, a new restriction imposed by the tagging mechanism. More will be said below, in Section 4-5, about packing more than one item into a tagged cell.

The other problem that arises in connection with variant records is that the tagging mechanism may introduce new inefficiencies in the manipulation of such records. In the code generated by a typical compiler, assignments of entire records are performed without examining the type discriminant; instead the whole record is just copied bit-for-bit. Assuming all possible variants of the record are represented using a block of storage large enough for the largest thereof, the assignment need not be varied in a data-dependent manner. On the other hand, if the representation is tagged differently for each possible format, then the processor will have to employ a different tag for each format. This may mean that such assignments will have to be split into code that dispatches on the discriminant, performing similar but slightly different operations for each possible representation.[25] Again this is not an insurmountable problem, but it is a change from traditional designs. Its impact on performance will have to be measured.

---

[25] If a read-before-write paradigm, as described in Section 4-1, is used and the assignment changes the discriminant, then one might have to dispatch according to *both* the old and the new discriminant values.

4-4. Slicing

Some languages, such as ALGOL 68 and (to a certain extent) PL/I and Ada, provide slicing facilities. These allow the user to treat a part of a data structure as if it were itself a top level structure. For instance, the two dimensional array *A[*,*]* might be addressed as a vector slice *A[3,*]*, thereby indicating one row-vector of the containing array. Alternatively, one might reference the rectangular slice *A[3:5,*]*, thereby indicating a two dimensional sub-array. Incidentally, a very restricted slicing facility is available in almost all languages: computing the address of a single element of an array, such as *A[3,7]*.

A more common sort of slicing is structure slicing. A structure *X* which has components *Y* and *Z* (each of which in turn might have subcomponents) can be sliced by referring to *X.Y* or *X.Z*. Finally, if *X[*]* is a vector of structured records, one might even be able to refer to *X[*].Z* or *X.Z[*]* as a vector of just its *Z* components. The common property of all these forms of slicing is that some of the parameters normally used when accessing elements of the structured object are bound in advance.

While in most cases slicing will add little complication, there is one case that can get a bit tricky: the creation of *pointer* values that denote a particular slice. This usually involves collapsing the information presented into a more concise form. For instance, a vector might be represented as an *address*, *delta*, and *bounds*, independent of whence the vector was sliced. References to the vector might check the actual subscript used against the bounds. If acceptable, then the address computation would evaluate the expression *address + delta*(subscript − lowerbound)*. This slicing process discards information; it is assumed that there will be no need to back up and retrieve the original array of which the vector is a slice. Nor can one determine the selector(s) used to specify the slice. Once a vector has been isolated from the containing array, this information is just not available unless retained separately.

Unfortunately, if one has chosen to use object-name tags, as described in Section 3-1.1.3, then the processor must be able to compute a unique name for each location, independent of how it is addressed. The obvious way to do so would be to retain with the pointer all the information presented to the slicing operation, such as the original array name and selectors.[26] Alternatively, if the sorts of slicing available to the programmer are appropriately restricted, it would be possible to utilize incremental hashing in a manner analogous to that used to implement type-tagging. Obviously the details of so doing would be specific to a particular implementation.

---

[26] It should be noted that, in simple implementations, some of this information may have to be retained anyway for use by the garbage collector. Otherwise, the original array might appear unreferenced even though there would still be outstanding references to slices thereof.

Many implementations of APL, starting with one proposed by Abrams [1], employ a technique know as *beating*. That is, for certain operations, such as matrix-transposition, no data is moved. Instead, only the items used for addressing the data are changed. For example, a vector could be reversed by replacing the *address* of the vector with *address+delta*(upperbound-lowerbound)* and negating the value of the *delta*. Given that the results of cascaded operations can become rather complex, it is not at all clear that object-name tags would be able to resolve any more detail than the original array. In other words, it is unlikely that one could gain much more error-detection capability than is already provided by instance tags.

4-5. Packed data structures

There is no easy rule for determining the appropriate size for the minimal tagged cell. Tagging each bit of storage with several bits of tag would be inefficient. Providing only one tag for large blocks of storage, on the other hand, would probably be overly restrictive on how one allocates storage for distinct purposes. A reasonable compromise might be something about the size of a typical machine word, between 25 and 100 bits. It is likely, however, that some applications would pack more than one field into such words. This raises a question as to which type-tags should be associated with a packed word. One could define the packed structure itself to be an abstract type and tag the word appropriately. While doing so would certainly be appropriate, it still only designates an abstract type and does not resolve which representation types should be included.

If all of the fields have the same type, as might be the case for character strings, then one could tag the packed word just as one would tag the corresponding untagged word except for adding an extra component which would reflect the packing. When one wishes to access just one of the fields in isolation, the selection operation would have to unseal this top-level structure, leaving a reference to an object of the base type. Although this does not resolve the issue of how to denote which bits of the word are to be accessed, such indication is orthogonal to the issue of type-tagging and would have to be considered anyway. Thus, a pointer to a packed character-string might contain just the instance-tag of the string. A pointer to the first character of that string would indicate the same address but its tag filed would consist of both the instance-tag and the tag corresponding to the *string* type. Each word used to store string (as opposed to the pointers to the string) would be tagged with the instance-tag, the tag corresponding to *string*, and the tag corresponding to *character*.

No similar approach exists for heterogeneous fields packed into a single word. An inelegant solution would be to include type information for all fields of the word in the tag for that word. The problem with this, however, is that in order to actually read or write any particular field one must

provide all these tag values. Unfortunately, the normal type-unsealing mechanism provides only the the type information corresponding to the field being referenced, not the other fields. Furthermore, just knowing the abstract types of all the other fields in the word is insufficient; one must also know all the underlying representation types as well. For a language such as Ada this can be handled—the language definition forces this information to be available to the compiler. Therefore a compiler could generate an appropriate operand modifier to offset the extraneous tag information. On the other hand, in many systems each level of abstraction needs to know only the immediately underlying representation type. The tagging proposed thus far is quite capable of working with such systems except for this one sticky problem. Without the rest of the information, it seems that packed words would have to be prohibited.

Unless storage allocators are allowed to return partial words, thereby artificially creating packed data structures, instance tags do not have any negative interaction with packing. This is because the instance tag is the same for all items in a given object.

4-6. Multi-cell values

Complementary to packing many different values into a single tagged cell is spreading a single value across multiple cells. In the most straightforward implementation, each of the cells would be tagged identically (other than for address-tagging). When fetching a *long integer*, for instance, the processor would assert similar software-derived tag-values for each of the parts of the variable. An obvious consideration is that to achieve reasonable performance one might have to replicate the tag-checking circuitry, thereby allowing all the cells to be processed in parallel.

4-7. Debugging tools

Traditional debugging tools, often designed for use with assembly-language programs, will probably be tripped up by tag checking. Even many of the so-called source language debuggers may have problems. This is primarily because it will no longer be possible to examine and/or set arbitrary locations in storage without first obtaining all the relevant tag components for the cells to be accessed. This is not really a deficiency of tagging but rather a deficiency in the tools. If sufficient information is available to a compiler for generating code to access certain data, then this same information ought to allow debugging tools to be able to do likewise. An appropriate communication mechanism for transmitting this information from the compiler to the run-time environment is all that must be added. For example, the symbol tables now used to map between names and addresses could be expanded to include appropriate tagging information, such as *type*. Note, however, that much of this

expansion would be necessary anyway if one wished to make the tools truly oriented toward working at the source level.

4-8. I/O and other "blind" accesses

There are two distinct sorts of operations that are often classified as *input/output.* The first group includes those that are explicitly programmed at a high level. Because such transactions do not differ from normal program execution with respect to tagging, all necessary type information is available. For example, when appending an integer value to a file it would be known that the value is an integer.

The other group includes "blind" accesses to storage Prototypical of these is the movement of pages between different storage media. Another example is a *checkpointing* mechanism which saves or restores the entire state of a computation. In either of these situations, words of storage must be read or written without knowledge of how those words are used. Adding tags to storage might make implementation difficult because the appropriate tag values would not be available for the storage accesses performed. That is, the processes moving the data would have to do so without knowing which tags to use. Such "blind" operations seem to conflict with the principle that all storage references include a tag.

On the other hand, if one does not know the tag of a particular word then it is unlikely that one would attempt to make any sense of its value. That is, only programs that understand the type of objects being manipulated having any business trying to interpret them. It does not matter whether other programs have sufficient tag information available to perform a normal read or write operation—instead they only need be able to move the data in such a way that they can still be read by processes that *do* have that information. Therefore, the simplest solution would appear to be to just bypass the tagging mechanism. For instance, one might copy entire words as stored, complete with tag bits, from one location to another. This would work even when encryption is used—decryption is necessary only for processes that interpret the contents of the words.

There are a few problems with such a scheme. If address tagging is used then migration of words between locations could cause problems if they are not returned to their original locations before being accessed in the normal manner. Such a situation might arise, for instance, in a system that combined virtual memory management with physical-address tagging. When data are moved from one physical location to another, they would have to be appropriately retagged. If the incremental hashing algorithm is appropriately chosen then this would indeed be possible. For instance, a sum

can be corrected by adding the difference between the new and old address-tag values. On the other hand, if the tag is used as a key for encryption, then such incremental compensation might well be impossible. In such situations it would be better not to employ physical address tags. With sufficient randomness in the other tags, accessing the wrong word of memory ought to be detectable anyway.

For some storage devices it might appear that some space would be wasted. For instance, in most present machines the check bits for parity or Hamming codes are discarded when data are moved from primary to secondary storage. In their place check bits and codes more oriented toward errors in large blocks, particularly burst errors, are used. If one has combined tag information with the primary storage's check bits then the tag information must not be discarded. Note, however, that this is not as expensive as it might at first seem. One would have to make provision for retaining the tag bits whether they were combined with the check bits or not. Furthermore, by retaining these check bits one might just detect an error that slips by (or is miscorrected by) the secondary storage's code.

The last problem I will bring up is perhaps the most serious. If one suppresses tag checking during storage migration then one also gives up the error detection that the checking was supposed to provide. In particular, an incorrect word of storage might be read or written. The best solution I have found seems to be to add a second level of tagging, this time at the block or page level. That is, each page of storage would have associated with it a tag that would be checked when doing blind accesses. At this level, the tags for each word would remain uninterpreted, just as if they were part of the data stored. Verification of this per-page tag would act as a consistency check that the proper page was indeed being accessed. Although some of the fine grain detection might be lost, it still ought to detect a large number of the operations that would erroneously access the wrong page.

4-9. Garbage collection

Garbage collection can add a few complications to the tagging methods proposed. Not only must the garbage collector be able to access active cells during its scanning phase but also it must be able to release the unreached storage for reuse. The former capability ought not be difficult to provide—the scanning process ought to know the structure and type (and so the tags) of any words it reads! The latter capability, however, is a bit more difficult; it is even possible that some of the relevant information was discarded simultaneously with the last active reference to the recyclable words. If one cannot access them, there would seem to be no simple way to overwrite them with the *empty* value mentioned in Section 4-1.

A copying garbage collector, such as those proposed by Bishop [6] and Baker [4], can avoid this problem. By making a copy of all the active words one can avoid having to touch the discarded ones. Instead one discards the containing page *en masse*, using a migration-like operation as proposed in Section 4-8. Knowing the tag for the entire page, rather than the tags for the individual words, is therefore sufficient.

4-10. Stacks

One commonly used data structure is the stack. Homogeneous stacks, which can hold only values of a single type, are really no different than any other compound data structure with respect to tagging. On the other hand, heterogeneous stacks, such as the typical main program stack, are probably better considered to be a form of storage allocation mechanism rather than data structure. Not only can the type of a given cell in the stack change but also one would probably want to detect dangling pointers which reference cells that have been deallocated (*i.e.*, popped off the stack) and then reallocated.

One of the key attributes of traditional stacks is the ability both to allocate and to free storage efficiently. The overhead normally associated with these operations is no more than incrementing or decrementing a register. Furthermore, unlike many other storage allocation methods, stacks must often reuse locations in memory almost immediately after freeing them. This means that even if one assumes that storage is erased before reuse by a parallel process (see Section 4-1), such erasure will still be in the critical path of the main computation. In many situations, the extra overhead of performing erasure for every reallocation of a stack location would not be tolerable.

Obviously one could switch to some other method for storage allocation, one that does not reuse storage as quickly. Such a move, however, is probably too radical to gain easy acceptance and would still have performance problems. If nothing else, it would require that more storage be available for allocation, thereby implying less efficient storage utilization. It is important to find a way for tagging and stacks to coexist without the former hobbling the latter.

As for many of the other implementation details, design of an improved mechanism for stacks is dependent upon the particular machine configuration, most notably stack representation. Nevertheless, I will show one implementation which could be adapted to a wide range of situations. To simplify things, I will limit discussion to the implementation of a traditional program stack.

One or two registers are used to address this stack: a top-of-stack pointer, ***SP***, and perhaps

(depending upon the rest of the architecture) a stack-frame pointer, ***FP***. The former indicates the boundary between allocated and free storage; it can be used for dynamic allocation of local storage. The latter register indicates the base of the current *stack frame*. Upon frame allocation, which normally happens as part of procedure invocation,

- a pseudo-unique instance tag is generated for the new stack frame (the exact method of generation is unimportant to this discussion);
- the old values of ***SP*** and ***FP*** are saved;
- a new value for ***FP*** is determined from ***SP***;
- the tag portions of ***SP*** and ***FP*** are set to the new instance tag.

Freeing a frame is much simpler: ***SP*** and ***FP*** are just restored to contain their old values.

All operations that modify ***SP***, including those not traditionally checked (*e.g.*, direct assignments), must be verified against the overflow and underflow boundaries of the stack.[27] This checking guarantees that the stack pointer will always point to cells reserved for use in the stack. It can therefore be assumed that any words allocated using ***SP*** are free even if they have not been explicitly erased to the empty value. This, in turn, implies that one no longer needs to erase cells when they are popped off the stack—old values can be left in storage until the locations are reused. Stack-pushing operations, unlike most other writing operations, would then not have to perform the read-before-write proposed in Section 4-1.

One slight imposition remains upon the implementation of stacks: it is no longer possible to allocate words by just adjusting the value in ***SP*** to appropriately skip over them. Instead those words must be initialized (either to legitimate values or the special empty value) so that future write operations, which do check tags, will succeed. It would seem, however, that there is no way to get rid of both initialization on allocation and erasure on deallocation. One or the other is necessary to making the transition between successive instance tags for the words involved. Note, however, that words are often written anyway when being allocated on a stack whereas they are frequently freed without being accessed. Therefore, the incremental cost for using initialization seems to be less than that for using erasure.

One last comment on stacks: it might be possible to achieve further checking by using the page-tags proposed in Section 4-8. Pages used for implementing stacks would be tagged differently than those used for other purposes. Detailed examination of this area is left for future research.

---

[27] In fact, one might compare ***SP*** against ***FP*** instead of the absolute underflow boundary.

4-11. Resonance

Because small tags are used, it is possible, indeed probable, that tag values will be reused. This can lead to a phenomenon that I call *resonance.* In particular, there might be an unintentional coupling between tag allocation and usage that would increase the probability of an undetected error above ψ. The underlying problem is that the assumption of statistical independence of tag values may not hold in some cases.

Consider, for example, a program which allocates and frees 256 blocks of storage during each iteration of a loop. If r is 8 and the instance-tag allocator uses a simple 8-bit counter to generate tag values, then it would be possible for successive iterations to receive precisely the same storage locations and instance tags for corresponding objects. Therefore, after the first iteration, no attempts to use uninitialized variables would be detected as such.

Even without explicit repetition of tag values, undesirable coupling might occur between tag allocation and usage. There might be an interaction between individual tag-component values and the hashing function. For example, suppose that type ***T1*** is represented using type ***T2*** and that type ***T3*** is represented using type ***T4***. Suppose further that the hashing algorithm used is a Boolean sum. Should the numbers allocated to the tags for ***T1*** through ***T4*** be 2, 4, 1, and 7, then it would follow that

$$\texttt{hash(T1,T2)} = 010_2 \oplus 100_2 = 110_2$$

and

$$\texttt{hash(T3,T4)} = 001_2 \oplus 111_2 = 110_2.$$

In this case, the combined tag values would coincide even though the individual components do not. The net effect is that objects of type ***T1*** would be tagged the same as objects of type ***T2*** (in the absence of other tag components) and so the program would lose some of the run-time type-checking that one would otherwise expect it to have.

Resonance is not an easy problem to avoid. What seems to be the most effective way to combat it is the introduction of randomization of tags wherever it can done safely. Several precautions aimed in this direction look promising:

- Tag-component values should be assigned using an algorithm that avoids patterns that might follow the usage patterns found in programs. Autocorrelation of successive tag values should also be avoided. Otherwise one might encounter the effect seen in the second example above. In other words, a good random number generator may help avoid resonance.

- Tag values should be changed whenever practical. Even if resonance happens to hide an error before the reallocation, there is a good chance that the error will be detected using the new tag assignments (or *vice versa*). For example, many of the tags associated with

> types or modules could be reassigned each time a program is linked, much as addresses are currently reassigned by relocation. In this case, however, it would be desirable for successive runs of the linker to generate different tag assignments, even when presented with the same input. It might even be possible to delay the binding of tag values until the program is "loaded" for execution, thereby allowing different tag settings each time that the program is run. In general, the longer one can delay binding of tag values, the better.

Other approaches may be suggested by actual experience. In fact, I suspect that empirical data may provide far more insight into the issue than could possibly be anticipated before a machine is built.

Chapter 5
An example

Most of this dissertation is intentionally general, lest unnecessary details seem to eliminate valid design options. Nevertheless, an example (or at least a rough outline of one) seems appropriate to help clarify some of the concepts. Remember, however, that an example is no more than that—many of the specifications found below reflect arbitrary decisions.

There are two distinct paths that could have been adopted for generating an example machine: modifying an existing architecture or specifying a new one. The latter approach can be extremely seductive—almost every "hacker" has at least some desire to design a text editor, a programming language, and a computer architecture. A new design, however, is likely to reflect many decisions that are irrelevant to the purpose of the example. Just as multiple differences between *experimental* and *control* groups complicates experimental science, so introducing more than the bare minimum of new ideas to an example can obscure the points to be made. Therefore, it seems important to start with an existing machine and change as little as possible.

Many architectures could be modified without too much difficulty to include tagging. The multi-level type-tagging proposed in Section 3-2.2, however, can be best implemented if sealing and unsealing of types are included in the parameter-passing mechanism. It is especially helpful if the processor knows which routine is being called before preparing the argument list; in this case a parameter-descriptor can be associated with each routine. Excluding some LISP-oriented architectures, only two general-purpose machines seem to provide subroutine-specific preparation of argument-lists.[28] One, the now defunct Berkeley Computer Corporation's (BCC) Model 500 [93, 43], is interesting but is a decade-old design which will probably not be revived. The other, the Nebula architecture [84] being developed for the United States Army and Air Force as MIL-STD-1862A [66], is more in line with current trends and might even become widely used.

---

[28] The PRIME 50 Series architecture [72, 82] comes close, but processing of the argument list is specified in the calling routine, not the called routine.

The remainder of this chapter discusses how one might modify the non-privileged portion of the Nebula architecture to include tagging. Although they changes are small, they are not invisible to the programmer or compiler. Therefore, no claim is made that the tagged machine could be directly substituted for an untagged machine with existing software. On the other hand, it should be trivial to write new programs (or modify existing ones) so that they will be able to work properly in either case.

5-1. A summary of Nebula

Nebula is not yet a well known architecture so a short description seems in order. Although primarily a 32-bit architecture, it also fully supports 8- and 16-bit operands. A few operations, including the floating-point instructions, can also manipulate 64-bit quantities. For the most part, Nebula's instruction set resembles that of Digital Equipment Corporation's VAX [14]. As for the VAX, instructions are coded as a stream of bytes. The first byte specifies the *opcode* and subsequent bytes may be used to specify operands. Nebula differs in a few significant ways:

- The general registers are not global in scope; instead a new set of registers is allocated[29] at each procedure invocation. Only one register, the stack pointer, is automatically initialized after being allocated.
- A special parameter-passing mechanism is provided; more is said about this below.
- The size of each operand (8, 16, 32, or 64 bits) is specified independently from that of the others. In the VAX, a different opcode is provided for each acceptable combination of operand sizes; in Nebula only one opcode is used for all possible combinations. In effect, the operand size information has been moved from the opcode field to the operand specifier fields.
- Operand decoding has no significant side-effects (as opposed to the VAX which has *auto-increment* and *auto-decrement* modes). Thus, operands can be decoded in any order (or even in parallel) without problem. Furthermore, instruction processing can be restarted at the beginning without having to restore state information should a trap (such as a page fault) occur before its completion.

The operand specifiers, like those of the VAX, each consist of one or more bytes. Because they also specify the operand size, it is not possible to provide quite as many distinct "addressing modes" as on the VAX and still retain a single eight-bit byte to specify the mode (and register number, if any). The primary deletions from the addressing modes of the VAX are *auto-increment*, *auto-decrement*, and indirection through storage (*deferred*). The modes still provided are as follows:

---

[29] The allocation method is intentionally hidden from the programmer so that a particular model can take advantage of appropriate implementation technology.

- literal (5-, 8-, 16-, 32-, or 64-bit operand)
- register (32-bit operand)
- parameter
- indirect-register plus 0-, 8-, or 32-bit signed displacement (8-, 16-, 32-, or 64-bit operand)
- indirect-PC plus 8- or 32-bit signed displacement (8-, 16-, 32-, or 64-bit operand)
- absolute 32-bit address (8-, 16-, 32-, or 64-bit operand)

The 5-bit literal format is really an abbreviation for the 8-bit literal format; one byte specifies both the operand type and the value of the literal. It is otherwise treated like the 8-bit format.

There are also three "compound" addressing modes which contain one or two nested operand specifiers (represented in "Polish" notation):

- general parameter
- unscaled index
- scaled index

Evaluation of such compound specifiers is a two-step process: evaluate the nested operands and then compute the effective operand from these values.[30] The first compound specifier, *general parameter*, includes a nested operand specifier (rather than a constant in the instruction stream as occurs with the normal *parameter* mode) that determines which parameter is selected. It might be used by a routine that accepts a varying number of parameters—the nested operand might be a loop-counter, for example. The latter two compound specifiers each include two nested operand specifiers, one of which indicates a base memory address and the other an offset value to be added to that base. For scaled indexing, the offset is first scaled according to the size of the base operand. This allows easy subscripting of arrays containing 16-, 32-, or 64-bit elements. It is also useful for parameter arrays whose element-size might not be known in advance by the called routine.

The instruction set is fairly typical, but, unlike many machines, multi-operand instructions are not artificially constrained to using only one or two operands or to using implicit operands.[31] For efficiency there are also special versions of some of the instructions which use fewer operands. Examples of such are the two-operand *integer subtract* instruction in which a single operand indicates both the minuend and the difference and the one-operand *increment* instruction in which the operand doubles as an addend and sum while the other addend is an implicit literal. The non-privileged instructions may be summarized as follows:

---

[30] Although arbitrarily deep nesting may be represented, current implementation policy restricts usage to only one level of recursion.

[31] The one exception to this rule is that many instructions implicitly access the *condition* bits which are part of the *processor status word*. As for many current machine designs, such bits are considered the most practical way to record information such as *carries, overflows, comparison results*, etc.

- move, unsigned move, floating-point move
- push, pop
- exchange
- clear, floating-point clear
- move operand's size, move operand's address
- float, fix
- negate, floating-point negate
- absolute-value, floating-point absolute-value
- add, increment, unsigned add, add with carry, floating-point add
- subtract, decrement, unsigned subtract, subtract with carry, floating-point subtract
- multiply, fixed-point multiply, unsigned multiply, floating-point multiply
- divide, fixed-point divide, unsigned divide, floating-point divide
- remainder, modulus, floating-point remainder
- floating-point square-root
- floating-point round
- shift, rotate, binary scale
- and, or, exclusive-or, complement
- compare, test, unsigned compare, compare with bounds, compare and swap, floating-point compare
- block move, block fill, block translate, block compare, block scan
- signed field extract, unsigned field extract, unsigned field insert
- bit test, bit test and set, bit test and clear, bit test and complement
- call, return
- jump, branch, branch conditionally
- loop, increment and branch conditionally, decrement and branch conditionally
- case
- move *true* or *false* based on condition-bits
- set condition-bits
- set exception-handler, read exception-handler
- raise exception, return exception, propagate exception, read exception-code
- debugging break-point
- no-op

Although the addressing modes provide for four distinct operand sizes, some instructions support only a subset of these sizes. For example, few instructions other than those that manipulate floating-point quantities support 64-bit operands. When a particular instruction invocation specifies operands whose sizes are not supported by the hardware, the software is given a chance to produce an appropriate result (if there is one). Thus, if none of the operands of an *add* instruction are 64 bits long, then the instruction is executed in hardware. Otherwise, a software-defined implementation is invoked with the same parameters; it might use several 32-bit additions to implement the 64-bit operation.

The *call* instruction's syntax is like that of any other. It has one major difference, however, from the other instructions: the number of operands used with it is not fixed. Instead, the number of remaining operands is determined after evaluating the first one, which is the address of the routine to be called. The first two bytes of this routine are a *procedure descriptor* which, among other things, indicates how many parameters are expected.[32] Once this count has been determined, the remaining

[32] If the routine itself can accept a varying number of arguments, the descriptor so indicates and the count is instead taken from the calling instruction stream.

operands (which are the actual parameters for the procedure) can be evaluated. The resulting references (parameter-passing is by *reference* rather than *value*) are then stored in a manner specific to the implementation of the processor. Although a "context stack" is anticipated as the typical implementation method, there is quite a bit of flexibility because the only access is through *parameter* operand specifiers.

Making the parameter-passing mechanism for routine-calls similar to that used for instructions has a number of advantages:

- The implementation details of the argument list can be varied in each instantiation of the Nebula architecture to take advantage of appropriate technology.
- The calling instruction stream might become shorter due to elimination of the *move* opcodes normally associated with argument list preparation—only the operand specifiers themselves are required.
- Operand evaluation can easily be performed in parallel for increased speed of execution.
- It is straightforward to implement some instructions as subroutine calls (with an implicit first parameter). This is helpful for low-end implementations (in which these instructions might be most economically implemented in software) and for maintaining compatibility of old machines with new ones (by implementing newly defined instructions in software).
- Although not currently done in Nebula, some or all parameters could be "processed" as they are entered in the argument list with little added cost. For example, one might automatically copy the value of (rather than a reference to) those parameters that are supposed to be passed by value (as indicated by the procedure descriptor).

5-2. Tags used

For simplicity, the modified architecture does not use all of the applications discussed in Chapter 3. Three sorts of tagging are explicitly included:

- physical address tags
- instance tags
- abstract type tags

Physical address tagging, which is invisible to the program (except, perhaps, when an exception is raised), is used primarily to detect memory addressing problems. Because this tag is automatically derived from the physical address on each access to memory, it will do nothing to catch software errors.

Instance tags, on the other hand, are oriented more toward detecting software addressing

problems such as dangling pointers, array bound exceptions, uninitialized variables, etc. As it turns out, they can also detect some errors in virtual-memory mapping, either hardware or software induced. This follows the general principal that high-level checks can often detect low-level errors.

Abstract type tags carry software checking even further. Representation type tagging (at the instruction-set level) is omitted from the example machine because it would require either expansion of the instruction set (such as to provide a *move* instruction for each type) or equivalent expansion of the operand specifiers.[33] The software can, if desired, achieve nearly equivalent checking by encasing each hardware-provided type in a corresponding abstract type. The expense of so doing will be the additional overhead of specifying type-unsealing, either as subroutine calls or as in-line operand modifiers.

In addition to the "built-in" tag components, others may be added by the software. As for type checking, operand modifiers and the tag fields in pointers provide an interface that allows the hardware to check tags whose interpretation is known only to the software.

5-3. Tagged-cell size

Nebula was designed to be a byte-oriented machine. To fully maintain the flexibility that this can provide would probably require that tagging also be byte-oriented. Unfortunately, the overhead of a tag-per-byte seems excessive by current standards. Therefore, I have chosen to tag 32-bit words instead of individual bytes. This decision could be reversed in a later implementation without major impact.

The most significant implication of tagging words is that heterogeneous data structures occasionally become awkward to manipulate if packed more tightly than the word level. Much of the time-benefit of byte-level addressing, however, is obtained only for homogeneous structures. In heterogeneous structures, the elements are usually accessed separately from each other. In fact, if typing is being strictly enforced, this must be the case because the accesses will be from different modules! Therefore, assuming that the data-paths are a full word wide anyway, little or no time is gained by accessing bytes rather than words. Of course unpacked records might use more space than packed records. Only if the record is very large (which is not typical) or it has many instantiations (the more common problem), however, does this problem become significant. On the other hand, by

---

[33] In other words, although there is redundancy between many of the opcodes and the data they manipulate, the example will not harness this redundancy because in Nebula it is not quite sufficient. In a more drastic modification or a completely new design, this decision might well be reversed.

explicitly unpacking and packing around accesses to individual elements, this problem too can be avoided. For further discussion, refer back to Section 4-5.

The words used for homogeneous data structures can be treated quite differently—they can be tagged similarly to single elements of the given type. Furthermore, when accesses are made in a sequential manner (as is often the case for instruction streams or character strings), time can be saved due to packing because each word-wide access will allow processing of several elements without further accesses.

5-4. Tag size

In addition to choosing the size of storage cells to be tagged, one must also choose the size of the tag itself.[34] This choice is more critical for the visible tag components such as instance and type tags than for physical address tags. In particular, the size of the former two affects the coding of the instruction stream (for operand modifiers) and of pointer values (for the tag part). To retain the Nebula specification that pointer values fit in 32 bits, one must decrease the number of bits available for addressing by the size of the tag. In the example machine, Bits 1 through 6 will be used for this purpose. Bit 0 (the sign bit) and Bits 7 through 31 remain as addressing bits.[35] Using a 6-bit tag reduces ψ below 2% yet still leave $32-6=26$ bits for addressing. Although somewhat on the low side according to contemporary standards for newly designed machines, 26 bits of address will certainly suffice for many applications. In fact, most machines in use today provide a smaller virtual address space. Remember too that the physical address space need not be similarly restricted. Using the C-tagging method proposed in Section 2-1.5, the tag can be combined with an extended Hamming code without requiring any further bits—the 32 data and 7 check bits suffice.

5-5. Hashing

Because it is inappropriate to make a first implementation very complex, especially when one has little idea of the effectiveness of the new ideas, the incremental hashing algorithm is the Boolean (exclusive-or) sum of all the tag components.

To be combined with the other tag components, the physical address tag should also be six bits

---

[34] Actually, one must choose both the sizes of the individual tag components and the size of the combined (hashed) tag—they can differ. This is a secondary complication, however, and so will be ignored.

[35] The justification for this split is compatibility with the untagged Nebula machine in which Bit 0 distinguishes addresses assigned to the supervisor from those assigned to the user.

long. To minimize the likelihood of an undetected addressing error, all of the bits of the physical address should be included in the generation of this tag. Although one could just break the address into groups of six bits each and add the groups, I believe that there is a slightly better encoding of up to 32 address bits. Under this scheme, no two words whose addresses differ by fewer than four bits will be encoded with the same address tag. Thus, many bit-failures which occur while transmitting the address to the storage module will be caught at no extra cost.

$$T_1 = A_1 \oplus A_3 \oplus A_5 \oplus A_7 \oplus A_9 \oplus A_{11} \oplus A_{13} \oplus A_{15} \oplus A_{17} \oplus A_{19} \oplus A_{21} \oplus A_{23} \oplus A_{25} \oplus A_{27} \oplus A_{29} \oplus A_{31}$$

$$T_2 = A_2 \oplus A_3 \oplus A_6 \oplus A_7 \oplus A_{10} \oplus A_{11} \oplus A_{14} \oplus A_{15} \oplus A_{18} \oplus A_{19} \oplus A_{22} \oplus A_{23} \oplus A_{26} \oplus A_{27} \oplus A_{30} \oplus A_{31}$$

$$T_3 = A_4 \oplus A_5 \oplus A_6 \oplus A_7 \oplus A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15} \oplus A_{20} \oplus A_{21} \oplus A_{22} \oplus A_{23} \oplus A_{28} \oplus A_{29} \oplus A_{30} \oplus A_{31}$$

$$T_4 = A_8 \oplus A_9 \oplus A_{10} \oplus A_{11} \oplus A_{12} \oplus A_{13} \oplus A_{14} \oplus A_{15} \oplus A_{24} \oplus A_{25} \oplus A_{26} \oplus A_{27} \oplus A_{28} \oplus A_{29} \oplus A_{30} \oplus A_{31}$$

$$T_5 = A_{16} \oplus A_{17} \oplus A_{18} \oplus A_{19} \oplus A_{20} \oplus A_{21} \oplus A_{22} \oplus A_{23} \oplus A_{24} \oplus A_{25} \oplus A_{26} \oplus A_{27} \oplus A_{28} \oplus A_{29} \oplus A_{30} \oplus A_{31}$$

$$T_6 = A_0 \oplus A_3 \oplus A_5 \oplus A_6 \oplus A_9 \oplus A_{10} \oplus A_{12} \oplus A_{15} \oplus A_{17} \oplus A_{18} \oplus A_{20} \oplus A_{23} \oplus A_{24} \oplus A_{27} \oplus A_{29} \oplus A_{30}$$

These equations, or ones like them, can be implemented in a machine that already has an extended Hamming code without requiring any extra parity trees (although replication might prevent speed loss during *write* operations).

5-6. Registers

General registers, which are used to store intermediate results and frequently-accessed variables, have several special properties. Although they often provide faster access than other storage locations, caches can often achieve similar performance. More significant is the small number of bits needed to select them—register-addresses can be short because there are only a few of them. Not only does this reduce the size of the program but also it reduces the time (and number of instruction-stream accesses) needed to decode it. Using tagging with registers might therefore be impractical—there is even a good chance that one would have to use more bits to specify a tag than to select the register to which it applies!

Because the scope of most variables stored in registers tends to be limited and because one cannot generate a pointer-value that denotes a register, the incidence of erroneous programs accessing the wrong register is likely to be low. This is particularly true in the Nebula architecture;

the only sharing of registers between procedures is as explicitly passed parameters. Therefore, software-visible tags are not used for registers in the example. Physical address tags might still be appropriate, depending on the details of register implementation.

Of course, there is no reasonable way to prevent programs from specifying tags for operands that end up in registers. If nothing else, a parameter that normally requires unsealing might turn out to be located in a caller's register. Therefore, although operand modifiers must be acceptable for register operands, they should be ignored.

One beneficial effect of not tagging the registers is that in-line expansion of procedures defined in other modules will not require operand modifiers for those parameters that reside in registers. Therefore, if the compiler is able to keep active variables in registers, in-line expansion will not cost much more than it would in an untagged machine (although some checking would be sacrificed). Another benefit is that one need not worry about specifying a tag modifier for both the register and the final operand when using either of the two register-indexed modes; a tag is necessary only for the latter.

Some implementations of Nebula might store some of the registers, especially those corresponding to inactive procedure invocations, on the context-stack in main memory. In this case, it would probably be appropriate to include some tagging information with the stored values. For instance, one might use the same instance tag as is indicated by the stack pointer and a type-tag of *stored register.* This would prevent accesses by other than the register-manipulating mechanisms. To keep the change in physical location invisible to a legitimate accessor, these tag values would still not be affected by any operand modifiers supplied by the software.

5-7. In-line literals

For reasons similar to those mentioned for registers, operand modifiers applied to short (5-bit) literals should be ignored. In fact, because short literals are identifiable as such only *after* being fetched, this is mandatory. Note, however, that as for registers there is no way to share in-line literals between procedures other than as explicitly passed parameters and so software errors detectable by tagging are less likely to occur in the first place.

In some implementations, the value of a short literal might not be copied into the implementation-specific parameter-list during operand evaluation for procedure invocations. Instead, its address might be stored. In this case, any tags applied to the the formal parameter by the

called routine must be ignored in favor of those applicable to the actual parameter, as specified by the caller. That is, the tag used for fetching the value of a literal should be exactly the one that would normally be used for fetching from the (calling) instruction stream.

One can make a similar argument for the longer (8-, 16-, 32-, and 64-bit) literal modes, even though the overhead attributable to tagging for such literals is far less significant. Because these literals need not be fetched until they are actually used, the tag can be dynamically generated as for other memory references. The principal problem, however, is that the literal values might not occupy words distinct from those used for other parts of the instruction stream. This break in homogeneity would make the tagging of these packed words difficult, as explained in Sections 4-5 and 5-3. Furthermore, even if each byte were tagged independently of the others, instruction-stream prefetch mechanisms would become more complicated if the literals had to be skipped. It seems simplest, therefore, to also tag the longer literals only as part of the instruction stream, independent of operand modifiers.

5-8. Stack allocation

To help detect dangling pointers, uninitialized variables, etc., each stack frame should be given a (pseudo-) unique instance tag (as described in Section 4-10). In Nebula, procedure invocations allocate a new set of registers, including a new stack-pointer. This one register is automatically initialized by copying the value from the caller's stack-pointer. On return, no special action is taken, thereby effectively restoring the stack to the same state as it had at the time of the call. To add instance tags, one need only change this initialization process so that the six tag bits in the new stack pointer are set from a random-number generator rather than from the old stack-pointer.

One simple random number generator that would suffice for this purpose is a feedback shift-register, initialized at system-startup from a real-time clock. Note, however, that even a method as simple as a counter might be acceptable if the scheduling of independent activities occurs sufficiently frequently and randomly. Most important is that erroneous software not go undetected due to "resonance" with the instance-tag generator, as discussed in Section 4-11.

Instance tags should also be used for objects other than those on the stack. Although one might add a new instruction that would set the tag portion of a pointer to a random number, it might be easier just to use the normal procedure-call mechanism to access the random-number generator. For instance, the instance tag of a pointer returned by a storage allocator could be set from the instance-tag portion of the stack-pointer value used by that allocator.

5-9. Operand specifier changes

Even though the six bits of tag are mixed in with the address portion of pointer values, computation of the program-visible portion of the tag can be quite simple. In particular, address evaluation proceeds just as it would in an untagged machine except that the resulting value is interpreted as a tag and address rather than just an address. Note that "carries" from the address portion to the tag portion during indexing should not cause problems for reasonably written programs: Any carry that would adversely impact the tag field would just as adversely affect the address in a machine with more than 26 bits per address. A program has no right to make any assumption that the address spaces "wraps" at a particular point. Instead, the only knowledge it ought to need is the total number of bits needed for storing pointer values (so that it can allocate space for pointer variables). Ordered comparisons of pointer values, as well as subtraction of two different pointers to determine the size of the block contained between, are operations that would be affected by the presence of the tag field, but it is not hard to "write around" such problems.

To provide type-unsealing, as well as any other tag modification that the software might need to include, a new compound operand specifier must be added. This specifier would consist of one nested operand specifier and a 6-bit constant in the instruction stream (represented as a full byte with the extra two bits reserved for future expansion). An appropriate assembly language notation might be

```
x{y}
```

which would denote the operand *x* with tag modification *y*. Because literals and registers are tagged only with physical address tags, the appropriate semantics of the new specifier are as follows:

1. Evaluate the nested operand.

2. If the resulting effective operand denotes either a register or a literal, then skip over the tag constant in the instruction stream.

3. Otherwise, "exclusive-or" the tag constant with bits 1–6 of the effective address, yielding the new effective address.

4. In either case, the type and size of the resulting operand would match that of the nested operand.

5-10. Instruction changes

Although I have changed the operand-specification and memory-accessing mechanisms, most instruction definitions can otherwise remain as they were. The functionality of only two instructions in Nebula need be modified to support tagging: *move operand's address* and *call.*

The changes to the former are perhaps obvious—any accumulated tag value must be included in the tag portion of the pointer value generated. For example, the instruction

```
mova (R2){channel-tag}, R3
```

(where the *mova* instruction moves the address of the first operand to the cell denoted by the second operand) might be used to generate an unsealed pointer to a variable of type *channel.* The address portion of the pointer value stored in ***R3*** would match that found in ***R2*** but the tag portion would be the Boolean sum of that found in ***R2*** and the literal *channel-tag.*

The changes to the *call* instruction can be split into two groups—those that are mandatory and those that improve performance. The only mandatory change is that the argument list generated by a *call* instruction must include the appropriate tag field for any parameters that appear in memory. This is so that, after execution of an instruction like

```
call PrintInteger, C{channel-tag}
```

by the *PrintChannel* routine, the *PrintInteger* routine would have a parameter that would appear to be an *integer* rather than a *channel.*

In many cases, a routine that manipulates objects of a specific type will deal only with their representation. To make the code for such cases efficient, one should unseal parameters only once, such as at at procedure invocation. In fact, because the address of each parameter is already copied to the argument-list anyway, procedure-invocation is an excellent time to do such unsealing. By extending the procedure descriptor already found at the entry to each routine, one can provide the new information that specifies the extra tag-component to be added to the tag portion of certain parameters. Although there are many encodings possible, I will arbitrarily choose a simple one; some more thought might yield a cheaper or more flexible layout. Procedure descriptors currently consist of sixteen bits, one of which is unused. I will designate this bit to be the *unseal* bit. If set, then the procedure descriptor will also include one extra byte for each parameter expected.[36] These bytes are the extra tag-modifiers to be applied to the corresponding parameters (where zero can be used for parameters that need no unsealing).

---

[36] For obvious reasons the *unseal* bit cannot be used by procedures that accept a varying number of parameters—such routines will have to unseal their parameters using code inside the routine.

5-11. Summary of the changes

Despite the length of the preceding prose, the changes needed in the hardware to support tagging are minor:

- Six bits produced by operand evaluation are treated as a tag instead of as part of the virtual address. Along with the physical address tag, they are checked by C-tagging each 32-bit word.
- A new compound operand specifier is provided which can alter the value produced for the above bits.
- The procedure entry mechanism is changed to "randomize" the tag field in the stack pointer when creating a new stack frame.
- Procedure invocation is changed to check an *unseal* bit in the procedure descriptor. If set, each operand is interpreted as if it had been nested inside a tag-modifying operand specifier. The tag constants, in this case, are retrieved from the procedure descriptor.
- The parameter-list mechanism is extended to retain tags along with the other information describing actual parameters.

The benefits gained, on the other hand, are significant. With little or no loss in execution speed and little extra hardware, it is now possible to detect:

- references to uninitialized variables
- array boundary violations
- violations of modular boundaries
- accesses to the wrong location in storage
- accesses to words that have been erroneously overwritten
- erroneous transfer of control

Such additions ought to not only increase programmer productivity but also increase the reliability of the end product.

5-12. Omissions

Before building an actual machine, consideration should be given to a number of matters avoided in this example. Three areas that should be given deeper examination are:

- To detect errant stores as well as fetches, one should use a read-before-write scheme, as described in Section 4-1. If this is done, however, then there must be a way to erase cells that are now longer allocated. The current Nebula implementation of the program stack, however, offers too much flexibility—there is no simple way to properly implement the proposals of Section 4-10 for verifying changes to the stack-pointer. If, like the program-counter, the stack counter were not a general register, things would be simpler.

- Provision must be made for the transfer of data between primary and secondary storage, as discussed in Section 4-8. This might include the addition of secondary tags, as described at the end of that section.

- To keep things simple, E-tagging was not used. Its error-exaggeration properties, as described in Section 2-2.2, could be useful enough to warrant use in a real design.

Chapter 6
Conclusion

> What I want to explain in the Introduction is this. We have been nearly three years writing this book. We began it when we were very young . . . and now we are six. So, of course, bits of it seem rather baby-ish to us, almost as if they had slipped out of some other book by mistake. On page whatever-it-is there is a thing which is simply three-ish, and when we read it to ourselves just now we said, "Well, well, well," and turned over rather quickly. So we want you to know that the name of the book doesn't mean that this is us being six all the time, but that it is about as far as we've got at present, and we half think of stopping there.
>
> A. A. Milne [58 (p. x)]

A number of premises form the background for this work:

- Overall system reliability is an important goal which has not yet been sufficiently achieved.
- Fundamental to achieving reliability is the ability to detect errors promptly, before their effects can be propagated.
- Operations which must be repeated frequently are usually best implemented at a low level, such as in the hardware, rather than at higher levels, such as compiled code.
- Many of the errors causing problems in current systems exist only at a high level. That is, error checks cannot be provided completely at low levels because the inconsistencies exist only in the high-level semantics.
- Many high-level error-checks are widely known but often ignored because of the high costs associated with using them.

From these, it can be concluded that the error-checks dealing with high-level semantics but implemented at low levels would be desirable. This dissertation, by describing possible implementations of such mechanisms, provides an existence-proof that this idea is indeed feasible. In fact, by taking advantage of redundancy already present in current systems, the error-detection described can be added at minimal incremental cost.

6-1. Summary

To provide a demonstration of the practicality of checking for high-level semantic errors using low-level checking mechanisms, the preceding chapters describe a system that employs hashed tagging, supplemented by some special implementation techniques, to verify certain assertions about the values contained in cells of storage. In addition, to show that such checking is useful, candidates are proposed for the sorts of assertions that might be checked. Left to the reader, however, is final evaluation of the usefulness of particular applications of the mechanisms.

6-1.1. Hashed tags

A primary proposal of this dissertation is for the use of hashed tags. These tags can be used for a number of different purposes, all of which involve checking that words of storage and the accessing context agree upon some property of the data being accessed (such as its type). Unlike some other tagged architectures, the tags are used for checking purposes only, not dispatching. That is, it is only possible to check the tag value stored with a word for equivalence with an asserted value; it is not possible to examine a stored tag and control the computation according to its value.

By storing fewer bits of information than would be necessary for complete comparison, it becomes possible not only to implement tag-checking at low cost (according to both space and time measures) but also to check an arbitrary number of independent tag values using a fixed-size storage field. Of course, hash-collisions may allow some tag-comparisons to improperly report equivalence of logically different tag values. The result of this situation, however, is that although some incorrect items may be accepted no correct item will be rejected. That is, the addition of hashed tagging will not introduce any new errors to a correct program. Rather, a few tag-mismatches might remain undetected in an incorrect program. Note, however, these same errors and many more would have been undetected in the absence of tagging. Furthermore, the probability that a non-matching tag will be overlooked can be made arbitrarily small by increasing the size of the hashed tag. In fact, this probability will depend only on the number of bits in the hashed tag—for each bit added to the small tag, the probability of missing a difference will be roughly halved.

It is important to understand that adding new tag-components to a hashed tag does not substantially affect the detection-probability for mismatches in old tag-components. For example, the probability that an address-tag mismatch might slip by remains constant even when type-tags are added. At worst, the inclusion of the new tag might redefine the equivalence classes determined by hash-collisions. That is, the membership of the set of incorrect address-tags that would be incorrectly

accepted in a particular context might differ before and after the addition of type-tagging. The size of this set, however, would remain constant across the change and so the probability of an undetected mismatch would also remain constant.

6-1.2. Implementation

Two special techniques, C-tagging (see Section 2-1) and E-tagging (see Section 2-2) can make hashed tagging even more useful. While neither is critical to the concept of hashed tagging, each can contribute to the practicality thereof.

C-tagging allows the tag bits to be combined with the check bits already used for detecting (and correcting) storage failures. That is, in the typical computer system, a certain number of tag bits can be stored for "free." For example, the modified Nebula architecture described in Chapter 5 needs no extra bits for tags on data. The primary incremental costs of tagging in this case are the storage for type- and instance-tags in pointer values and the little extra CPU-logic necessary for performing the checking.

E-tagging, which uses encryption to append tags to data, provides slightly more subtle advantages. One is that many encryption functions can also be quite effective as hash functions—enough bit-shuffling is performed that the input domain (key and plaintext) will be mapped quasi-randomly yet uniformly onto the output domain. Thus, separate logic for encryption and hashing may not be required. More significantly, however, even a small perturbation of a value on one side of an encryption function can result in a fairly large perturbation in the corresponding value on the other side. This can allow detection of many of the errors that, due to hash-collisions, would slip past the primary tag comparison. In particular, those incorrect data items that are not detected by the low-level tag-checking mechanism might be detected at a higher (software) level because of distortion of the values fetched.

6-1.3. Applications

The traditional use for storage tags has been to denote the type of the data stored in each cell. Type-tagging is also a prime candidate for use with hashed tagging. There are a few differences, however, between the proposed tagging method and previous ones. Most notably, the tags are used only for checking, not for dispatch. That is, no provision is made for varying the computation according to the type of the data manipulated. Generic *add* operations, the behavior of which varies according to the tags of the operands, are not directly supported. Thus, the program must specify the

type of each operand accessed. For hardware-defined types, this specification would normally be implicitly indicated by the choice of instruction (*e.g.*, *integer-add* or *floating-add*). For software-defined types, it would normally be explicitly indicated as part of the definition of the type-defining module. Neither of these schemes adds significantly to program size or complexity. Unlike previous tagging schemes, however, hashed tags are quite efficient at representing multiple types for a single object. In particular, the addition of another level of abstraction to a particular object does not increase the size of its representation at all! This is because tag values that correspond to each abstract type can all be combined by the hashing function into a single tag to be stored with the data. In previous schemes, abstract types were either eliminated by the compiler (so that all data was tagged according to its representation type) or required extra tag fields, one for each level of abstraction.

Another use for storage tags—one that has been rarely used in the past—is for address-tagging. When each word is tagged with its address, it becomes possible to detect many addressing failures. That is, if the wrong word in storage is accessed then tag-comparison should detect the fact that the data fetched are not those desired. Although error-checking has often been performed on the transmission of addresses from the central processor to the storage module, the behavior of the address-decoding logic inside the storage module has usually gone unchecked (other, perhaps, than in off-line test programs). Furthermore, little checking has been done that the proper value is received from a multiplexed channel. For example, timing errors which cause a preceding or following word to be accepted from a bus instead of the desired one would typically not be detected except due to improper program behavior. On the other hand, if each word were tagged with its address, then selection of an incorrect word would be immediately detected. Address-tagging can be performed at a number of levels. For example, physical addresses could be automatically generated and checked by the hardware. Virtual address checking might also involve the operating system. At an even higher level, tagging of a word according to the programmed object in which it appears would allow detection of software-addressing errors such as exceeding the extent of an array.

A variation on address-tagging is instance-tagging. The primary virtue of an instance tag is that it changes each time that the corresponding storage cell is reallocated. Because the tag will be different when a cell is reused, errors such as the use of uninitialized variables or dangling pointers are detectable.

6-2. Evaluation

The purpose of this dissertation has been to introduce a novel form of tagging and to explore some of its potential applications. Because no prototype machine has been built, it is difficult to quantify the viability of the idea. Even if a machine had been built, the results from a single experimental model might be subject to question due to variations in implementation technology. Changes in configuration, such as which tag components are included for checking, might significantly affect any measured results. Nevertheless, a rough evaluation is possible.

Depending on what the combined tag contains, one might detect any of the errors listed in Chapter 3. As stated above, even the simple machine described in Chapter 5 can detect many instances of the following errors:

- references to uninitialized variables
- array boundary violations
- violations of modular boundaries
- accesses to the wrong location in storage
- accesses to words that have been erroneously overwritten
- erroneous transfer of control

Although detection of the first two errors has been implemented in a number of systems, such checking is usually confined to "debugging" or "checkout". The checks are considered too expensive for "production" usage. Tagging can make these checks practical even in the latter case. The remaining errors listed above have typically not been detected directly. Instead, they have been located only after secondary symptoms were observed and traced.

There is little objective data on the frequency of these errors. Experience has shown, however, that they are reasonably common. Furthermore, the cost of locating them can be quite high. Only a potential user can evaluate just how expensive they are. In addition to frequency of occurrence, some important factors to consider are:

- **The time needed to detect the errors:** It may take quite a bit of time just to discover that an error is present.
- **The time needed to isolate the errors, once detected:** The errors listed above are usually considered to be among the "nastiest" to locate because the symptoms appear when it is "too late" to determine their source.

- **The cost of undetected errors:** For man-rated (life-critical) situations, this can be unmeasurable. For financial systems, the expense can be large in an obvious manner. Even in non-critical applications the cost can be substantial—production down-time, loss of goodwill, etc.

All of these will vary from one application to another. In fact, it is possible that only actual experience with tagging will allow a fair evaluation. Nevertheless, the cost of each error is likely to be large. Whatever reduction can be achieved in this cost through the use of tagging is, by definition, equal to the benefit to be obtained from tagging.

The cost of adding tagging, on the other hand, is easier to estimate in advance. The example machine, for instance, has the following expenses over those of a similar untagged machine:

- The operand-evaluation mechanism has to be extended to generate a tag in addition to an address.
- Pointer values take up six more bits. (It might better reflect the implementation to say that six fewer bits are available for addresses.)
- Seven extra bits (or about 20% extra overhead) must be stored with each word of secondary storage used for implementing virtual memory. Words used for explicitly referenced files, on the other hand, need not be affected.
- The parity-trees used to generate the error-correcting code bits require a few extra inputs for the tag.[37]

For many systems in use today, such increases in expense would be small relative to total hardware cost.

In summary, it would be impossible for this dissertation to specify exactly what the costs and benefits of tagging would be for a particular application. That must be determined by someone better acquainted with the circumstances. Still, because the cost of the errors involved is likely to be high and the cost of catching them with tagging is low, it is probable that tagging will prove worthwhile. The benefit to be obtained by detecting even a few of them ought to be sufficient to justify the widespread use of tagging.

---

[37] It turns out that, for 32-bit words and 6-bit tags, binary trees would require no extra *depth* (and hence propagation delay).

6-3. Future research

As is normally the case, this research has raised at least as many questions as it answers. This section lists some of these outstanding problems and possibilities, along with hints to how they might be approached:

- This dissertation deals primarily with the detection of errors. Just as important, however, is recovery from errors after detection. Because they are used only in exceptional cases, it is usually not critical that error-correction mechanisms be highly optimized. Furthermore, the semantic information necessary for successful correction may be more extensive than that necessary for detection. Therefore, this seems to be an area best left to higher levels.

  Nevertheless, some support by low-level hardware may be appropriate. Randell [73] defined three features necessary for coping with error situations:

  (i) preparations for the possibility of errors;
  (ii) error detection facilities;
  (iii) error recovery facilities.

  Later, he along with others [31, 74] defined a mechanism which implements recovery from errors. Although their mechanism is quite general, they did not concentrate on error-*detecting* mechanisms; rather they dealt mostly with recovery after an error has been detected. In other words, they looked at (i) and (iii). My work, which concentrates on (ii), therefore fits nicely together with theirs.

- Robust data structures, as proposed by Taylor, Morgan, and Black [86, 87], also seem well suited for combination with my proposals.

- It is not at all clear whether resonance will be a significant problem. Although Section 4-11 offers some suggestions for avoiding difficulties, more research in this area may be necessary.

- There is a possibility that one could increase r and thus decrease ψ by tagging the cells of a multi-cell object quite differently from each other. One way to implement this would be for the incremental hash function to accumulate more bits of tag than are stored with each cell. For accesses to single cells, only a subset of the bits of the accumulated tag would be used. For multi-cell accesses, however, a different subset might be used for each cell. Thus, the effective value of r would depend upon the number of cells being accessed. It might increase linearly with the number of cells accessed, up to the limit imposed by the tag accumulation mechanism. Note, however, that one's choice of this upper limit might be strongly affected by how many bits one is willing to include in the tag portion of pointer values (as described in Section 3-1.2).

- As discussed in Section 2-1.7, C-tagging could be used with codes with minimum distance greater than four (*i.e.*, better than SEC-DED). Although the *one-of-n* model described in Section 2-1.5 can be used with greater-distance codes, it is possible that other methods for encoding the tag would allow more distinct tag values.

- There are several outstanding issues with respect to E-tagging. Most notable is choice of a

practical encryption function. Although security from intentional attack is not required, the cipher used should be sensitive to changes, exaggerating small changes in the key or ciphertext to be large changes in the decrypted cleartext. Although DES can be executed quickly relative to many other encryption functions, it still is slow relative to the data-accessing rate of most current machines. Although pipelined implementations would be able to keep up with primary storage bandwidths, the startup latency would still be relatively slow. One promising approach would be to place encryption between primary storage and the processor's cache, which would contain both the cleartext of each word and the key used to obtain it.

- An interesting possibility arises if E-tagging is performed using a public-key encryption algorithm: tagging could enforce selective access to storage (either *read* or *read/write*). To fetch from a location, one would need only the decryption key. To write into a location, however, one would have to present both the decryption key (to check that the correct word is being accessed) and the encryption key (to store the new value).

- The example machine described in Chapter 5 uses conventional addressing methods. There seems to be no reason that my tagging proposals could not also be used with capability-based addressing. In fact, the tag-checking would provide a good check that the capability system is working properly.

6-4. Parting words

At this point, it seems that the most appropriate course of action would be to build a machine that incorporates the ideas presented. The additional hardware and execution time required for tagging are trivial. Even if only a few errors are detected by tag checking, the added cost will have been worthwhile. Although none was built as part of the research to date, all indications are that there will be substantial gains at minimal cost. In the end, however, only actual usage can provide the data necessary to make a complete evaluation. To quote an old advertisement, *"Try it; you'll like it!"*

Appendix A
An observation concerning DES

One interesting technique employed as part of DES is the one that makes it invertible. The central part of the encryption process uses several iterations of the same function, each using a different key. This can be expressed using the iterative formula

$$X_{i+1} = f(X_i,K_i).$$

The most obvious way to allow decryption is to make f invertible so that one can just iterate backwards using f^{-1} in the formula

$$X_{i-1} = f^{-1}(X_i,K_{i-1}).$$

Rather than risk weakening the basic encryption function in order to make it invertible, the designers of DES added a post-processing step which maps *any* function into an invertible one. That is, they defined the above mentioned f applied to X in terms of another function f' applied to the two halves of X (which are called L and R) alternately. Thus the first formula above was replaced by the pair of formulae[38]

$$L_{i+2} = L_i \oplus f'(R_i,K_i)$$

and

$$R_{i+2} = R_i \oplus f'(L_{i+2},K_{i+1}).$$

Decryption can then be performed just by using the inverse iteration formulae

$$R_{i-2} = R_i \oplus f'(L_i,K_i)$$

and

$$L_{i-2} = L_i \oplus f'(R_{i-2},K_{i-1}).$$

Because f' is always used in the "forward" direction, there is no need for one to ever try to compute its inverse. In fact, the function chosen for use in DES employs non-invertible substitutions.

One side effect of this implementation is that one can modify the algorithm to accept 64 data bits plus check bits as input and yield 64 encrypted bits plus check bits corresponding to the output word without explicitly regenerating them. As a simple example, consider the case of parity. Assume that

---

[38] The subscripts of iteration increment by two rather than one for consistency with the wording of the official definition [65].

in addition to L and R the encryption function is presented a parity bit P. The encryption formulae given above are then augmented with

$$P_{i+2} = P_i \oplus p(f'(R_i,K_i)) \oplus p(f'(L_{i+2},K_{i+1}))$$

where p is a function that computes the parity of its argument. Because f' is computed by table lookup, it is trivial to precompute appropriate extra bits in the tables that can be used in the generation of the parity function listed above. If there are no errors, then at each step in the iteration, P_i will be the appropriate parity bit for L_i and R_i. Furthermore, the data paths used during the encryption itself will be checked; if any single-bit error occurs either before, during, or after encryption then the resulting output will reflect this with "bad" parity. The only paths that are not included in this check are those used to carry addressing information during the table lookups for f'. All other paths and intermediate registers are covered.[39] Decryption under DES is nearly identical to encryption so the same assertions can be made for that case as well.

By obvious extension, one can substitute a Hamming (or other linear binary) code for the simple parity code described above. For example, one could accept as input 64 bits of plaintext plus the 8 corresponding extended Hamming check bits and generate as output 64 bits of ciphertext plus 8 corresponding check bits. Due to the spreading of information across bits in the DES algorithm, correction of errors that occurred before or during encryption would not be possible after encryption. Although one could determine which position had failed, the erroneous value would have already been propagated to many other positions. Nevertheless, detection of errors would be quite practical. Furthermore, one would not have to delay further processing to allow time for generation of the new check bits; they would be computed incrementally in parallel with the ciphertext itself. Note also that, in the absence of errors, any intentional bias (as proposed in Section 2-1.5) in the check bits of the input will be preserved in those of the output.

[39] Failure of a particular bit position in several different stages of the iteration are considered a multiple-bit failure for this analysis.

Appendix B
A fast parity encoder

Traditional parity encoders require *O(log m)* time and *O(m)* gates to generate an $(m+1)$-bit word with even parity from an uncoded m-bit word. One interesting side-result of the research for this dissertation was the realization that this is not necessary—it is possible to build a parity encoder that works in constant time. Although the resulting code is not *separable*, it still can be reasonably decoded.

The encoding method chosen is similar to that standard used for converting straight binary numbers to their Gray code equivalents except that the $m+1$ bits are produced. That is, the original word is combined using a bit-wise exclusive-or operation with a copy of the word shifted by one position. For an input word such as

$$B_1 \quad B_2 \quad B_3 \quad \ldots \quad B_m$$

,the output word would be

$$B_1 \quad (B_1 \oplus B_2) \quad (B_2 \oplus B_3) \quad \ldots \quad (B_{m-1} \oplus B_m) \quad B_m.$$

Decoding of the word can be implemented either serially (using a ripple-carry like scheme) in *O(m)* time and *O(m)* gates or in parallel using *O(log m)* time and *O(m log m)* gates. The latter configuration can be quite naturally implemented in a manner like that proposed by Brent and Kung [8] for carry propagation in standard addition.

The primary disadvantage of using this scheme is that decoding is slightly more expensive, either in time or in gate-count, than the traditional scheme. Nevertheless, there are situations in which this trade-off is appropriate. Principal candidates are those in which generation of coded values is more of a bottleneck than decoding. One special case comes to mind: to merge tags with Hsiao's modified extended Hamming code, as discussed in Section 2-1.6, the tag must have even parity. Given that one only encodes such tags and never decodes them, the extended Gray coding seems quite appropriate.

Glossary

Because this dissertation draws on several distinct areas of research, it is likely that the reader will be unfamiliar with some of the terminology used. Therefore, this glossary gives appropriate informal definitions.

For the sake of brevity, explanations that appear in Section 1-2.1 are not repeated here. See that section, therefore, for further information on many of the terms listed below.

Special symbols

$\wedge$: the Boolean *intersection (and)* operator.

$\oplus$: the Boolean *sum/difference (exclusive-or)* operator.

ψ: the (small) probability that an unpatterned error will go undetected. Approximately equal to 2^{-r}, for non-trivial values of r.

(n,k,d): a notation used for describing error codes. n is the number of *characters* (equivalent to *bits* for binary codes) in each *codeword*, k of which correspond to useful data. The minimum distance between any two codewords is d.

r: an abbreviation for $n-k$. This can be considered a measure of the redundancy of the code.

Technical terms

address tag: a tag (component) that is derived from the address at which the tagged value will be stored.

alias: an alternate name, with a given name-space, for a single object.

capability: a pointer value that includes both the address of an object and an indication of the operations that may be performed on the object using that capability.

character: the basic element from which a codeblock is formed. For binary codes, this is equivalent to a *bit*.

cipher: an encryption method that employs transposition or substitution at the character (as opposed to word, phrase, or higher) level.

ciphertext: encrypted text, the result of encrypting plaintext.

cleartext: unencrypted text (a synonym for *plaintext*).

codeblock: the collection of characters that are checked by an error-detecting or -correcting code.

codeword: those codeblock values that are designated in a code as "valid" (*i.e.*, error-free).

cryptanalysis: the process of deciphering encrypted text without knowledge of the "secret" portion (*e.g.*, key) of the encryption method used.

C-tag: a tag that is merged with the check bits of an error-detecting or -correcting code.

DES: Data Encryption Standard—an encryption algorithm adopted as a federal standard [65] by the United States National Bureau of Standards.

distance: a metric indicating the difference between two values. Hamming distance, which is the number of characters that differ between the two values, is most often used for binary codes. When used in the context of codes, this term normally refers to the minimum distance between any two codewords. This is the minimum number of characters that must change for an error to be totally undetected.

external redundancy: describes a code in which part of the codeblock is kept externally from the rest.

error vector: the Boolean difference between the intended codeword and the codeblock value actually received.

E-tag: a tag merged with data using encryption.

hash function: a function that maps from a large domain onto a relatively small range. A "good" hash function "spreads" the results uniformly throughout its range when presented with a "typical" set of inputs.

Iliffe vector: a vector of pointers to vectors. Iliffe vectors are normally used to implement multi-dimensioned arrays with one level of indirection for each dimension except the last.

instance tag: a tag (component) that indicates the allocation-instance of a storage cell. This changes each time that a cell is freed and reallocated. Its primary purpose is to detect dangling (obsolete) references to storage.

internal redundancy: describes a code in which the entire codeblock appears together.

key: That portion of an encryption method that must be kept secret in order to preserve its security. Typically, changing keys is simpler than switching to a different encryption algorithm.

linear code: a subspace of the space of all n-tuples of a particular set of characters. Binary codes are the transitive closure of Boolean addition on a set of linearly independent *generator* Boolean vectors,

one-of-*n* code: an *n*-character code which has *n* codewords, each of which has only one non-zero character. For example, a binary one-of-three code has the codewords 001, 010, and 100. All other three-bit values (000, 011, 101, 110, and 111) are invalid and indicate an error.

memory failure: a change in a stored value between when it was stored and when it is fetched.

operand modifier: a field used as part of operand specification in machine-language instructions. See Section 3-2.3.2.

overloaded operator: an operator which has multiple definitions, distinguished by the type(s) of its operands.

plaintext: unencrypted text (a synonym for *cleartext*).

product cipher: a cipher produced by applying two or more ciphers sequentially.

public-key encryption: encryption using an algorithm which uses different keys for encryption and decryption. See Section 1-3.

random error: an error that follows a pattern in which each character of the codeblock is affected independently of the others. That is, there is no correlation between the failure of one character and the failure of another.

residue code: a code in which the check-bits are the modular residue of the data bits. For example, a binary residue-three code would have two check bits indicating a value congruent, modulo three, to the data value. Residue codes are often used to check arithmetic operations because the check bits can be added independently of the data and the two sums can be checked against each other. Note also that a binary odd-residue code can detect single-bit memory failures.

resonance: an undesired (an unintentional) coupling between tag allocation and tag reference patterns. See Section 4-11.

sealing: the process of converting an object (or a reference to it) from its representation type to its abstract type.

SEC-DED: single error correcting, double error detecting. This term is used to describe codes with minimum distance four that are used for correction.

selector: an item which selects a part of a structured object, such as a subscript or record field-name.

separable code: a code in which the data-bits can be distinguished from the check-bits (a synonym for *systematic code*). That is, the unencoded data characters appear as a subset of the encoded codeblock.

slicing: the process of applying a selector to a structured object, yielding a sub-object.

symptom: those things which are visibly changed because of an error.

systematic code: a code in which the data-bits can be distinguished from the check-bits (a synonym for *separable code*). That is, the unencoded data characters appear as a subset of the encoded codeblock.

tag: a field appended to a data value which describes some attribute of that value.

truncated code: a code which is derived from some other code by dropping some of the codewords and, perhaps, some of the characters of the codeblock.

unsealing (of types): the process of converting an object (or a reference to it) from its abstract type to its representation type.

unpatterned error: an error which follows no particular pattern. That is, when an error occurs one can expect half the bits to be affected.

weight: a metric indicating the distance of a value from the zero-vector. Hamming weight, which is the number of non-zero bits in the value, is most often used for binary codes.

References

[1] Philip S. Abrams.
An APL Machine.
PhD thesis, Stanford University, February, 1970.

[2] Alfred V. Aho and Jeffrey D. Ullman.
Principles of Compiler Design.
Addison-Wesley Publishing Company, 1977.

[3] *ANSI standard X3.53-1976: Programming Language PL/I*
American National Standards Institute, Inc., New York, 1976.

[4] Henry G. Baker, Jr.
List Processing in Real Time on a Serial Computer.
Communications of the Association for Computing Machinery 21(4):280–294, April, 1978.

[5] H. Bekić.
An Introduction to ALGOL 68.
Annual Review in Automatic Programming 7(3):143–169, 1973.

[6] Peter B. Bishop.
Computer Systems with a Very Large Address Space and Garbage Collection.
PhD thesis, MIT/LCS/TR-178, Massachusetts Institute of Technology, Laboratory for Computer Science, May, 1977.

[7] Harry Q. Bovik.
Report on the Programming Language Pidgin 81.
Technical Report, Carnegie-Mellon University, Department of Computer Science, February 30, 1981.

[8] R. P. Brent and H. T. Kung.
A Regular Layout for Parallel Adders.
Technical Report CMU-CS-79-131, Carnegie-Mellon University, Department of Computer Science, June, 1979.

[9] D. L. Chaum and Robert S. Fabry.
Implementing Capability-Based Protection Using Encryption.
Memorandum UCB/ERL M78/46, University of California at Berkeley, Electronics Research Laboratory, July, 1978.

[10] Jack B. Dennis and Earl C. Van Horn.
Programming Semantics for Multiprogrammed Computations.
Communications of the Association for Computing Machinery 9(3):143–155, March, 1966.

[11] Whitfield Diffie and Martin E. Hellman.
New Directions in Cryptography.
IEEE Transactions on Information Theory IT-22(6):644–654, November, 1976.

[12] Whitfield Diffie and Martin E. Hellman.
Exhaustive Cryptanalysis of the NBS Data Encryption Standard.
Computer 10(6):74–84, June, 1977.

[13] Whitfield Diffie and Martin E. Hellman.
Privacy and Authentication: An Introduction to Cryptography.
Proceedings of the IEEE 67(3):397–427, March, 1979.

[14] *VAX11/780 Architecture Handbook*
Digital Equipment Corporation, 1977.

[15] Arthur Evans, Jr.
PAL—A Language Designed for Teaching Programming Linguistics.
In *Proceedings of the 23rd ACM National Conference*, pages 395–403. Association for Computing Machinery, 1968.

[16] Horst Feistel.
Cryptography and Computer Privacy.
Scientific American 228(5):15–23, May, 1973.

[17] Edward A. Feustel.
The Rice Research Computer—A tagged architecture.
In *Proceedings of the 1972 Spring Joint Computer Conference*, pages 369–377. American Federation of Information Processing Societies, Montvale, New Jersey, May, 1972.

[18] Edward A. Feustel.
On the Advantages of Tagged Architecture.
IEEE Transactions on Computers C-22(7):644–656, July, 1973.

[19] Edward F. Gehringer.
Functionality and Performance in Capability-Based Operating Systems.
PhD thesis, Purdue University, May, 1979.

[20] E. Gelenbe and Claude Kaiser (editors).
Lecture Notes in Computer Science. Number 16: *Operating Systems: Proceedings of an International Symposium held at Rocquencourt, April 23–25, 1974.*
Springer-Verlag, 1974.
Note: this is a reprint of *Colloques IRIA: Aspects Théoriques et Pratiques des Systèmes d'Exploitation.*

[21] E. N. Gilbert, Florence Jessie MacWilliams, and Neil James Alexander Sloane.
Codes Which Detect Deception.
Bell System Technical Journal 53(3):405–424, March, 1974.

[22] Virgil D. Gligor and Bruce G. Lindsay.
Object Migration and Authentication.
IEEE Transactions on Software Engineering SE-5(6):607–611, November, 1979.

[23] David Gries.
Compiler Construction for Digital Computers.
John Wiley & Sons, 1971.

[24] Richard W. Hamming.
Error Detecting and Error Correcting Codes.
Bell System Technical Journal 29(2):147–160, April, 1950.

[25] Richard W. Hamming.
Coding and Information Theory.
Prentice-Hall, 1980.

[26] Hewlett-Packard Company.
"HP-35 Errata".
Descriptive card available on request from HP Customer Support, 1000 N.E. Circle Boulevard, Corvallis, Oregon 97330.

[27] C. A. R. Hoare.
Hints on Programming Language Design.
Technical Report AIM-224 (STAN-CS-73-403), Stanford Artificial Intelligence Laboratory (Computer Science Department, Stanford University), 1973.

[28] Jack Holloway.
PDP-10 Paging Device.
Hardware Memo 2, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February, 1970.

[29] *Series 60 (Level 68) Multics Processor Manual*
Honeywell Information Systems Inc., 1979.
Order number AL39.

[30] Honeywell, Inc. and CII Honeywell Bull.
Reference Manual for the Ada Programming Language.
Proposed Standard Document, United States Department of Defense, July, 1980.

[31] James J. Horning, Hugh C. Lauer, Peter M. Melliar-Smith, and Brian Randell.
A Program Structure for Error Detection and Recovery.
In *Colloques IRIA: Aspects-Théoriques et Pratiques des Systèmes d'Exploitation*, pages 177–193. Institut de Recherche d'Informatique et d'Automatique, BP5-Rocquencourt, 78150 Le Chesnay, France, 23–25 avril 1974.
Complete proceedings reprinted by Gelenbe and Kaiser [20]; this article is on pages 171–187.

[32] Mu-Yue Hsiao.
A Class of Optimal Minimum Odd-weight-column SEC-DED Codes.
IBM Journal of Research and Development 14(4):395–401, July, 1970.

[33] Jean D. Ichbiah *et al.*
Preliminary ADA Reference Manual
Honeywell, Inc. and CII Honeywell Bull, 1979.
Reprinted in *ACM SIGPLAN Notices* 14(6), June, 1979.

[34] J. K. Iliffe.
Basic Machine Principles.
American Elsevier Publishing Company, 1968.

[35] J. K. Iliffe.
Elements of BLM.
Computer Journal 12(3):251–258, August, 1969.

[36] *Introduction to the iAPX 432 Architecture*
Intel Corporation, 1981.
Order number 171821-001.

[37] *Special Feature Instructions: IBM 1401 Data Processing System & IBM 1460 Data Processing System*
International Business Machines Corporation, 1964.
Form A24-3071-2.

[38] *IBM System/370 Principles of Operation*
Fifth edition, International Business Machines Corporation, 1976.
Order number GA22-7000-5.

[39] David Kahn.
The Codebreakers: The Story of Secret Writing.
The Macmillan Company, 1967.

[40] Peter M. Kogge and Harold S. Stone.
A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations.
IEEE Transactions on Computers C-22(8):786–793, August, 1973.

[41] Peter M. Kogge.
Maximal Rate Pipelined Solutions to Recurrence Problems.
In *Proceedings of the First Annual Symposium on Computer Architecture*, pages 71–76. December, 1973.
Computer Architecture News 2(4), December, 1973.

[42] Peter M. Kogge.
The Architecture of Pipelined Computers.
Hemisphere Publishing Corporation & McGraw-Hill Book Company, 1981.

[43] Butler W. Lampson.
Some Remarks on a Large New Time-Sharing System.
In *Computer 70*, pages 74–81. United States Department of Commerce and The Association for Computing Machinery, October, 1970.

[44] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek.
Report on the Programming Language Euclid
1976.
Reprinted as ACM SIGPLAN Notices 12(2), February 1977.

[45] Butler W. Lampson and Robert F. Sproull.
An Open Operating System for a Single-User Machine.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 98–105. Association for Computing Machinery, December, 1979.

[46] Abraham Lempel.
Cryptology in Transition.
Computing Surveys 11(4):285–303, December, 1979.

[47] Sik K. Leung-Yan-Cheong and Martin E. Hellman.
Concerning a Bound on Undetected Error Probability.
IEEE Transactions on Information Theory IT-22(2):235–237, March, 1976.

[48] Sik K. Leung-Yan-Cheong, Earl R. Barnes, and Daniel U. Friedman.
On Some Properties of the Undetected Error Probability of Linear Codes.
IEEE Transactions on Information Theory IT-25(1):110–112, January, 1979.

[49] C. H. Lindsey and S. G. van der Meulen.
Informal Introduction to ALGOL 68.
North-Holland Publishing Company, 1977.

[50] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert.
Abstraction Mechanisms in CLU.
Communications of the Association for Computing Machinery 20(8):564–576, August, 1977.

[51] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder.
CLU Reference Manual.
Technical Report MIT/LCS/TR-225, Massachusetts Institute of Technology, Laboratory for Computer Science, October, 1979.

[52] Allen W. Luniewski.
The Architecture of an Object Based Personal Computer.
PhD thesis, MIT/LCS/TR-232, Massachusetts Institute of Technology, Laboratory for Computer Science, December, 1979.

[53] Florence Jessie MacWilliams.
Error Correcting Codes—An Historical Survey.
In Henry B. Mann (editor), *Error Correcting Codes.* John Wiley & Sons, 1968.

[54] Florence Jessie MacWilliams and Neil James Alexander Sloane.
The Theory of Error-Correcting Codes.
North-Holland Publishing Company, 1977.

[55] Ralph C. Merkle.
Secure Communications over Insecure Channels.
Communications of the Association for Computing Machinery 21(4):294–299, April, 1978.

[56] A. A. Milne.
When We Were Very Young.
E. P. Dutton & Co., 1924.

[57] A. A. Milne.
Winnie-the-Pooh.
Methuen & Co. Ltd., London, 1926.

[58] A. A. Milne.
Now We are Six.
E. P. Dutton & Co., 1927.

[59] Christopher Milne.
The Enchanted Places.
E. P. Dutton & Co., 1974.

[60] James G. Mitchell, William Maybury, and Richard E. Sweet.
Mesa Language Manual.
Technical Report CSL-79-3, Xerox, Inc. (Palo Alto Research Center, Systems Development Department), Palo Alto, California, April, 1979.

[61] James H. Morris, Jr.
Protection in Programming Languages.
Communications of the Association for Computing Machinery 16(1):15–21, January, 1973.

[62] Glenford J. Myers.
Software Reliability, Principles and Practices.
John Wiley & Sons, 1976.

[63] Glenford J. Myers.
The Design of Computer Architectures to Enhance Software Reliability.
PhD thesis, Computer Science Division, Polytechnic Institute of New York, June, 1977.

[64] Glenford J. Myers.
Advances in Computer Architecture.
John Wiley & Sons, 1978.

[65] *Data Encryption Standard*
National Bureau of Standards, 1977.
Federal Information Processing Standard (FIPS) Publication Number 26.

[66] *Nebula Instruction Set Architecture (MIL-STD-1862A)*
Naval Publications Center, Philadelphia, 1981.

[67] Roger M. Needham.
Adding Capability Access to Conventional File Servers.
ACM Operating Systems Review 13(1):3–4, January, 1979.

[68] Joseph M. Newcomer *et al.*
HYDRA: Basic Kernel Reference Manual.
Technical Report, Carnegie-Mellon University, Department of Computer Science, November, 1976.

[69] Elliott I. Organick.
Computer System Organization: The B5700/B6700 Series.
Academic Press, 1973.

[70] William Wesley Peterson and E. J. Weldon, Jr.
Error-Correcting Codes (second edition).
The MIT Press, 1972.

[71] Steven C. Pohlig.
Algebraic and Combinatoric aspects of Cryptography.
PhD thesis, Stanford University, October, 1977.

[72] *System Architecture Reference Guide*
PRIME Computer, Inc., 1981.
Document number PDR3060.

[73] Brian Randell.
Operating Systems: the Problems of Performance and Reliability.
In *Information Processing 71: Proceedings of IFIP Congress 71*, pages 281–290. International Federation for Information Processing, North-Holland Publishing Company, 1972.

[74] Brian Randell.
System Structure for Software Fault Tolerance.
IEEE Transactions on Software Engineering SE-1(2):220–232, June, 1975.

[75] David D. Redell.
Naming and Protection in Extensible Operating Systems.
Technical Report MAC-TR-140, Massachusetts Institute of Technology, Laboratory for Computer Science (formerly Project MAC), November, 1974.
Originally a PhD thesis, University of California at Berkeley, September 1974.

[76] Marian Rejewski.
How Polish Mathematicians Broke the Enigma Cipher.
Annals of the History of Computing 3(3):213–234, July, 1981.

[77] Ronald L. Rivest, Adi Shamir, and Leonard Adleman.
A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.
Communications of the Association for Computing Machinery 21(2):120–126, February, 1978.

[78] Ronald L. Rivest.
A Description of a Single-Chip Implementation of the RSA Cipher.
Lambda 1(3):14–18, Fourth Quarter, 1980.
Redwood Systems Group, P.O. Box 50503, Palo Alto, California 94303.

[79] Richard M. Sedmak.
Availability, Reliability, and Maintainability.
1979.
Preliminary draft of a chapter of a book in preparation at Sperry Univac.

[80] Claude E. Shannon.
Communication Theory of Secrecy Systems.
Bell System Technical Journal 28(4):656–715, October, 1949.

[81] Peter W. Shantz, R. Angus German, James G. Mitchell, Richard S. K. Shirley, and C. Robert Zarnke.
WATFOR—The University of Waterloo FORTRAN IV Compiler.
Communications of the Association for Computing Machinery 10(1):41–44, January, 1967.

[82] Rosemary Shields.
The Assembly Language Programmer's Guide
PRIME Computer, Inc., Framingham, Massachusetts, 1979.
Document number FDR3059-101A.

[83] J. Lynn Smith.
The design of Lucifer, a cryptographic device for data communications.
Technical Report RC 3326, IBM T.J. Watson Research Center, Yorktown Heights, New York, April, 1971.

[84] Leland Szewerenko, William B. Dietz, and Frank E. Ward.
Nebula: A New Architecture and Its Relationship to Computer Hardware.
Computer 14(2):35–41, February, 1981.

[85] Donald T. Tang and Robert T. Chien.
Coding for error control.
IBM Systems Journal 8(1):48–86, 1969.

[86] David J. Taylor, David E. Morgan, and James P. Black.
Redundancy in Data Structures: Improving Software Fault Tolerance.
IEEE Transactions on Software Engineering SE-6(6):585–594, November, 1980.

[87] David J. Taylor, David E. Morgan, and James P. Black.
Redundancy in Data Structures: Some Theoretical Results.
IEEE Transactions on Software Engineering SE-6(6):595–602, November, 1980.

[88] *990 Computer Family Systems Handbook*
Second edition, Texas Instruments Incorporated, 1975.
Manual number 945250-9701.

[89] Wing N. Toy.
Fault-Tolerant Design of Local ESS Processors.
Proceedings of the IEEE 66(10):1126–1145, October, 1978.

[90] John C. Traupman.
The New College Latin & English Dictionary.
Grosset & Dunlap, 1966.

[91] Paul Tyner.
iAPX 432 General Data Processor Architecture Reference Manual
Intel Corporation, 1981.
Order number 171860-001.

[92] John F. Wakerly.
Error Detecting Codes, Self-Checking Circuits, and Applications.
Elsevier North-Holland, 1978.

[93] *BCC 500 CPU Reference Manual*
University of Hawaii, The Aloha System, 1973.

[94] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker.
Revised Report on the Algorithmic Language ALGOL 68.
Acta Informatica 5(1–3):1–236, 1975.
Reprinted by Springer-Verlag, 1976 and in *ACM SIGPLAN Notices* 12(5):1-70, May, 1977.

[95] F. W. Winterbotham.
The Ultra Secret.
Harper & Row, Publishers, 1974.

[96] William A. Wulf, Ellis Cohen, William M. Corwin, Anita K. Jones, Roy Levin, Charles Pierson, and Frederick Pollack.
HYDRA: The Kernel of a Multiprocessor Operating System.
Communications of the Association for Computing Machinery 17(6):337–345, June, 1974.

[97] William A. Wulf, Roy Levin, and Samuel P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill Book Company, 1981.