MAGPIE: PRECISE GARBAGE COLLECTION FOR C

by

Adam Wick

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

June 2006

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Adam Wick

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Matthew Flatt
	John Regehr
	Gary Lindstrom
	Wilson Hsieh
	R. Kent Dybvig

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Adam Wick in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Matthew Flatt
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

C and C++ provide fast, flexible substrata for programs requiring speed or tight coupling with the operating system or hardware. Both languages have well established user and code bases, including programs still in use after decades of development. Unfortunately, with C and C++'s speed and flexibility come increased complexity, including complication in managing memory. Programs must create and destroy objects explicitly, and small mistakes in doing so can cause severe complications.

In other languages, precise garbage collection solves these problems by having the computer manage the program's memory. However, until now, adding precise garbage collection to standard C programs has been a considerable amount of work. This dissertation describes Magpie, a system that uses several analyses and conversion techniques to relieve much of the burden of this conversion. It also describes the effects of the conversion on several sample programs.

Finally, debugging tools and language runtimes can perform additional interesting tasks given an existing garbage collection infrastructure. This dissertation outlines several such extensions, and discusses one — memory accounting — in detail.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

PRECISE COLLECTION AND C PROGRAMS

Memory management is one of the most tedious and error-prone tasks in software development. Small, unnoticed memory-management mistakes can cause crashes, security problems, slow degradation of program performance, and OS crashes. While testing catches many of these errors, some remain even in released software. See Figure 1.1 for an example of Apple's Safari web browser afflicted with a slow memory leak.

Reliance on legacy code exacerbates the memory-management problem. Many companies and institutions rely on programs they have used for decades; programs that have been modified by many different hands as managers add new requirements and users find new bugs. Often, the original programmer(s) for the application have moved to other companies, and program maintenance is left to people unfamiliar with the program's design. Worse, documentation on the program's design and implementation is usually either out of date or nonexistent, particularly with regard to memory management conventions. A programmer new to the project may need to spend weeks or months to find and correctly fix memory errors.

Rewriting legacy programs is often impractical or unwise; redeveloping a complex system using modern languages, tools, and designs may take years. Further, the redesign will introduce new bugs to be tracked and fixed. If the original program is critical for the company or institution, spending years to correct one problem — only to potentially introduce different problems — is an expensive risk with minimal hope of reward.

The subject of this dissertation, Magpie, is one solution to this problem. Magpie solves much of the problem of memory-management bugs by modifying a program

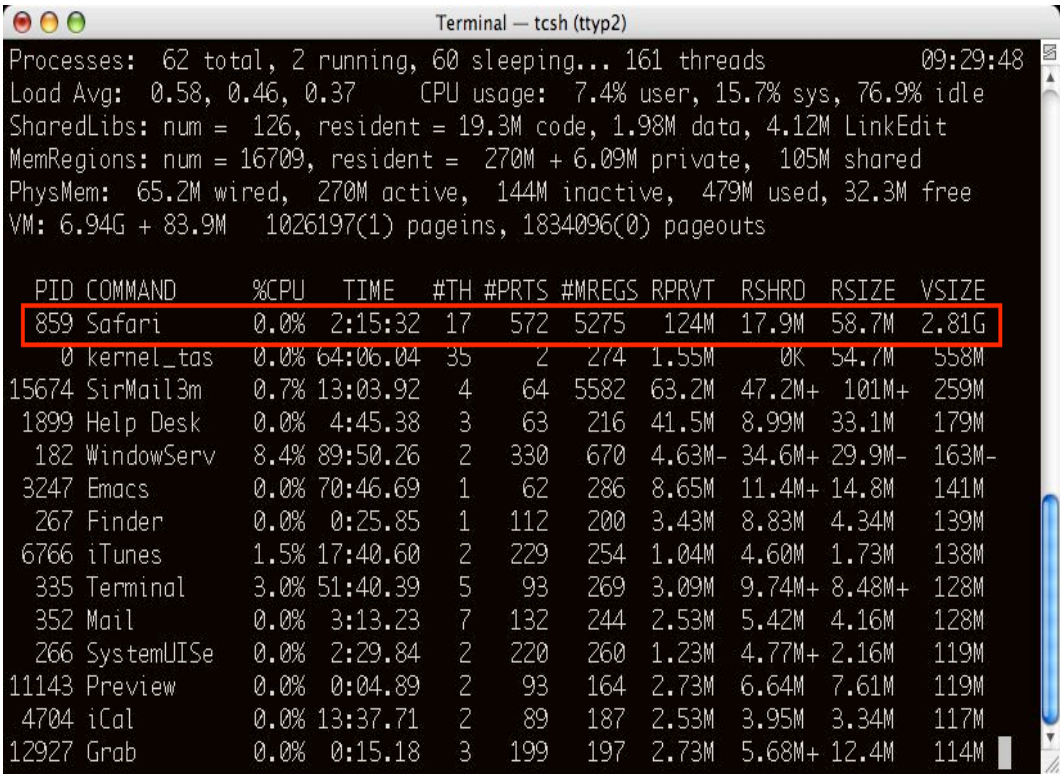

Figure 1.1. A screenshot of Apple's Safari web browser using nearly 3 gigabytes of memory after a couple of hours of normal usage.

to use garbage collection. Thus, the converted program automatically performs its own memory management, rather than relying on the programmer to get everything correct.

> **Thesis:** Precise garbage collection offers advantages to programmers over manual memory management, through ease of programming, a lessening of memory errors, and increased tool support. Furthermore, these advantages are available for typical C-implemented programs with proper tool support. A tool can simplify the process of converting existing code to use precise collection, bringing these advantages to normal C programmers.

Magpie is a tool to demonstrate this thesis. Magpie contrasts with existing tools to aid in detecting memory errors in existing programs. These tools run

the gamut from academic type systems, reworking of language runtimes, dynamic checking tools and complicated software analyses. This chapter continues with a survey of the subject of memory management, and the problems inherent with each memory management strategy, with the conclusion that precise garbage collection is often the best solution. It then outlines the contributions of this dissertation, and concludes with a roadmap for the remainder of the dissertation.

1.1 Memory Management Paradigms

Most programs manage memory using one or more of four basic memory-management strategies: static allocation, manual memory management, reference counting and garbage collection. Each of the basic strategies has advantages and disadvantages with regard to space utilization and performance. Most have tool sets associated with them to aid in their adoption or in their use. This section examines each of these four basic strategies.

1.1.1 Static Allocation / No Deallocation

In some basic programs, very little memory is used; either no memory is allocated, or there is no need to deallocate any memory allocated. Simple student exercises, some simple command-line utilities, and even some more complex utilities (such as compression utilities) may fall under this category. Thus, the memory management strategy is simple: that program declares or allocates what it needs, and then leaves it allocated until program termination.

More generally, applications can use the program call stack as their memory-management device. Although this solution does not scale well in the general case, it is commonly used in practice. Kernel programmers and C++ programmers often allocate as much data on the stack as possible; partly for potential speed gains, and partly for the easy and automatic memory management.

1.1.2 Manual Memory Management

In C, manual memory management is probably the most commonly used memory management strategy. In this case, the programmer explicitly allocates objects

using `malloc` or some similar command, and explicitly deallocates them using `free`. The advantages of manual memory management include predictability with regard to program execution and a fine level of control over object lifetimes.

Unfortunately, this increased level of control leads to increased complexity in nontrivial programs. For example, in a program with several different components accessing a common data structures, it may be extraordinarily difficult for a programmer to determine when the program may safely deallocate an object. Because manually managed memory systems do not export any information about a pointer beyond its existence, programmers must create complicated protocols, add their own metadata, or manually prove when an object can be deallocated.

C compounds this problem by classifying invalid `free`s as actions with undefined effects. A `free` that occurs too early in a program causes no direct error. However, it may cause the program to crash immediately, crash a few function calls later, produce incorrect results, or run fine on most machines most of the time. Many implementations of `libc` display warnings when the program attempts to free an object two or more times, but even these warnings may not occur in some edge cases. Finally, a C compiler obviously cannot produce warnings around suspected memory leaks.

Although errors, warnings and incorrect behavior usually appear before software releases, memory leaks may not. Simple regression tests discover memory leaks only if a programmer thinks to add checks for leaks or the leaks are particularly egregious. When even the simplest of applications may have a lifespan of days or weeks, even a slow leak can cause anger and frustration in users if not caught before deployment.

Identifying and fixing manual memory management bugs has been a focus of research for decades. Work in this area is split between unsound tools that detect possible memory errors, and dialects of C or C++ that greatly restrict the probability of errors occurring.

The former includes mechanisms as simple as a protocol for logging allocations and deallocations and finding mismatches [6], or writing specialized macros that

cause the program to run checks dynamically [40]. More complicated solutions involve program transformations and/or additional compiler annotations to pass more detailed information to dynamic tools [2, 16, 22, 37].

Dynamic tools provide obvious advantages for the programmer. They usually require little work: the programmer recompiles the program, linking it to the tool, and checks for errors. Unfortunately, dynamic tools do not give particularly strong guarantees on their results. A successful run tells the programmer that, on one particular execution, the program did not violate any invariants that the tool happened to check. However, other executions may include memory errors, and errors may have occurred that the tool did not notice. Thus, using these tools reduces to the general problem of complete coverage and complete observation in testing. Both add a considerable burden to the testing process.

Static analyses do not suffer from coverage problems, but have their own disadvantages. Many perform static error checking by attempting to convert existing source code to some safe language. Dhurjati et al., for example, force the use of strong types in C (amongst other restrictions), and then soundly check the code for memory errors. Although their approach works — given their restrictions on the language — it still does not test for some forms of memory errors. For example, their system does not attempt to guarantee that the program will not reference a deallocated object [15].

Another example in this line of research is CCured [36], which includes a tool for automatically converting legacy code in some circumstances. Although compelling, CCured may require programmers to rewrite large sections of their code, or learn a new annotation scheme and determine how to insert it into their code. These limitations are particularly harsh if the programmer is not familiar with the code base being converted.

Finally, safe language tools restrict the kind of C the programmer can write by their very nature, often in many different ways. In some cases, the added restraints are too burdensome, since the programmers have used C because their programs are considerably easier to implement in a low-level, unsafe language. Further, the

added safety checks (bounds checks, for example) may add performance overhead beyond what the programmer thinks is acceptable.

As a last resort, several companies and research groups have encouraged programmers to switch to safe languages with C-like behavior and syntax. C-- and C# [28, 32] are particularly good examples of this trend. These new languages may entice programmers writing new programs, but leave the question of legacy code unanswered. With programs now having large code bases and decades-long lifespans, reimplementation in these languages is often impractical.

1.1.3 Reference Counting

Reference counting manages memory by annotating each object in the system with a count. When some part of the program references the object or some data structure holds the object, the count is incremented. When the object is no longer referenced by the program or is removed from the data structure, the count is decremented. When an object's count equals zero, memory for the object is freed, and objects referenced by that object have their counts decremented.

Reference counting generally requires the programmer to increment and decrement counters programmatically. Some compilers or language constructs perform the reference-count updates automatically, but such systems are rare in general-purpose programming languages. Unfortunately, manually modifying reference counts quickly becomes burdensome for both a programmer writing a program or library and later readers.

Despite its drawbacks, reference counting is a mature technology with a long history, and is used in many major systems. COM [11] uses reference counting, as do Apple's Cocoa and Carbon libraries [34, 44], among a wide variety of other examples. In addition, many varieties of reference counting exist, including standard reference counting [25], single bit reference counting [45], and distributed reference counting [30]. Several improvements over traditional reference count updating strategies have also been created [39].

Programmers familiar with reference counting, however, still describe problems similar to those of manual memory management and garbage collection. Many

libraries must create and guarantee conventions and protocols on their functions, such as what the reference count of a new object is and how particular functions modify reference counts. Even within the program, often it is unclear which functions need to update a reference count and which functions do not. Updating a reference count too often leads to performance degradation, but not updating it often enough can cause memory errors or leaks. The latter problem becomes particularly difficult in multithreaded and event-driven applications in which the object may be referenced in several different concurrent contexts.

Further, unless the reference-counting machinery works in an analogous way to incremental garbage collectors (with the analogous difficulties in implementation), reference counting may still lead to unbounded pause times. If the program decrements an object's reference count to zero, the reference-counting subsystem will decrement the reference counts of any objects it references and then free the object. If the object is the only reference keeping a large structure in memory, this may trigger an unbounded cascade of reference-count decrements and frees.

Finally, reference counting fails when programs make use of cyclic data structures. Such programs must either take additional (and tedious) precautions to ensure that no object is in a cyclic structure before the program loses all references to it, or employ the use of a garbage collector. Hybrid systems are the most common, using reference counting in the general case but occasionally running a simple garbage collection routine to free unreferenced, cyclic structures.

1.1.4 Garbage Collection

Garbage collection shifts the burden of memory management to the language runtime. Doing so eliminates most problems of programmer error in memory management. This results in both fewer memory management errors and increased programmer productivity, because the programmer need not spend time on memory management concerns.

Garbage collection, in the general case, works as follows:

1. Find the *roots* of the system. These include local variables, stack frames and

global variables. They are necessarily still live.

2. Mark the roots as "still live," and add them to a mark queue.

3. For each item in the queue, mark any objects that the item refers to, adding these objects to the queue.

4. Repeat until the queue is empty.

5. Deallocate the space used by any unmarked objects.

The exact mechanisms used for garbage collection vary widely, as do the conditions in which garbage collection is triggered. Some garbage collectors prioritize low space usage [5, 46, 49]. Others work for the fastest wall clock program execution speeds [9, 12, 24]. Still others focus on providing a high degree of program responsiveness [1]. Wilson provides a good survey of the design space [53]. Although a particular program may perform better with one particular garbage collector, in most cases there are a wide variety of collectors that will function with a program without modifying the program source code.

The following terms are used extensively throughout this dissertation:

- *Conservative garbage collection* is a style of garbage collection that functions without exact information about whether or not a given word in the heap or on the stack is a pointer.
- *Precise garbage collection* is a style of garbage collection that requires exact information about whether or not a given word in the heap is a pointer or not. Precise garbage collection is sometimes referred to as *nonconservative garbage collection*, *type-accurate garbage collection* or *accurate garbage collection*. For simplicity, this dissertation uses only "precise" to describe such collectors.
- *In-place garbage collectors* perform garbage collection without moving objects in the heap. In-place collection is required for conservative garbage collectors except in the presence of specialized virtual memory systems [43].

- *Moving garbage collectors* may move objects in the heap during garbage collection. Moving garbage collection is required for most real-time garbage collectors and other collectors designed to lower garbage-collection latency.

Most compiled, garbage-collected languages — such as Java — use precise collectors. Many of these collectors also move objects as necessary, making them precise, moving collectors. On the other hand, the vast majority of garbage collected C programs use conservative, in-place collection, via the Boehm collector [8, 27]. The Boehm collector's advantages include a well-established and mature code base that links somewhat easily into existing C programs.

The Boehm collector is conservative because C does not provide any information about whether a word in the heap is a pointer or not. Although this flexibility is occasionally useful in C programs, most programs do not reuse pointer storage for numbers, or number storage for pointers. Many programmers consider doing so bad style, as it makes the program more difficult to understand.

Although conservative collection links easily into C programs, it does have some disadvantages. Specifically, the conservatism of the collector occasionally causes numerical values to be interpreted as live pointers to otherwise dead objects, causing memory leaks. More significantly, conservative collectors may mistake dead pointers as roots, leaving objects in the heap indefinitely. The leaks caused by these two problems can be anywhere from minor to severe, depending on the program [10]. Further, in-place collection can lead to fragmentation, because the live data in memory cannot be compacted. Although moving, conservative collectors have been described in the literature [43], they have not become popular, nor do they solve the problem of incorrect root identification.

Unlike conservative collectors, precise collectors can move objects in the heap, because there is no chance of the collector mistakenly identifying a number as a pointer and updating it. Further, precise collectors have exact information as to what words in the heap are roots, and thus will never mistakenly consider a nonroot a root. Thus, they do not suffer from leaks caused by mistaken pointers or roots, and they may compact objects to avoid fragmentation.

However, this precision is not without cost. As previously stated, most precise collectors are used in association with a compiler and runtime for a safe language. In these cases, the compiler can emit all the information the collector requires [38]. Even if the compiler emits C code, this information can be added with full knowledge of the original, type safe program [23].

Until Magpie, generating the root and pointer / nonpointer information for arbitrary C and C++ programs required either an extraordinary amount of hand coding on the part of the programmer [26] or the additional design and maintenance of an ad hoc transformer [20].

1.2 Contributions

The contributions of this dissertation are as follows:

- The design and implementation of a tool for converting arbitrary C code to use precise garbage collection, without specifying the compiler or a particular garbage collection style. Previous work focuses only on conservative collection for C, or for performing precise garbage collection on the limited subset of C generated by a specific compiler.
- The design, implementation and evaluation of a set of analyses designed to limit the amount of programmer effort required for the transformation. Previous work required the programmer to write their code in a very specific style, to add in a considerable number of annotations or library calls, or to perform all the transformations contained in Magpie manually.
- An experience report on using Magpie to convert existing programs to use precise garbage collection, including measuring the effects of this transformation in time and space.

Secondarily, this dissertation reports on one example of using the infrastructure of garbage collection for another, useful purpose: memory accounting.

I intend this work to be evaluated based on four metrics:

- *Applicability*: The range of correct C programs that Magpie handles. Syntactically, Magpie handles most C. Further, while Magpie imposes restrictions

on some patterns used in C programs, it handles most C programs I have tried it on.

- *Ease of use*: The amount of programmer effort required to convert a program using Magpie. In most cases, Magpie requires little to no effort by the programmer. In fact, it is frequently easier to use than the Boehm collector.
- *Efficiency in time*: In the common case, the impact of Magpie on performance. Benchmarking results show that, in most cases, the performance of a Magpie-converted program is within 20% (faster or slower) than the original.
- *Efficiency in space*: How well Magpie-converted programs track the space usage of the original program. Benchmarking suggests that Magpie-converted programs will use more space than the original (generally, less than 100% overhead on the benchmarks tested), but track the usage of the original.

1.3 Compilers and Garbage Collection

There has been considerable previous work on using compiler analyses to increase the performance of an existing garbage collector. In contrast, Magpie gathers the information required to perform garbage collection. In the future, Magpie could, in addition to the analyses and conversions described in this dissertation, also implement these optimizations to generate faster or more space efficient programs.

For example, Magpie could be modified to inline allocations into the converted code, rather than translating existing allocations to use the garbage collector's interface. Many compilers for garbage-collected languages perform this optimization, particularly compilers for functional languages. While Magpie does not support such an ability directly, the implementer of a garbage collector for Magpie could implement some of this functionality. Magpie converts all allocations in the program to a limited set of allocations functions in the garbage collector. A collector author could thus define these symbols as macros, rather than functions, and the final C compiler would inline them into the program.

However, such a solution would not be able to use any information gathered in Magpie. Magpie could gather more information about variable and object types,

and use this information to allow faster garbage collection. The TIL [50] compiler for ML, for example, uses type information to improve the performance of the garbage collector. Again, the purpose of Magpie is to convert C to use precise garbage collection; once this conversion exists, additional literature on compiling garbage-collected languages may be applied.

1.4 Roadmap

This dissertation is divided into seven chapters. Chapter 2 describes the problems in adding support for precise garbage collection to C code, and Chapter 3 gives an example of converting a standard utility program. The latter chapter serves as a guide for the rest of the thesis, but may be useful on its own to Magpie users.

Chapter 4 provides technical information on the analyses, conversions and techniques used in Magpie. Those interesting in learning about the internal structure of Magpie or extending Magpie may find this chapter most useful.

Chapter 5 presents the benefits and costs of conversion, in terms of time spent converting the program, the memory use of converted programs, and the efficiency of converted programs.

Chapter 6 explores one way in which the infrastructure of precise garbage collection can be used to provide other, useful functionality. Specifically, it describes a memory accounting system that allows programs to query and limit the memory use of their subthreads.

Finally, Chapter 7 concludes this dissertation. It reiterates the contributions of this work, discusses additional situations in which Magpie could be used, and discusses several areas of future work.

CHAPTER 2

THE HIGH LEVEL DESIGN OF MAGPIE

This chapter describes the high-level design of Magpie, including the goals of Magpie, what is required for precise garbage collection, a basic idea of how Magpie satisfies these requirements, and some cases Magpie does not yet handle.

2.1 Goals

Magpie serves as a bridge between C and C++ and precise garbage collection. It works by gathering sufficient information for the collector via static analyses and queries to the programmer. This information is transferred to the runtime by translating the original source code.

Magpie primarily targets preexisting C applications, although with some additional work, it could handle applications in development. As stated previously, corporations and institutions rely on programs written many years ago by programmers who have moved on to other things. These programs may have existed sufficiently long that they contain code to handle situations that no longer exist, and have been written and modified by many different programmers. For a programmer new to the project, finding and fixing memory problems is a daunting task.

Finally, the goal of Magpie is to handle the largest subset of C possible without tying Magpie to a particular compiler or garbage collector. As of the writing of this dissertation, Magpie handles most programs I have tried it on. The exceptions include only those programs where the programmer played strange games with pointers.

Because I strove to make Magpie compiler-independent, it functions by taking C source as input and generating translated C source. Although translating C to C increases the complexity of Magpie, linking Magpie to a particular compiler is a

considerable barrier to adoption. Further, linking Magpie to a particular compiler increases Magpie's maintenance burden, because even patch level updates to the original compiler may change internal forms and data placement.

2.2 The Mechanics of Garbage Collection

Although the exact implementation of a garbage collector may vary greatly, all garbage collectors require certain information about a program to function. In particular, garbage collectors require information on the following three subjects:

- *Which words in the heap are root references.* Obviously, to perform the first step of garbage collection, the garbage collector must know which objects in the heap are roots. Typically, systems inform the garbage collector of particular pointers in memory that should be used as root references.
- *Where references exist in each kind of object.* To propagate marks, the garbage collector must know where the references are in every kind of object extant in the heap. Typically, garbage collectors use *traversal routines* (also known as *traversers*) rather than mappings on the memory an object uses. Traversers allow more flexibility in the layout of objects while presenting a simple interface to the collector.
- *What kind of object each object in the heap is.* Finally, to propagate marks, the garbage collector must have a way to map an object to its kind. Typically, garbage collectors create this mapping with a *tag* that is associated with the object upon allocation. Some collectors associate the tag directly with the object, whereas others sort objects into large blocks and then tag the blocks.

With this information, basic garbage collection is straightforward. The collector iterates through the list of root references, marking the referenced objects as it goes. For each object in the set of marked objects, propagation works by looking up what kind of object that object is and then invoking the appropriate traversal function on it. When a fixpoint is reached — no more objects have been added to the set of marked objects — any unmarked objects are deallocated.

Specific garbage collectors implement this routine in different ways. Incremental and real time collectors break this process up into discrete, time-limited chunks, and allow the main program (also known as the *mutator*) to execute even as the collector runs. Generational garbage collectors work by only running this routine over subparts of the heap during most collections. Copying collectors mark an object by copying it into new heap space, and then deallocate unmarked objects by deallocating the entire previous heap space. Other garbage collectors modify the routine in other ways.

2.2.1 The Design of Magpie

To add support for garbage collection to C, Magpie must satisfy the three requirements described previously. Precise garbage collection requires that this information not include any conservatism; a tag must state that the object is of kind k, not that it may be of kind k. Equally importantly, a traversal function for the mark-propagation phase must identify exactly those words in an object that are references to other objects.

Figure 2.1 shows the high-level design of Magpie. Magpie uses a five-pass system for inferring the necessary information, generating code, and optimizing the output. Information is transferred from pass to pass through a persistent data store. The five passes of Magpie are as follows:

- *Allocation Analysis.* The allocation analysis determines what kind of object each allocation point creates. Magpie uses this information to tag allocated objects as having a particular type.
- *Structure Analysis.* The structure analysis determines which words in an object kind are pointers. Magpie uses this information to generate traversal functions.
- *Call Graph Analysis.* The call graph analysis generates a conservative approximation of what functions each function calls. This analysis is only used for an optimization on the final, translated source code, and may be skipped if optimizations are disabled.

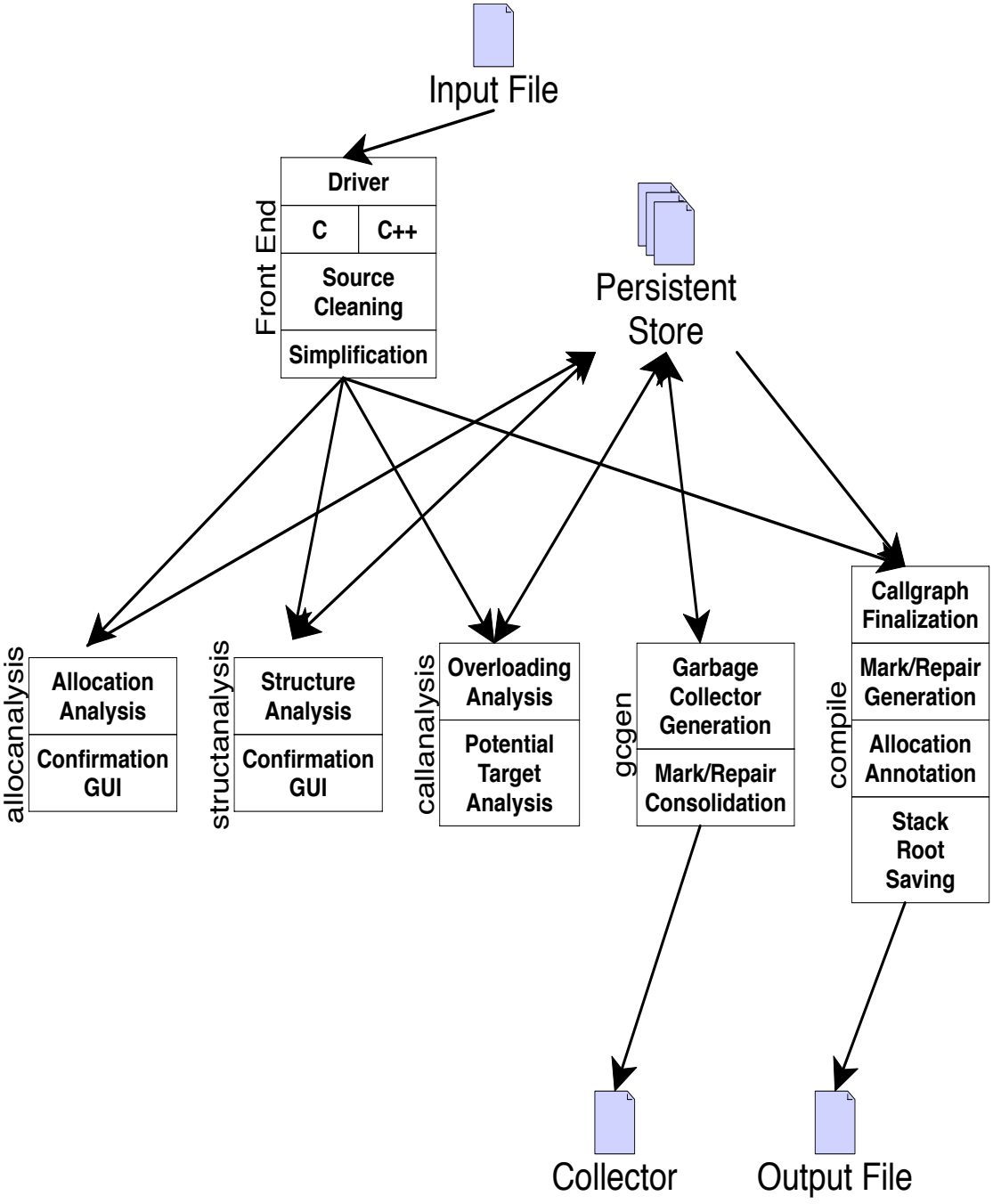

Figure 2.1. The high-level design of the Magpie toolset.

- *Garbage Collector Generation.* Although Magpie currently exports only one garbage collector, it does allow programmers to tune the collector in some ways. This pass generates a tuned collector for use with the converted program.
- *Conversion.* This final pass performs the final conversion, generating a new C file for each C file in the program. It makes several modifications to the program, including annotating roots, annotating the calls with tags, generating traversal functions, and associating each traversal function with the appropriate tag.

One goal for Magpie is support for moving collectors. Moving collectors require the program to pass additional information to the collector so that the collector can repair references to an object should that object move. For the most part, this requires only an additional traversal function: the original traversal function marks the objects an object references, whereas this additional traversal function updates references in the object should the collector move any of the referenced objects.

However, moving collectors require Magpie to handle roots in a particular way. Translating C to use nonmoving collectors might allow Magpie to annotate roots by directly marking an object as a root, rather than by annotating the references. More concretely, a nonmoving system may either pass information about roots by passing the address of the root reference, or by passing the object itself. In a moving system, the latter option is not available. Because an object may move, the collector must know the actual word in memory of the root reference, because it may need to repair the reference. Implementing this requirement is a significant part of the final conversion phase.

2.2.2 Dealing with Libraries

Most libraries work with Magpie without any problems. However, some libraries may cause problems for Magpie by saving references to garbage collected objects.

All other libraries — including libraries that save non-pointer data or use callbacks[1] — should work without problems.

In situations in which libraries save pointers, the pointers may become corrupt if a garbage collection occurs while the library holds them. Essentially, such libraries hide roots from Magpie. This hiding does not allow the runtime to mark the objects pointed to or repair the hidden root pointer.

If the program being converted links to libraries in this class, the programmer has three options:

- Convert the library using Magpie.
- Use an annotation — `__saves_pointers__` — in the library headers for those functions that save pointers.
- Because the library headers may not be editable by the programmer, Magpie allows the programmer to wrap a call with the `__force_immobility__` construct. This construct functions as an expression, which tells Magpie to force immobility in any pointers found in the expression. This option likely requires more typing than the previous option, because a library function is declared once but may be called often, but is necessary if the library headers cannot be modified.

Obviously, the first option is highly recommended, but may not be feasible. Forcing immobility impairs the ability of the garbage collector, and may create serious memory leaks. Because immobility is only used in cases where there is a pointer outside the "view" of the collector, the collector must keep the object alive forever, as it will be unable to prove that all references to it have been dropped. Thus, if the program calls such functions frequently, a serious memory leak may arise. Converting the library will solve this problem.

---

[1] Additional care must be taken in the case of callbacks. Many libraries that make use of callbacks allow the programmer to attach arbitrary data to a callback, which is then passed to the callback on execution. If this is the case with the library in question, and the data passed is a pointer, that pointer should be considered an internally-held pointer.

2.2.3 In-Source Flags

In many cases, Magpie requires no modification of the program source. In some cases, however, modifying the program is necessary or helpful. For example, if a single-word union field should always be considered a nonpointer, whether or not it has a pointer option, then adding an in-source hint to Magpie may reduce the amount of time and effort spent in the allocation or structure analyses.

Magpie accepts four in-source hints. All the flags, except the last, should be treated as "storage class specifiers" in the C/C++ grammar (e.g., `static` or `register`), and may be placed before any field or variable declaration. The flags are fairly self-explanatory:

- `__preempt_3mify_noptr__`: Regardless of any other hints or information, Magpie should treat this item as a nonpointer.
- `__preempt_3mify_ptr__`: Regardless of any other hints or information, Magpie should treat this item as a pointer.
- `__saves_pointers__`: The given function saves pointers within the library.
- `__force_immobility__(exp)`: Forces immobility for any objects referenced by pointer in the expression *exp*.

As an aside, "3mify" is due to a historical name for Magpie, and will likely be changed to "magpie" in the future.

If the programmer wishes to compile both converted and unconverted versions of their program, these items can be removed via standard C/C++ preprocessor commands.

2.3 Limitations of Magpie vs. Boehm

Both Magpie and existing conservative collection technology operate on a subset of correct C programs. However, these subsets are different. This section begins with a discussion of the limitations of Magpie, and then compares the differences between the two systems.

2.3.1 Limitations of Magpie

First, while Magpie can parse most syntactically correct C programs, it will fail in certain, limited cases. As currently implemented, Magpie contains both a C and C++ front end, both of which correctly parse a subset of their respective languages. For C programs, in most cases, the existing C parser will work. Unfortunately, the C parser will fail in some cases in which a lexical symbol is used as both a type and an identifier. However, the C++ front end handles these cases, and Magpie includes an option allowing users to parse C code using the C++ front end, while still generating syntactically correct C in the back end. However, as previously noted, the C++ front end also handles only a syntactic subset of the language. Specifically, it will fail when programs use function type casts without a `typedef`.[2]

Thus, Magpie can handle all syntactic C programs except those that contain examples of both cases within the same file. In the rare case that this does occur, the second problem (function type casts) is easily solved with the use of an additional `typedef`.

Semantically, Magpie contains neither support for C++ nor support for multithreaded programming. Internally, Magpie contains a parser, several data structures, and several analyses to handle C++, but support for C++ was dropped due to time constraints. Completing the work would require considerable additional technical work, but no additional research insights.

In contrast, simple extensions for multithreaded programs would be easy to add, but would most likely have excessive locking costs. In general, the problem of adding more efficient support for concurrent programs simplifies to the general problem of adding minimal locking, which is an unsolved problem. See Section 4.8 for more information.

Finally, Magpie cannot handle programs with implicit types. Since Magpie bases its analyses on the structure and union declarations within the program, it cannot handle cases in which important structure and union declarations are left out of the program source. For example, a program that allocates blocks of memory and

[2]For example, `(void (*)(int))my_var`.

then interacts with these blocks using pointer arithmetic — as opposed to programs that use structure declarations and field accessors — will fail with Magpie. More commonly, C programs that use implicit unions will fail; if a program uses fields or local variables to store both pointer and nonpointer values, but does not declare the fields or variables as a union between these types, the Magpie conversion will fail.

2.3.2 Comparisons to Boehm

The Boehm conservative garbage collector is designed as a completely separate subsystem from the original program. In the ideal case, this separation of concerns is quite clear: all the Boehm collector requires of the programmer is linking their program with the Boehm collector. In some cases, additional work may be required to identify global and static variables as garbage collection roots. This separation allows Boehm to function in many cases where Magpie would not; the Boehm collector will not fail based on the program source, and will not fail to implicit types. Further, the Boehm collector has been extended to handle multithreaded programs and C++.

However, this separation limits the applicability of the Boehm collector in some circumstances. These include cases in which the program obfuscates roots or references from the collector. For example, a program could write objects to disk or mask pointers using some cryptographic key. Programmers concerned with space efficiency may also perform a simple optimization on sparse arrays: a function allocates the space needed for the important part of the array, but returns a pointer to where the complete array would have begun. Since the design of the Boehm collector segregates it from the original program, analyses and programmer annotations cannot be used to transmit information about these obfuscated roots or references to the collector.

In contrast, the design of Magpie tightly couples the conversion with the original program and involves the programmer in the conversion process. This allows the programmer to identify and intervene in cases in which pointers are obfuscated from the collector. In the case of a program writing objects to disk, the programmer

could treat the subsystem that performs the write as a library that saves pointers. The mechanism to handle this case is discussed in Section 4.6. As of the current implementation, Magpie can also handle obfuscated references on the heap (e.g., encrypted pointers or pointers placed before or after their associated objects), by having the programmer write code to translate these references for the garbage collector. However, Magpie cannot handle obfuscated pointers on the C stack.

A system combining both the conservative approximations of the Boehm collector and the interactive, program-specific conversions of Magpie would extend the domain of both systems. Conservative garbage collection allows for implicit structure and union declarations, while the interactivity of Magpie would allow for more exact analysis of roots, additional information for the conservative collector, and the handling of some types of obfuscated pointers and roots.

CHAPTER 3

USING MAGPIE

This chapter describes the use of Magpie on C and C++ programs. For clarity, this chapter uses the `UNIX` utility `top` as a running example. The specific version of `top` used is Apple's `top` for Darwin 8.3 (OS/X 10.4.3), and can be found at the Apple Developer Connection.[1]

The conversion of a C program to a precisely collected C program requires the following steps:

1. Generating the input to the system
2. Allocation analysis
3. Structure analysis
4. Callgraph analysis
5. Garbage collector generation
6. File conversion
7. Compilation

This chapter discusses all these steps in more detail, including the requirements for each step. Although some steps may be performed out of order, the given order is strongly recommended. For completeness, steps #1 and #2 must be completed for every file in the system before beginning step #3. Steps #3 and #4 or steps #4 and #5 may be interleaved; Magpie requires only that step #3 be performed

[1] http://developer.apple.com

before step #5. Finally, steps #6 and #7 may be interleaved at the program level, although, for any particular file, step #6 must be performed before #7.

3.1 Generating the Input

As previously stated, Magpie is designed to handle both C and parts of C++. The C system is far more mature and has been used to handle a wide variety of programs, including simple UNIX utilities, large applications and Linux kernel drivers. The C++ extensions are far more limited and far less well tested, and are included largely as a proof of concept.

For simplicity, Magpie only handles preprocessed source code. Although Magpie accepts command line arguments that allow it to invoke the C/C++ preprocessor itself, in most cases it is simpler to generate the preprocessed source once, and save the result to disk. Generating preprocessed source is simple, and merely requires minor changes in the Makefiles of the original source.

In the case of top, the generation of the preprocessed source code was simple, requiring only the addition of a single command. The original, relevant lines of the Makefile are as follows:

```
$(OBJROOT)/%.o : $(SRCROOT)/%.c ...
        $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
```

Generating preprocessed source during the build process requires only a change to the following:

```
$(OBJROOT)/%.o : $(SRCROOT)/%.c ...
        $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
        $(CC) -E $(ECPPFLAGS) $(ECFLAGS) $< > ../topsrc/$<
```

In this example, the additional line preprocesses the original source using the GCC argument "-E", and saves it to the directory *topsrc* using standard UNIX redirection.

The use of a single, separate directory for the preprocessed source is simply a convenience. Others methodologies may be used if they work better for a particular

program. Further, Magpie does not require the preprocessed file to have the same name as the original source file. Again, doing so is simply a convenience.

3.2 Allocation Analysis

The allocation analysis attempts to determine, for each allocation point in the program, what kind of object is allocated at that point. The user converting the program may choose to do this analysis one file at a time, or all at one time.

At this point in the process, the user must create or choose some directory in which Magpie will store information for use by later passes. Although the user may choose an existing directory, the `top` example uses a separate directory, `topinfo`. At this point in the `top` example, we have a base directory (containing the original `top` source), plus two additional directories: `topsrc`, containing the preprocessed source, and `topinfo`, which contains any information generated by the individual analyses.

Invoking the allocation analysis requires the following command:

```
magpie allocanalysis --info-dir topinfo topsrc/*.c
```

If the invocation of the allocation analysis occurs in a directory other than the directory where the preprocessed files were created, the additional `--base-dir` flag is *required*.

Further, if the program uses any allocators other than the standard C allocators (`malloc`, `calloc`, etc.), the names of all the allocators used in the program must be passed via the `--allocators` flag. This flag can be useful not only in cases where different allocating functions are used, but also in situations where the program places a facade over the system allocator for portability reasons. Most benchmarks in the SPEC benchmark suite, for example, use internal allocator facades over *malloc* and *calloc*.

The `--allocators` flag accepts both singular names and equivalences as arguments, to correctly match an unknown allocator with its interface. The standard allocators (*malloc*, *calloc*, *realloc* and *kmalloc*) may be given simply as their names,

and Magpie will correctly determine the interface. For example, the command line argument `--allocators malloc,calloc` informs Magpie that the program uses two allocators, *malloc* and *calloc*, which have their default implementations.

When another name is given without an equivalence, it is assumed that the allocator has the same interface as *malloc*. For example, the command line argument `--allocators malloc,xmalloc` informs Magpie that the program uses two allocators, *malloc* and *xmalloc*, both with the calling interface of *malloc*. In cases where the new allocator has a calling convention of another built-in allocator, an equivalence may be used. For example the following:

```
--allocators xmalloc,xcalloc=calloc,xrealloc=realloc
```

Defines three allocators: *xmalloc*, *xcalloc* and *xrealloc*. These allocators have the calling interface of *malloc*, *calloc* and *realloc*, respectively

After parsing each file, the analysis engine attempts to determine whether a file performs any allocations. If it does not (as is the case in several of the files comprising `top`), the allocator simply notes this in its information database and informs the user as follows:

```
No allocations found in this file (topsrc/ch.c).
```

If allocations are found, Magpie analyzes the code surrounding the allocation site, to make its best guess as to what kind of object is allocated at that site. Then it creates a window, requiring the user to make the final determination for each particular allocation site. In most cases, the analysis correctly determines what is allocated at each allocation site, but user intervention may be required in some cases.

Figure 3.1 shows the window for the file `libtop.c`. The window shows the number of allocations found in the file (and the user's progress through them), the file name, the line on which the allocation site was found, the name of the function the allocation is in, and a syntax-highlighted display of the allocation site and the code around it.

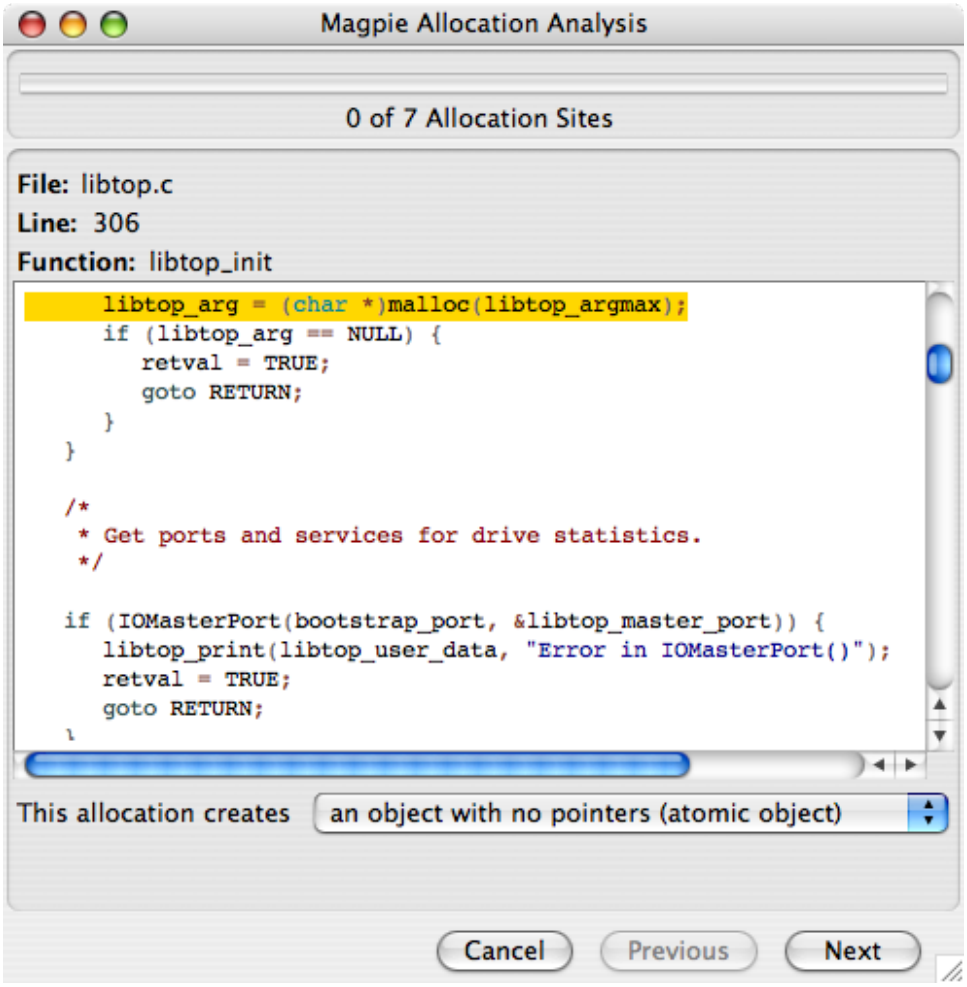

Figure 3.1. The allocation analysis window for the `top` source file `libtop.c`.

At this point, the user must decide if the analysis determined the correct kind for the allocation or not. If it has, she may simply move on. Otherwise, she must select a different options. The options are as follows:

- *An object outside the collected heap*: In this case, the object allocated at this allocation site should be allocated outside the garbage collected heap. The collector will never move nor collect these objects.
- *An object with no pointers*: In this case, the object allocated at this allocation site has no pointers to any other object in the heap. For example, a heap-allocated string satisfies this requirement, as does a heap-allocated array of

numbers. The allocation in Figure 3.1 allocates a string, so this is the correct option for that allocation.

- *A tagged object*: A *tagged object* is a structure allocated to the heap. The division between the previous option and this option depends on the user's preferences. Some users may wish all structures to be considered tagged objects, even if they contain no pointers, or they may only wish to use the tagged object form for structures that contain at least one pointer. The disadvantage of the former approach is a possible increase in code size and fragmentation, as it may cause Magpie to generate additional traversal functions or unnecessarily segregate objects in the heap.

  In the case of tagged objects, the user must specify the structure being allocated. Figure 3.2 shows the second allocation site in `libtop.c`, which allocates a structure of type `libtop_pinfo_t`.
- *An array of pointers*: In this case, the program allocates an array of pointers to other objects. Magpie does not require information about the type of the referenced objects.
- *An array of tagged objects*: In some cases, the program allocates an array of structures, rather than an array of pointers to structures (the more common case). For example, in Figure 3.2, if `calloc` had taken some other first argument than 1, the program would allocate that many contiguous `libtop_pinfo_t` structures. In these cases, the analysis will ask for the type of the object.

In most cases, the analysis chooses the correct values for the allocation. In some cases, however, Magpie will guess incorrectly, and the programmer must choose the correct item from the available options. Every time the user selects "Next", Magpie saves all the information it has to a per-project, persistent file. The location of this and all other persistent information files is given by the `--info-dir` argument to magpie.

Magpie uses information gathered from this analysis throughout the rest of the system, so it is important that Magpie users take time to ensure that their

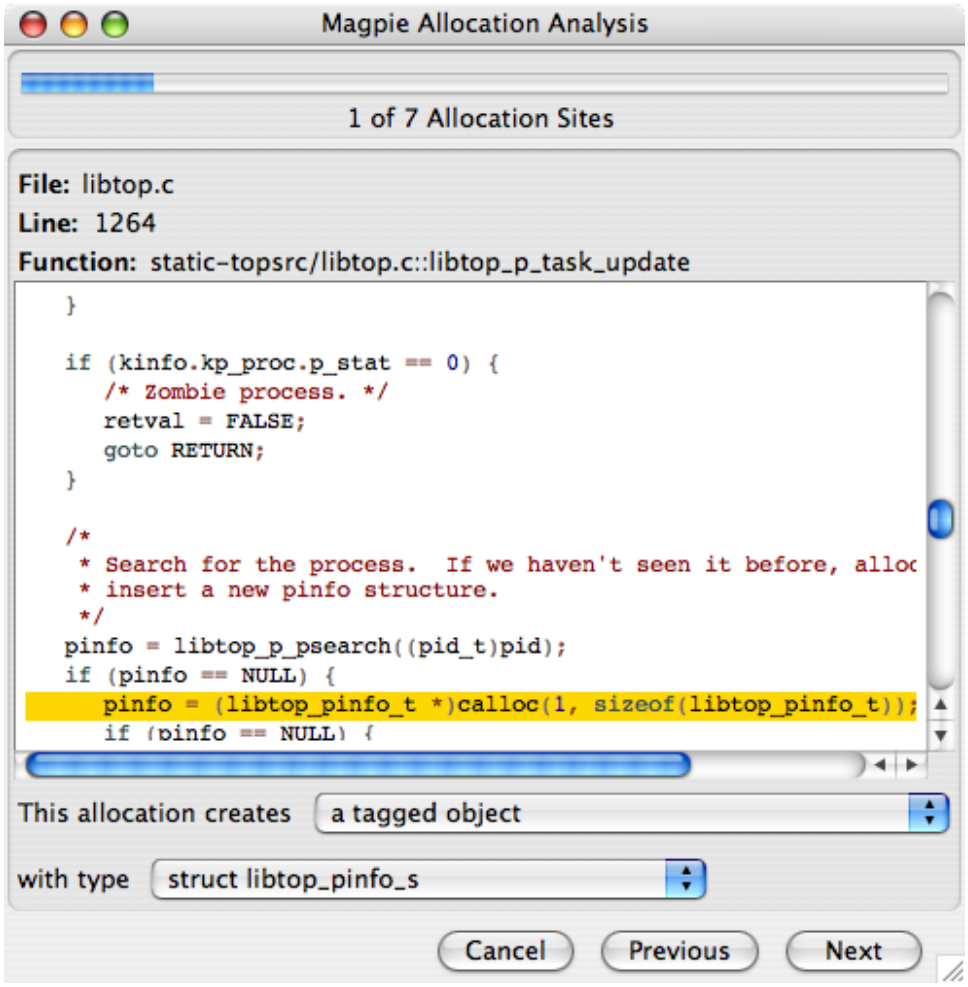

Figure 3.2. An example of the allocation analysis window where the object in question is a tagged object.

analysis results are correct. For example, the structure analysis uses the gathered information to determine which structures to analyze, because the structure analysis need only deal with structures allocated onto the garbage collected heap. Magpie does, however, include a command-line flag — `--auto-run` — for cases in which the user knows the analysis is correct for all items.[2] This flag causes Magpie to run automatically at the command line, without querying the user or bringing up

[2]I have also used this flag as an initial, optimistic, attempt, often saving me the few minutes spent examining the allocation sites.

a GUI.

As stated previously, the user must finish the allocation analysis for every file in the system before moving to the next stage in the process.

3.3 Structure Analysis

The structure analysis attempts to determine the shape of all objects in the garbage collected heap, so that the garbage collector can walk the structure correctly during collection. Thus, for every word within an object, the analysis must determine if the word represents a pointer or some other datum. Again, the user may decide to do this analysis one file at a time or all at once, and again, this step is an interactive analysis unless the `--auto-run` flag is used.

To perform this analysis for a file, the user must first run the allocation analysis on the file. Magpie uses the allocation analysis information, which contains information as to what structures are placed in the heap, to limit the information required from the structure analysis and user.

Invoking the structure analysis requires the following command:

```
magpie structanalysis --info-dir topinfo topsrc/*.c
```

Note that the structure analysis must take the same info directory; in fact, this info directory is used in all subsequent steps. Again, in most cases, the analysis will correctly identify the structure components, but in some cases the developer will need to override the selected option.

For some files, no structures need be analyzed by the system. In these cases, Magpie informs the user that it does not need to do anything with the given file. In cases where there is work to do for the file, the structure analysis creates a GUI. Figure 3.3 shows the initial GUI for the file `libtop.c`.

The structure analysis has many more potential answers than the allocation analysis. Although some answers are not available in some cases, the following are all the potential options:

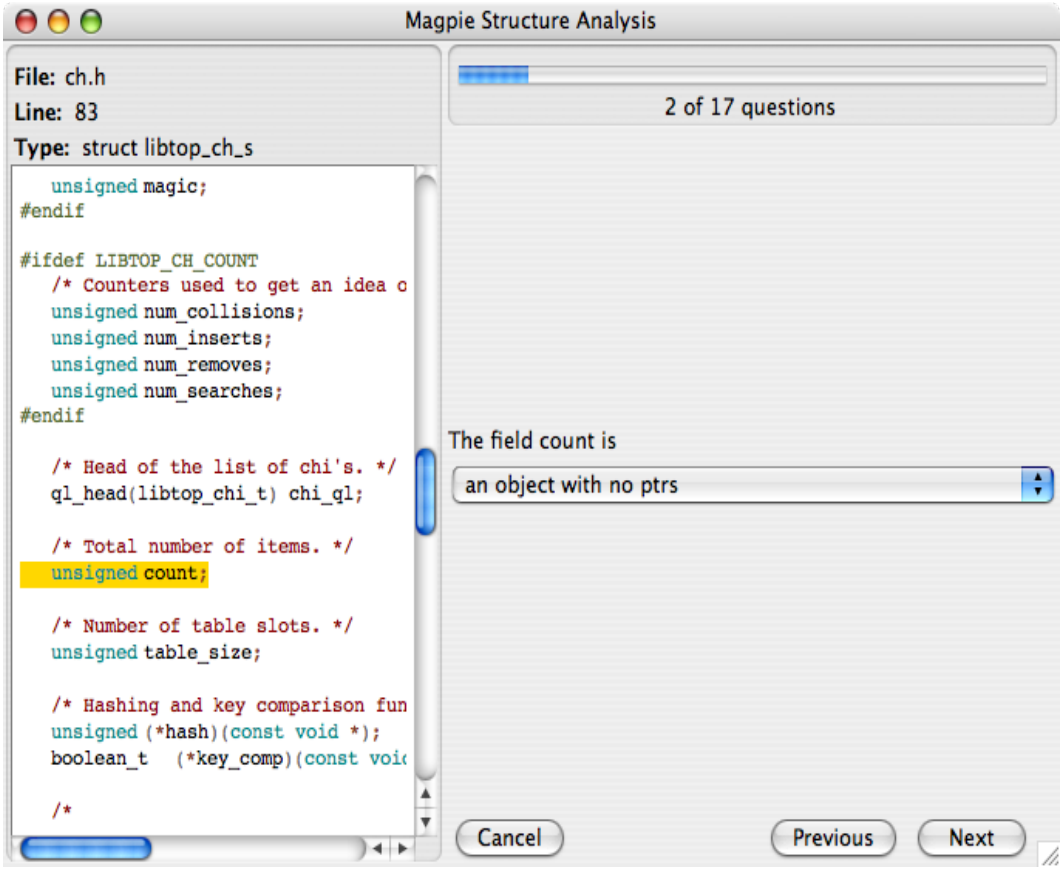

Figure 3.3. The structure analysis window for the `top` source file `libtop.c`.

- *An object with no ptrs*: This type or structure / union field contains a number, set of bit fields, set of characters, or other datum that does not contain any pointers. Figure 3.3 shows an example of this case.
- *A pointer*: This type or field is a simple pointer to some other object.
- *An array of pointers*: This option is only available for types (as defined with `typedef`), not structure fields. In this case, the type is an array of pointers to other objects. This option may also be used for multidimensional arrays of pointers and arrays of structures that contain only pointers.
- *An array of pointers, of size*: This type or field is an array of pointers with a fixed size. The GUI includes space for the developer to provide the size. The

size may be given as a simple number or as an arbitrary C expression that produces the proper number.

- *An array of pointers, lasting to the end of the object*: This field, which must be the last field of the structure, is an array of pointers that lasts to the end of the object. This is useful in cases where the size of the last field is dynamic.
- *An array of inline objects*: This option is available only for types (as defined with `typedef`), not structure fields. In this case, the type is an array of some structure. This case is used when a type defines an array of some structure, not when it defines an array of pointers to some structure type; Figure 3.4 shows the difference between the two cases. In this case, the user interface will ask the user to choose which type is in the array.
- *An array of inline objects, of size*: As the previous item, but the field in question has a fixed size. Note that the size is given in array elements; so an object of eight elements, each of which is 50 bytes in size, should have eight as its size. Although `top` does not contain any examples of this case, Figure 3.5 shows the user interface structure for this kind of object. First, the user selects the "an array of inline objects, of size" option, and then she enters in the size of the field and the element type.
- *An array of inline objects, lasting until the end of the object*: The analogous option to "An array of pointers, lasting to the end of the object." Again, see Figure 3.4 for an example distinguishing arrays of objects from arrays of pointers to objects.
- *Something I need to write my own mark code for*: In some cases, the standard analyses and traversal generators will not work for an object. These cases include situations in which a field's type may depend on some outside state

(a) `typedef␣struct␣foo␣array_of_foo[5]`
(b) `typedef␣struct␣foo␣*array_of_pointers[5]`

Figure 3.4. The difference between (a) an array of `struct foo`s and (b) an array of pointers to `struct foo`s. The latter case is considerably more common in practice.

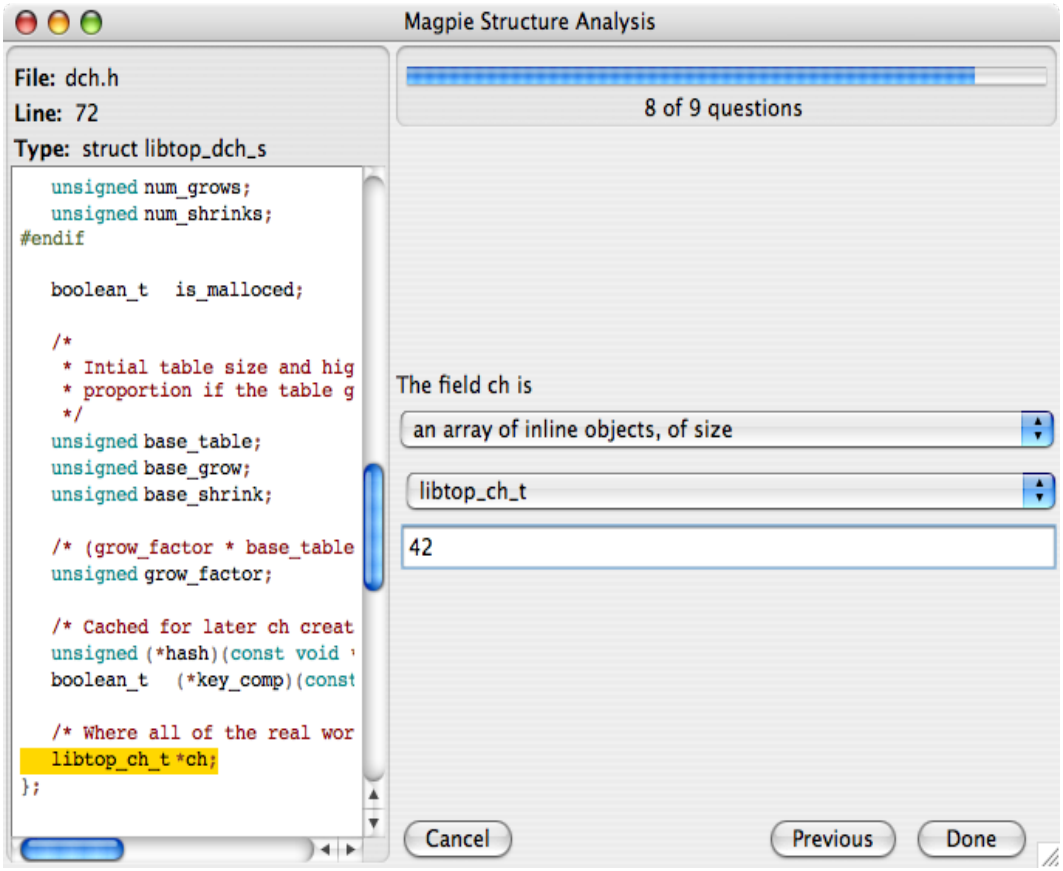

Figure 3.5. An example of entering in the information for the "an array of inline objects, of size" case. Note that the field in question does not, in fact, declare such a thing; this figure is merely an example of the information needed in these cases.

in the program, or situations in which the programmer has obfuscated the pointer. In these cases, the programmer must write her own mark routine for the field or fields. After completing the analysis for all the other fields in the object, the GUI requests the user fill in the information for the fields marked with this option. Figure 3.6 shows an example of this case.

In the case of `top`, the structure analysis requires determinations on 111 fields in 2 files. In all cases, the analyses correctly identifies the kind of the field, and the total time spent in this phase (not counting parsing and loading) was less than five minutes.

Magpie uses the information gathered in the structure analysis in the garbage

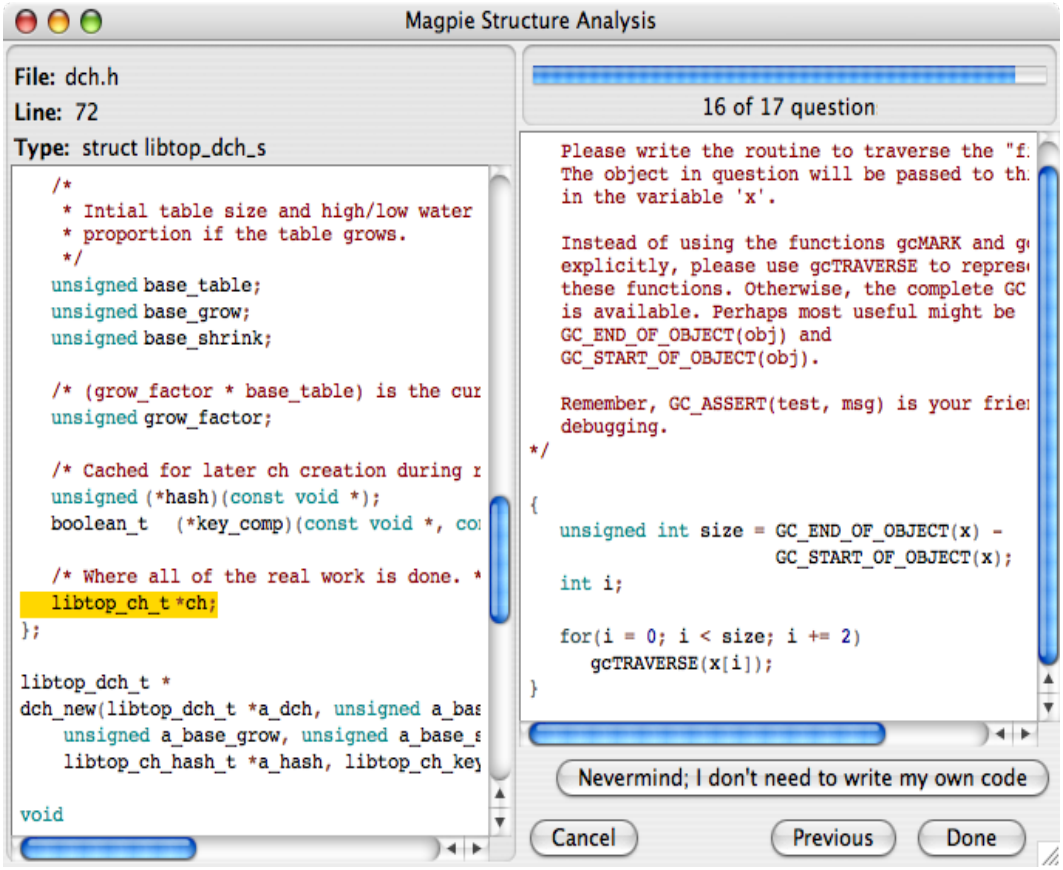

Figure 3.6. An example of entering in a custom traversal function. Again, this is a fictitious example; there is no reason to write a traverser for this field.

collector generation phase (Section 3.5) and the conversion phase (Section 3.6).

3.3.1 Handling Unions

The `top` utility does not use any unions in its execution, but many programs do. When the structure analysis discovers a union — either as an allocated type or as part of one — it first gathers information from the user about each case of the union. Sometimes Magpie can determine that all the cases have the same shape, so the collector can handle the union by always using the first union case. Similar shapes appear most frequently in unions of nonpointer types, where the collector regards the entire union as atomic.

If Magpie cannot conclude that all the union cases have the same shape, it

must determine how to inform the collector which union case is in effect at every collection point. Magpie can handle unions in one of two ways, and asks the user to select one of them.

The first, and preferable, way is to have the user write code to select which union case is in effect. Figure 3.7 shows an example of a user writing code for the `197.parser` benchmark. Doing so is feasible if the structure contains some field that determines which union case is in effect or if this information can be inferred from global program state. Having the user write their own code generates faster code than the alternatives, and is less subject to edge cases where the alternative will not work. Unfortunately, not all programs make distinguishing information available to the programmer. Further, when working with legacy code, the programmer may

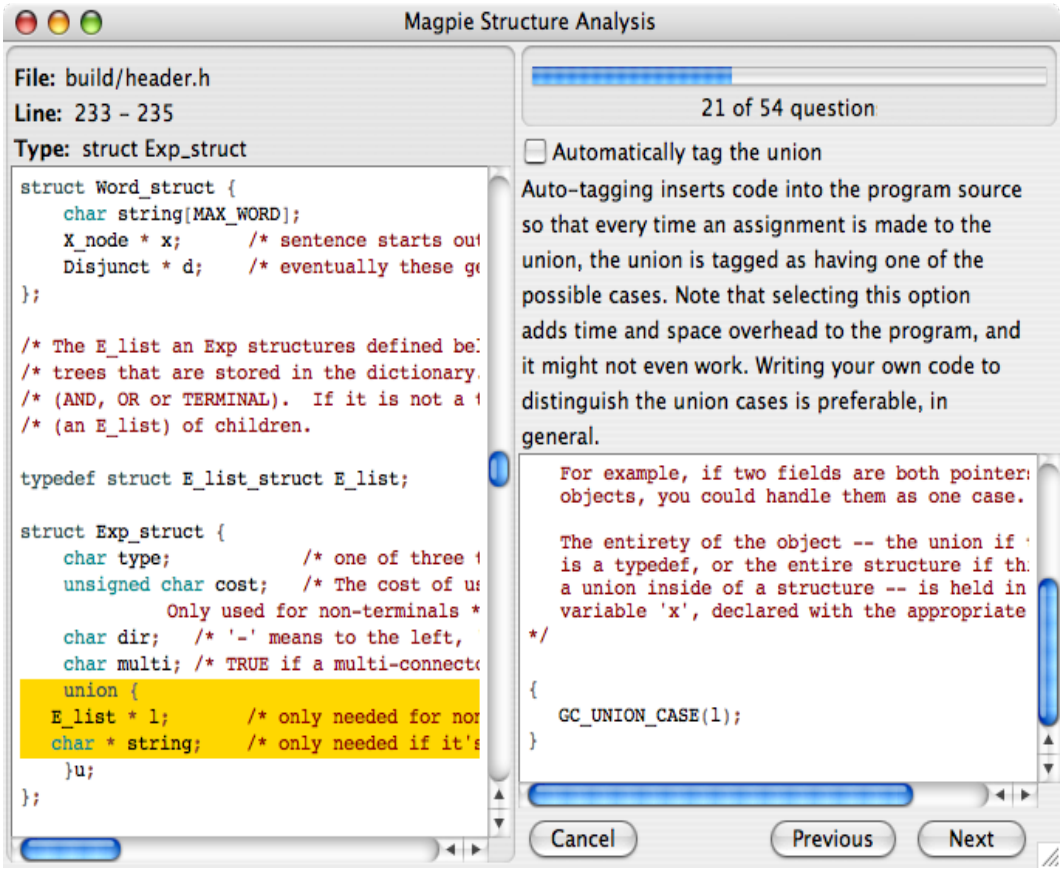

Figure 3.7. The structure analysis GUI for unions

not be aware of this information even if it is available.

In these latter cases, Magpie can handle unions via *autotagging*. Magpie's autotagging automatically inserts code to inform the collector that an object currently uses a particular union case. Essentially, this code functions as a software write barrier that updates the collector whenever a particular union case is used in the left hand side of an assignment. The generated traversal function then checks the current value of the tag for the union, and traverses over the appropriate union case.

As noted in the GUI, autotagging is a method of last resort. It imposes overhead on the converted program, as it installs a software write barrier on the union and requires the collector to track the gathered information at runtime. Further, it may not work when the programmer plays games with the union, such as writing a pointer into an integer field or writing data without accessing the union (via pointer arithmetic). Perhaps worse, if the program passes the structure to an external library for initialization or modification, Magpie cannot insert code to autotag the union.

Nevertheless, autotagging provides a simple way for the programmer to handle cases where distinguishing information is not immediately available or obvious. Further, it handles the common case where distinguishing information is not available without extensive modifications to the original program.

3.4 Call Graph Analysis

The call graph analysis attempts to determine the target of every call site. In C, function pointers complicate this analysis, so the results are imprecise. Magpie gathers this information and then stores it for use in a later optimization phase. Thus, if optimizations are disabled, this pass may be skipped.

For completeness, Magpie uses this information to eliminate roots in the local stack. If a program calls an unknown function, Magpie must assume that the function may trigger a garbage collection. Thus, it must convert the source to notify the collector of any live local pointer variables for the collector to use them as roots. If, however, the call graph analysis has determined that the call will not

lead to a garbage collection, it does not need to transfer this information to the collector.

The callgraph analysis requires only the input files used in the other passes, and produces information only within the persistent store. Using the `top` example, the command for running the callgraph analysis is performed as follows:

```
magpie callanalysis --info-dir topinfo topsrc/*.c
```

This command gathers all the call information for all the functions in the program, and saves them to the persistent store. Unlike the previous two phases, the call graph analysis is automatic, so no further user intervention is required.

3.5 Garbage Collector Generation

Although Magpie includes only a single, basic garbage collector, it exports some flexibility to users. Currently, Magpie allows the user to specify constant value for the following collector parameters: the page size, the initial size of the nursery, and two constants used in computing the size of the new nursery after garbage collection.

The default values for page size and initial nursery size are decent base lines, and work well enough for most programs. However, if speed and/or memory use is important, programmers can tune these values to improve performance. See Section 4.7.3 for more information and suggestions for garbage collector tuning. Collector generation creates several files to be compiled and linked to the converted program. These include `gc_interface.h`, `gc_implementation.c`, `gc_tuning.h` (which contains the options passed to the generator and initialization code), and several system-dependent files (prefixed with `vm_`). By default, Magpie outputs these files to the current directory. In general, however, it is more convenient to place all the files generated by Magpie in a separate directory. For the `top` example, we will add one final directory, `topout`.

Using the standard collector flags, the command for this phase of the `top` conversion is as follows:

```
magpie gcgen --info-dir topinfo --output-directory topout
```

When the phase completes, the `topout` directory will contain all the files involved in garbage collection.

3.6 Conversion

Finally, Magpie must convert the source files. Files may be converted all in one pass, or in a group. I suggest that users convert each file individually, using a `Makefile` or similar utility, so that small changes in the source code do not require a full reconversion of the entire program. With optimizations enabled (the default), this step requires the full, global call graph generated in the call graph analysis phase.

This pass makes assumptions that should hold for standard C/C++ applications, but may require modifications for nonstandard cases:

- *Entrance Function*: To make sure the collector is initialized before the program allocates its first object, Magpie requires the entrance function of the program. The default value is `main`, which should suffice for most C/C++ applications. Libraries, plug-ins and similar code may require a different entrance function or functions. As the collector ignores any initialization calls after the first, listing multiple functions — or functions that may be called more than once — is safe.
- *Closed/Open World*: When finalizing the call graph, the analysis *by default* assumes a closed world. Specifically, it assumes that any functions not defined in the program code observed by the call graph analysis phase (see Section 3.4) will not explicitly or implicitly trigger a garbage collection before returning to the main program. Because unconverted libraries do not allocate garbage-collected objects, they will not trigger a garbage collection, so this assumption is safe except in the presence of callbacks.
- *Deallocators*: Because garbage collection removes the need for deallocation, the conversion phase removes any deallocations found in the program. Magpie, by default, assumes that there is only one deallocation function, named

`free`. If this is not the case, the user must transmit the names of the deallocators to the converter.

The only option used in the `top` example is the `--output-directory` flag, which again places the generated files into the `topout` directory. The following converts all the files in `top`, storing them in `topout`:

```
magpie gcgen --info-dir topinfo --output-directory topout topsrc/*.c
```

3.7 Compilation

The final step is not truly part of Magpie, but obviously important. Magpie generates uncompiled C source, so the user must compile these files using the available C compiler. For example, using the `gcc` compiler with no optimization, `top` is compiled using the following sequence of commands:

```
gcc -c ch.c dch.c disp.c libtop.c log.c samp.c top.c
gcc -c gc_implementation.c
gcc -o topgc *.o
```

This series of commands compiles all the (converted) source files for `top`, then compiles the garbage collector, and finally links the files into the `topgc` executable.

CHAPTER 4

IMPLEMENTING MAGPIE

This chapter explains the implementation of Magpie. Magpie is a 20,000 line Scheme program, with many of the passes and phases sharing common libraries and data structures. As previously noted, Magpie uses a simple, external persistent data store to store information across phases. Figure 2.1 shows the high-level structure of the program.

Magpie's front-end parses the preprocessed C, finesses it into a set of internal structures, and then simplifies the resulting structures. These simplifications include resolving references to their declarations, mapping names to types and transforming the source to comply with various invariants required by Magpie.[1]

The Magpie internal language (*il*) is broadly separated into four kinds of objects: declarations (*decl*), statements (*stmt*), types (*type*) and expressions (*exp*). In all cases but types, the forms mirror the source language as much as possible. This restricts the effect translations have on performance. For example, Magpie contains forms for `for` loops, `do...while` loops and `while` loops. Although it could convert all these to a single looping form, reports [42] suggest that this can have a noticeable negative influence on the quality of the code `gcc` emits. Thus, they are left in their original form.

In the case of types, the structures represent the semantic meaning of the types in the program. This causes considerable complication in Magpie's front and back ends, but makes the internal passes much simpler.

---

[1]For example, Magpie does not allow calls in the arguments of a call, so it will rewrite `foo(bar())` as `(temp = bar(), foo(temp))`.

The following describe a subset of the Magpie *il*, and are included to simplify further discussions:

$$
\begin{array}{rcll}
il & \equiv & decl & \text{(declarations)} \\
 & | & stmt & \text{(statements)} \\
 & | & type & \text{(types)} \\
 & | & exp & \text{(expressions)}
\end{array}
$$

$$
\begin{array}{rcll}
decl & \equiv & decl : fun(name, type, stmt) & \text{(function declarations)} \\
 & | & decl : var(name, type, exp) & \text{(variable declarations)} \\
 & | & decl : type(name, type) & \text{(type declarations)}
\end{array}
$$

$$
\begin{array}{rcll}
exp & \equiv & exp : empty() & \text{(empty expressions)} \\
 & | & exp : const(type, str) & \text{(constants)} \\
 & | & exp : ref(name) & \text{(references)} \\
 & | & exp : array\_acc(exp, exp) & \text{(array accesses)} \\
 & | & exp : field\_acc(exp, name) & \text{(field accesses)} \\
 & | & exp : deref(exp) & \text{(pointer dereferneces)} \\
 & | & exp : addr\_of(exp) & \text{(address of operations)} \\
 & | & exp : call(exp, list(exp)) & \text{(calls)} \\
 & | & exp : builtin(name, list(exp)) & \text{(\texttt{sizeof} et al)} \\
 & | & exp : unop(name, exp) & \text{(unary operations)} \\
 & | & exp : binop(name, exp, exp) & \text{(binary operations)} \\
 & | & exp : cast(type, exp) & \text{(casts)} \\
 & | & exp : assign(exp, exp) & \text{(assignments)} \\
 & | & exp : seq(exp, exp) & \text{(expression sequences)}
\end{array}
$$

$$
\begin{array}{rcll}
type & \equiv & type : basic(str) & \text{(builtin types)} \\
 & | & type : ref(name) & \text{(type references)} \\
 & | & type : struct(name, list(decl)) & \text{(struct refs/decls)} \\
 & | & type : union(name, list(decl)) & \text{(union refs/decls)} \\
 & | & type : fun(type, list(type)) & \text{(function types)} \\
 & | & type : array(type, exp) & \text{(array types)} \\
 & | & type : ptr(type) & \text{(pointer types)}
\end{array}
$$

$stmt$	$\equiv$	$stmt : empty()$	
	$\mid$	$stmt : if(exp, stmt, stmt)$	(if statements)
	$\mid$	$stmt : switch(exp, stmt)$	(switch statements)
	$\mid$	$stmt : exp(exp)$	(expressions as statements)
	$\mid$	$stmt : while(bool, exp, stmt)$	(while loops)
	$\mid$	$stmt : for(exp, exp, exp, stmt)$	(for loops)
	$\mid$	$stmt : return(exp)$	(returns)
	$\mid$	$stmt : goto(name)$	(gotos)
	$\mid$	$stmt : break$	(breaks)
	$\mid$	$stmt : continue$	(continues)
	$\mid$	$stmt : target(name)$	(goto destinations)
	$\mid$	$stmt : case(exp, stmt)$	(case statements)
	$\mid$	$stmt : block(list(stmt))$	(blocks)

This overview is intended to give the flavor of the internal forms. It is not, however, complete; Magpie saves considerably more information and includes many additional forms. For example, the *type* definition is missing a form for `typeof` forms, and the *exp* definition is missing a form for conditional expressions. Further, the implementation of Magpie includes considerable internal support for C++. This includes several additional forms and a considerable amount of additional analysis in the front end. However, for the purposes of describing the basic functioning of Magpie, these forms present sufficient information and give a sufficient flavor for the internals.

For the most part, the analyses and conversions described in later sections have obvious cases for most of the internal forms. Therefore, in most cases, the description gives a broad outline of how the analysis or conversion works, and then describes the exceptional cases in more detail.

4.1 Implementing the Allocation Analysis

The allocation analysis attempts to determine, for each object in the heap, what type of object it is and what its boundaries are. The allocation analysis does this by annotating allocation points. The boundaries of the object are implicitly discovered, based on the allocated pointer and its size. The type is more difficult. To determine the type, Magpie analyzes the source, computes a best guess based on these analyses, and asks the user to confirm the guess.

In some cases, Magpie can identify allocation points easily, because they use standard `libc` calls such as `malloc`, `calloc`, and so forth. In many cases, however, applications use facades for these functions to increase portability. In those situations, the facade function is considered the allocator for the purpose of the allocation analysis.

Like the structure analysis phase that follows, the allocation analysis phase gathers information and then asks the user to confirm what it has gathered. This allows Magpie to function even in cases where more simplistic systems would incorrectly analyze the source. Although it is true that Magpie seldom incorrectly identifies an allocation, this check is necessary; see Chapter 5 for more information.

4.1.1 Gathering the Allocation Analysis Information

The allocation analysis gathers information using three mutually recursive subroutines over the structure of a function definition. The analyses function as mutually recursive routines to distinguish between expressions within the `size` argument of an allocation, types and everything else. These cases are distinguished for several implementation reasons, including the different ways they handle multiplicative operations and the difference in any implicit pointers in the returned type.

At a high level, the allocation analyses are designed to watch for hints in the source about the type of a particular allocation. These include types inside a `sizeof` expression in the `size` argument of an allocation (consider `malloc(size)` and `calloc(num, size)`), numerical operations found in the `size` argument, casts on the return value of the allocation, the type of the object being reallocated in `realloc` calls, and the inferred type of the allocation based on statements such as `x = malloc(4)`.

Although each of these items could be the subject of its own analysis, Magpie combines them into a single recursive descent over every function in the program, using the three subroutines mentioned previously. The result of each of these subroutines is a list of a recursive data types describing all the possible types found in that subexpression / substatement / subtype. The recursive data type is defined as follows:

$$
\begin{aligned}
aa\_result \quad &\equiv basic \\
&\mid user(str) \\
&\mid pointer(aa\_result) \\
&\mid array(aa\_result)
\end{aligned}
$$

These four cases have exactly their expected meanings; *basic* items describe the built-in C types, *user* items describe user-defined types, the recursive *pointer* items describe a pointer to the subitem described, and the recursive *array* items describe an array of the subitem described.

The allocation analyses recur over the file in the standard way for most internal representation forms, combining the guesses for those items that have multiple recursive subtrees (`if` statements, sequencing expressions, and so on) using a standard union operator. When the recursion encounters a type, it generates the recursive return value in exactly the expected way. For simplicity, I call the recursion over types *guess_typetype*, the recursion over allocator arguments *guess_argtype* and the recursion over anything else *guess_type*. Note that *guess_type* takes an additional argument: the type analyzed for the left hand side of the parent. Passing this extra information simplifies the analysis of assignments and casts.

For *guess_argtype*, the exceptional cases involve only `sizeof` forms and binary operations. In the former case, Magpie uses this only to forward to the appropriate subroutine. Binary operations require slightly more work. In these cases, the analysis looks for cases such as `malloc(4 * sizeof(struct foo))`. In this example, the analyses should guess that the allocation point is allocating an array of structures, rather than a simple structure.

$$
\frac{name = \text{“}sizeof\text{”} \quad is\_type(il) \quad res = guess\_typetype(list(il))}{guess\_argtype(exp : builtin(name, list(il))) = res} \tag{4.1}
$$

$$
\frac{name = \text{“}sizeof\text{”} \quad is\_exp(il) \quad res = guess\_type(list(il), \epsilon)}{guess\_argtype(exp : builtin(name, list(il))) = res} \tag{4.2}
$$

$$\frac{\begin{array}{c} name \in \{*, <<\} \\ res1 = guess\_argtype(arg1) \quad res2 = guess\_argtype(arg2) \\ res1 \equiv \epsilon \quad res2 \not\equiv \epsilon \\ res = \{x \mid x = array(y) \wedge y \in res2\} \end{array}}{guess\_argtype(exp : binop(name, exp1, exp2)) = res} \tag{4.3}$$

The first two rules (4.1 and 4.2) merely dispatch the recursion to the correct function based on the kind of the argument to `sizeof`. The last equation, 4.3, is more interesting. For brevity, the rule assumes that the type information occurs in the right hand side and the multiplier in the left, but simple if statements sort this out in the implementation of the rule.

In the case of binary operations, the analysis is concerned only with multiplicative operations: multiplication and left shifts. It first determines if the form is a constant (or referenced value) multiplied by something informing the analysis of a type. If these situations hold, then the result of the analysis is the types returned from the recursion on the non-constant subexpression, wrapped in *array* forms.

Note that other operations either do not make much sense from the analysis's point of view (division, for example), or do not imply that the allocation is an array of the type (addition, for example). Therefore, in these situations, the analysis returns the union of the two recursive computations, as per normal.

The function *guess_type* has more exceptional cases, primarily involved in gathering information about the inferred type at a particular allocation point. The following is an overview of the exceptional cases, with a some left out for brevity:[2]

$$\frac{\begin{array}{c} is\_allocator(exp_f) \quad res_{type} = guess\_argtype(exp\_size) \\ res_{call} = \begin{cases} \{x \mid x = array(y) \wedge y \in res_{type}\} & \text{if } is\_calloc(exp_{fun}) \\ append(res_{type}, guess\_type(exp_{other}, \epsilon)) & \text{if } is\_realloc(exp_{fun}) \\ res_{type} & \text{otherwise} \end{cases} \\ res = append(res_{call}, res_{left}) \end{array}}{guess\_type(exp : call(exp_f, exp_1 ... exp_n), res_{left}) = res} \tag{4.4}$$

[2]For example, a side condition involving `calloc(1,type)`. Even though the allocation is a `calloc`, which strongly indicates an array, the analysis will return `type` instead of an array of `type`.

The case for calls is obviously the most important, because it is here that we gather and save the information for a particular allocation site. Because call forms differ, some case logic is required here. Magpie treats `malloc` without any post-processing. However, for `realloc` calls, Magpie takes into account the type of the pointer being reallocated, and for `calloc`, Magpie wraps all the return values as arrays.

$$\frac{\begin{array}{c} res_{type} = guess\_typetype(type) \\ res_{exp} = guess\_type(exp, res_{type}) \\ res = res_{type} \cup res_{exp} \end{array}}{guess\_type(exp : cast(type, exp), res_{left}) = res} \tag{4.5}$$

$$\frac{\begin{array}{c} res_{lval} = guess\_type(exp_{lval}, res_{left}) \\ res_{rval} = guess\_type(exp_{rval}, res_{lval}) \\ res = res_{lval} \cup res_{rval} \end{array}}{guess\_type(exp : assign(exp_{lval}, exp_{rval}), res_{left}) = res} \tag{4.6}$$

The cases for casts and assignments are the primary reason for the additional argument to *guess_type*. Magpie takes the information gathered in either the type of the cast or lvalue of the assignment, and transfers it to the expression or rvalue. Doing so allows Magpie to catch several common cases in C programs, as in the following examples:

```
{
   char *my_str = malloc(128);
   my_ptr = (int*)malloc(128);
}
```

In most cases, when Magpie cannot gather any information about an allocation, guessing that the value is atomic is a good guess. However, the cast of `malloc`'s return value — or the assignment of the allocation to something else — can give stronger suggestions that the value is atomic. Further, in some cases the additional

information from a cast can help narrow down types more accurately, particularly when `calloc` is the allocator in question.

$$\frac{\begin{array}{c} res_{array} = guess\_type(exp_{array}, \epsilon) \\ res = \{x \mid (array(x) \in res_{array}) \vee (ptr(x) \in res_{array})\} \end{array}}{guess\_type(exp : array\_acc(exp_{array}, exp_{size}), res_{left}) = res} \tag{4.7}$$

$$\frac{\begin{array}{c} res_{exp} = guess\_type(exp, \epsilon) \\ res = \{x \mid (array(x) \in res_{exp}) \vee (ptr(x) \in res_{exp})\} \end{array}}{guess\_type(exp : deref(exp), res_{left}) = res} \tag{4.8}$$

These additional rules are included as a sample of the other rules. These two rules, for array accesses and dereferences, modify the types inferred from their subexpressions — specifically by removing any pointer or array wrappings and eliding any return values that would not be valid for a dereference or array access. The address-of operator works in the reverse way.

Magpie also includes several additional rules to deal with a case not found in the internal language described previously: a GCC extension that allows programmers to place blocks in expression position. This requires some additional logic to deal with statement forms and the "return value" of such blocks.

Once the allocation analysis has gathered the information for each allocation point, it must select a best guess. The process for doing is straightforward. First, Magpie removes guesses it considers invalid, such as *basic* types. Then, it checks to see if one item appears in the list of possibilities more than any of the others. Note that Magpie can get duplicates at a call site due to res_{left} values and values from the recursion over reallocated objects in `realloc`. If it finds one type that appears more than all others, it uses that value.

Otherwise, Magpie takes the values that have the highest number of appearances in the list, and treats certain possibilities preferentially. Magpie will select a tagged array if one appears. If not, it chooses a tagged item. If neither of those appear, it looks for an array of pointers. If none of those appear, it gives up and chooses whatever happens to be first in the list it was given.

In the case where Magpie has no information about a particular allocation site, it guesses that the allocation point allocates an atomic object.

4.1.2 Converting the Allocation Points

Converting the program to transfer the gathered information to the collector is straightforward. The conversion pass walks over the program until it finds allocation points notated in the persistent store. When it finds one, it converts the call to use the appropriate allocator in the collector, adding an additional argument transferring the tag information.

Magpie generates tag names in a standard, repeatable fashion, based on the unique name assigned to the type by the front end. The creation of the tags in the C code is done by the structure analysis conversion, but `extern` declarations guarantee that the values are available to the allocation points, even if the tag is defined in another file.

4.2 Implementing the Structure Analysis

A simple analysis suffices for gathering structure information. Experience shows that, while C programmers may obfuscate their allocations from an analysis's point of view, they do not obfuscate their structure definitions. Thus, a simple recursion over the type provides high accuracy.

4.2.1 Coping with Too Many Structures

The most difficult part of the structure analysis is limiting the number of questions asked of the programmer. Asking a question for every structure in the file is an extreme burden, particularly in systems with large system headers, such as Mac OS X. The structure analysis thus attempts to limit the number of questions by only inquiring about allocated structures, by generating questions to ask on demand, and by inquiring about a structure only once per program.

By generating questions for structures on demand, Magpie avoids asking questions in the case where the allocated structure foo appears to have an inlined instance of structure bar, but in fact does not. Thus, a mistaken analysis result

does not penalize the user with multiple questions. Of course, as previously noted, this does not happen often, as the structure analysis is seldom incorrect.

The final policy — attempting to only inquire about a structure once per program — adds considerably more benefit. To do this, every time Magpie discovers a new structure that requires analysis, it first looks up that item in the persistent store. If it is found, the structure is ignored. If it is not, the structure is assigned to the current file. Afterwards, other files will not ask about it. At this point, Magpie does not handle the case where a program defines two different structure forms with the same name.

Unfortunately, the current implementation of Magpie includes a small design flaw, the result of prematurely identifying and attempting to handle an optimization problem. This optimization problem has to do with trade-offs involving inlined substructures. Consider the following example:

```
struct foo {
   int ident;
   void *ref;
};

struct bar {
   void *key;
   struct foo *my_foo;
};

struct baz {
   int num;
   struct foo *my_foo;
};

struct goo {
   struct bar *bar1;
```

```
    struct baz *baz1;
}
```

Now, consider the possibility that the program itself allocates only instances of `struct goo` and `struct bar`. One solution involves generating two sets of traversal functions: one for `struct goo` and one for `struct bar`. This solution duplicates the code required to traverse `struct foo`. If `struct foo` were a large, complex structure, this duplication could create considerable problems with regard to code size, increasing pressure on all caching levels of the memory hierarchy (including the operating system's paging routines). If the traverser for `struct foo` includes branches at the instruction level, duplicating also creates increased pressure in the processor's branch prediction mechanism.

On the other hand, if Magpie generates a traverser for every structure in the example, then Magpie creates slower code for simple traversal functions like the one generated for `struct foo`. The question, then, reduces to the general problem of inlining functions. At this point in its development, Magpie makes the optimization choice in the structure analysis, rather than during the conversion process. For the most part, it chooses to use the first option, generating only traversers for `struct goo` and `struct bar`.

For most tested programs, this choice works well. Unfortunately, in some cases, Magpie inquires about the fields of a structure multiple times within a single file, if that structure is inlined into other structures. For the most part, this redundancy is not a problem. However, in some cases, a repeatedly-inlined large structure generates a large increase in the number of questions. For example, OpenGL programs — such as the SPEC2000 benchmark Mesa — repeatedly inline a structure containing several hundred fields. See Section 5.2.2 for more detailed information.

4.2.2 Creating the Traversal Routines

Magpie generates the traversal routines as a straightforward recursion over the shape determined by the structure analysis. The conversion also notes the allocation

analysis case where an allocation generates an array of tagged objects, and generates traversers for these cases if they do not already exist. The collector currently requires only routines to mark and repair the object, so the conversion generates routines only for those things. However, additional traversal routines could be generated to add other useful functionality.

The function in Magpie that generates the traversal functions is abstracted over the specific garbage collector routine to call. For completeness, here is an overview of the routine:

- If the current item is atomic, return an empty statement.
- If the current item is a pointer, use the supplied garbage collection routine on the field.
- If the current item is an array of pointers, iterate over the array and use the supplied garbage collection routine on every element in the array. The array bounds are determined from the structure analysis and/or user, or by querying the collector to determine where the end of the object is.
- If the current item is an array of tagged objects, iterate over the array in a similar manner as the previous. However, instead of simply using a collector routine in the body of the loop, recur on the type of the array elements.
- If the current item is an inlined structure, create a block with the results of recurring over the fields in the structure.
- If the current item is an autotagged union, generate a switch statement querying the current case from the collector, and then generate each case in the switch statement by recurring over the union cases.
- If the current item is a union with user-defined distinguishing code, then insert that code directly. Then replace the stubs remaining in that code for handling each union case with the results of recurring over that union case. Specifically, when asked to write distinguishing code, the user performs the required conditionals and then leaves stubs of the form $GC\_UNION\_CASE(field)$. Magpie then looks up the fields involved and recurs over them.
- If the current item is something the user wrote their own code for, insert the

code directly.

This overview elides a few unimportant side conditions and a considerable amount of machinery, but hopefully gives a flavor of the traversal function generation process. After generating all the functions necessary for the given file, the conversion also adds declarations for all the tags required and an initialization function for the file. When called, the initialization function assign new tag values to each tag, and registers the traversal functions with the collector. Each file's initialization function is called via a global initialization function called as the first line of *main* or whichever entrance function the programmer selected.

4.3 Implementing the Call Graph Analysis

Magpie includes a call graph analysis to allow the optimization of the generated source code. Otherwise, no call graph information is necessary. For the purposes of this dissertation, the call graph analysis is simple. However, it was designed and implemented to handle C++, and thus contains considerable complications to deal with operator overloading and inheritance. It does not attempt to resolve calls through function pointers.

The call graph analysis works in two phases, to lessen the penalty of a global analysis as much as possible. The first phase simply infers the call targets for each function or method within a particular file. No attempt is made to include information about transitivity. Additional passes add information about possible invocations of subclasses in the case of dynamic method calls.

Upon demand in the final conversion, the analysis then collapses this information to determine whether a given function calls `setjmp`, an allocator, or both. A command-line flag determines whether or not Magpie will consider external functions (functions that were not available Magpie) as calling neither or both. In most cases, it is safe to assume a closed world and thus assume that external functions call neither `setjmp` nor an allocator. Exceptions include user-level threading libraries (for `setjmp`) and libraries using callbacks into converted code where the callback functions allocate.

The collapsing loop is a simple fixpoint operation. The result is cached in the persistent store, and needs to be recomputed only when a file changes. Further, because the intermediate results are also saved in the persistent store, not every file need be reparsed and reanalyzed when individual files in the program change. However, if the new call graph differs significantly from the original call graph, some files may need to be recompiled.

4.4 Implementing the Stack Conversion

The stack conversion determines which words in the C stack are pointers, and communicates this information to the garbage collector. This pass is, by far, the most complicated pass, and it is the pass that has the largest effect on performance. The pass is implemented as a series of ten subpasses, each of which modifies the source code to add or remove important information. For brevity, this dissertation gives only a high-level overview of some of the passes, and it elides one or two completely.

This section begins with an overview of the stack conversion process, and then describes the subpasses at a high level. It then discusses the implementation and important edge cases involved in some of those subpasses.

4.4.1 Overview of the Stack Conversion

As stated previously, the goal of the stack analysis pass is to identify pointers on the stack and communicate this information to the garbage collector. The conversion performs this communication by generating code to create shadow stack frames on the C stack, which the collector can then traverse to find the pointers in the normal C stack. Unfortunately, given that C programs may place arbitrary data structures on the stack, several kinds of stack frames are necessary to cope with all the possible cases in as little space as possible.

The converted code generates four possible kinds of stack frames. All these frames have their first two words in common. The first word is a pointer to the previous stack frame. A global variable tracks the current top of the stack. The second word is separated into two bitfields. The least two significant bits determine

the frame type, and the rest of the bits are used as a length field. The interpretation of the length field varies between the different kinds of frames. Figure 4.1 shows the basic format of all four kinds of stack frames.

In simple frames, the length field gives the total size of the frame. For array and tagged frames, the length refers to the number of arrays or tagged items in the frame. For complex frames, the length refers to the number of informational words appended to the end of the frame. In all cases but complex frames, the frames are capable of storing information about more than one stack-bound variable at a time. This merging avoids as much space overhead as possible.

The ten subpasses are as follows:

1. *Potential Save Inference*: This pass recurs over the function definitions and determines which stack-bound variables contain pointers. It then saves information about these variables at each call site. An overview of the information gathered is described in Section 4.4.2.

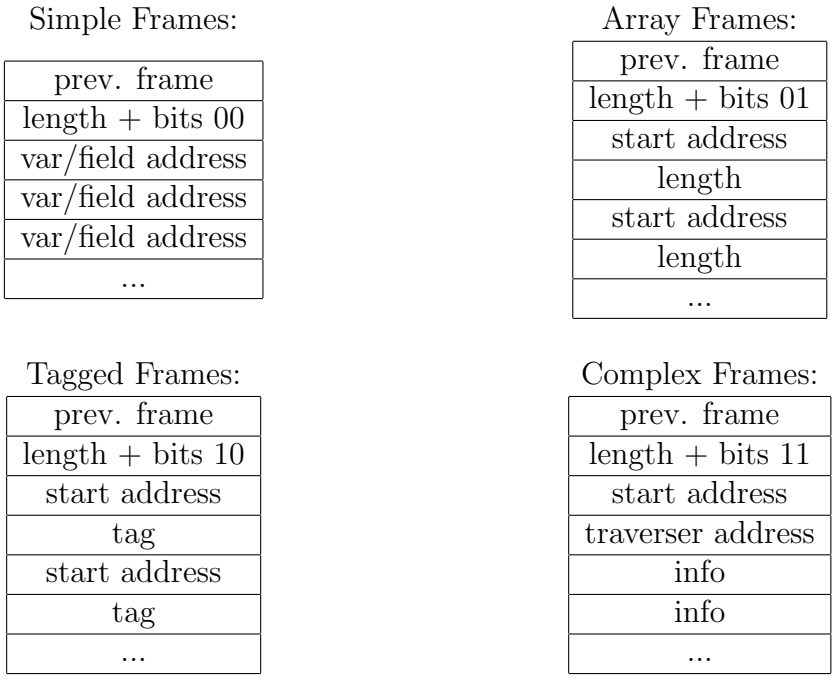

Figure 4.1. Exemplars of the four kinds of shadow stack frames in Magpie.

2. *Call Optimization*: This pass recurs over the function definitions and examines each call site. If the target of the call does not reach an allocation point, it removes the variable information from the call. If the call does not cause an allocation, then the collector will not run, so there is no need to save any information about the variables.

3. *Initialization Optimization*: In this pass, Magpie removes variable information from call sites where the variable is not initialized before reaching the call site. See Section 4.4.3 for more information about the analyses used for this pass and the following, and caveats about their behavior.

4. *Liveness Optimization*: In this pass, Magpie removes variable information from call sites if the variable is either not referenced after the call or is written to before being read from. Again, see 4.4.3 for more information.

5. *Switch To Save Statements*: This pass takes information about the call sites and lifts it to the nearest preceding statement position. This essentially creates a stub statement where Magpie will add the generated code.

6. *Remove Dominated Saves*: This pass removes variable information from the save statements generated in the case where a previously-occurring save statement is already saving that variable.

7. *Simplification*: At this point, Magpie runs a simplification pass to clean up code left by the previous analyses and conversions.

8. *Insert Stack Pushes*: This pass creates the code to save all the information denoted in each save statement. This pass, thus, does most of the work of the stack conversion, and is described in more detail in Section 4.4.4.

9. *Insert Stack Ejects*: This pass creates the code to pop shadow stacks when their scope ends. There are a few odd cases here, described in Section 4.4.5.

10. *Final Clean-Up*: This pass simplifies the generated code; it removes unnecessary blocks, breaks expression sequences into separate statements when possible, and so forth.

The goal of the optimizations is to reduce — ideally to zero — the number of variables Magpie needs to save within a given function. Magpie allocates frames within the C stack, which allows for fast frame creation and deletion. Thus, the actual allocation of the space is essentially free, and the only performance loss from frame generation comes as a result of the additional cache pressure caused by the increased space consumption. Further, frame initialization is most likely also cheap, as the relevant portions of the stack are likely in the L1 or L2 caches (or would have been shortly, regardless).

Instead, the slowdown created by the stack frames has to do with the restrictions imposed on the final C compiler by taking the address of a local variable. Taking the address is necessary, since Magpie must be compiler-agnostic while conveying information about relevant local variables in the stack. By taking the address of the local variable, Magpie essentially forbids the final C compiler from placing that local variable in a register. This restriction not only affects the performance of the program *a priori*, it may also inhibit further optimizations of the generated assembly code.

As noted, Magpie performs three optimizations: a call graph-based optimization, and two liveness-based optimizations. The latter are applications of typical compiler-based liveness analyses and optimizations to the domain of Magpie [35]. Thus, the considerable research into more effective liveness analyses and optimizations in compilers could be used instead of the simplistic approaches used in Magpie, with potential gains in their effectiveness.

4.4.2 Internal Variable Shape Forms

The initial analysis gathers information about all variables that may need to be saved in the conversion process. This information includes the unique, internal name of the variable, the tag names (if applicable) for the type, an expression (in

internal format) for accessing the item, and information about the item's shape. The shape itself is described in a recursive data type with the following five cases:

1. *Simple*: These items are simple pointers, and can be a pointer variable, a pointer field within a structure, or a list of pointer fields within a structure.

2. *Array*: These items are simple arrays of pointers.

3. *Tagged*: These items are stack-bound data structures, the type of which was found in the results of the structure analysis. The conversion will prefer this pseudo-shape over compound simple shapes, complex shapes or union shapes, because the programmer was involved in creating the traversal function for the object.

4. *Union*: These items are unions. This shape saves additional information mapping the original definition of each union case with the shape information generated by recurring over it.

5. *Complex*: Complex shapes are shapes not described by any of the previous item, and each complex shape contains a list describing its subparts.

Although simplistic, the standard usage of each of these is as follows. Simple items are simple local pointer variables. Array items are stack-bound arrays of pointers, including multidimensional arrays of pointers and arrays of structures in which every word within the structure is a pointer. Union items are obvious. Complex items are only needed with stack-bound arrays of structures, where the structures contain a mix of pointer and non-pointer items.

When performing the conversion between save information and frame generation, Magpie puts union shapes in complex frames and all other shapes in the frame kind with the same name.

4.4.3 Caveats Regarding Optimizations

Both the initialization and liveness optimizations use straightforward, near-linear time,[3] flow-insensitive analyses. Both analyses use pessimistic assumptions on a function if it can, at any transitive point, call `setjmp`, because returning to a particular point may invalidate the liveness information gathered at that point.

Neither analysis pass has particularly interesting exceptional cases. Most of the implementation work is in assuring that the recursion operates in the correct order for the pass. The initialization optimization walks the function definition in the order the statements would be executed. The liveness optimization walks the tree in the reverse order.

For the rest of this section, I discuss some of the limitations of the analyses in detail. For simplicity, I discuss only the initialization analysis, but the limitations apply to both analyses.

The initialization analysis attempts to discover variables that, although noted as potentially needing to be saved, have not been initialized. It also detects variables that have been obviously initialized to $NULL$. However, the analysis's notion of an obvious initialization to $NULL$ may seem restrictive to programmers. The analysis does not perform any constant folding, and variables with compound data types are considered initialized once they are referenced on the left hand side of an assignment. Thus, all elements of a 20 pointer array are assumed initialized as soon as one of the elements is initialized. Because Magpie ensures that all stack and root variables are initialized before they are used, this assumption is safe.

Further, once a variable is considered initialized, it is considered initialized throughout the entirety of the function. Thus, in the following case, the variable foo is considered initialized in `after code` even though it has been set to $NULL$:

```
{
   void *foo = something;
```

[3]The analyses are linear except in the case of loop structures, where the loop test (and modification in the case of `for` loops) is analyzed twice. This is necessary because these positions may contain arbitrary C code.

```
   ... before code ...
   foo = NULL;
   ... after code ...
}
```

Because the analysis is flow insensitive, it will also consider *foo* to be initialized in `after code` in the following case:

```
{
   void *foo;

   if(...) {
      ...
      foo = something;
      ...
   } else {
      ...
   }
   ... after code ...
}
```

Both analyses share similar limitations due to their simplicity. More thorough analyses would generate better information and thus potentially uncover more optimization opportunities. However one goal in the design of Magpie was to make every analysis and conversion pass as close as possible to linear in the size of the program to avoid excessive compilation time.

4.4.4 Adding Stack Frames

Magpie converts the save-information stubs into the final code by replacing the stubs with all the declarations and statements required to save every variable listed in the stub. For convenience, this transformation places variable declarations in

arbitrary positions within a block, which violates the C standard. A later pass either moves these declarations to the head of the block or adds blocks to produce valid C. The conversion attempts to place the saves as late as possible in the execution stream, in the hopes that early returns and conditionals will avoid the need to run the code. However, in the case of looping forms, Magpie attempts to pull any stack frame additions outside the loop, for performance reasons.

4.4.4.1 Simple Saves

Because simple saves are the most frequent form of saves, Magpie attempts to minimize the number of simple frames generated. Thus, at any point where a simple-shaped variable needs to be saved, that variable is saved in a pre-existing simple frame if one exists, even if that frame was generated at a much higher-level block. Whenever possible, the conversion will reuse slots in the simple frame for new simple saves when they are no longer used by other variables.

However, simple frames are still generated at the latest possible point, so a function may have multiple simple frames, or a single simple save frame in one branch of a conditional. The former case occurs wherever one simple frame does not flow-insensitively dominate the others.

The mechanics behind implementing this functionality are somewhat heinous, involving a considerable number of additional function arguments and the use of lazy evaluation. However, from a research perspective, the implementation is uninteresting.

4.4.4.2 Array and Tagged Saves

Because array and tagged saves are not nearly as common, every save stub that includes array and/or tagged saves adds fresh stack frames. This potentially wastes space (for the additional frame headers) and time (to generate the frames) at the benefit of greatly increased simplicity in implementation.

The code to generate these frames is straightforward. Each case generates the appropriate frame type with a sufficient number of slots, and then adds the necessary information. In both cases, this includes the address of the variable. It

also includes the size of the array for array frames (found by examining its type and involving multiplication in the case of multidimensional arrays), or the tag appropriate to the variable for tagged frames.

4.4.4.3 Complex Saves

Unlike simple, array, and tagged saves, complex frames hold information about only one variable per frame. This strategy is due to the lack of copy propagation and constant folding in Magpie, meaning that Magpie may not be able to determine a numerical size for an array statically. Consider the following code snippet:

```
void function()
{
   int foo = 5;
   struct { int num; void *ptr; } my_info[foo * 4];

   ...
}
```

This code will compile under GCC, because GCC can determine that `my_info` is a stack-bound array of 20 structures. Magpie, lacking the necessary analyses and optimizations, cannot. Therefore, in this case, Magpie will transfer the size information in one of the `info` fields of a complex frame (see Figure 4.1).

More generally, Magpie handles situations like the previous — a complex shape involving either a union or an array of a nonhomogeneous structure — by creating a traversal function and passing the address of the function to the collector, along with any array size information the traversal function requires. The generation of these structures is directly comparable to the generation of traversers for the structure analysis. The differences involve the arguments to the generated procedure. Traversers generated by the structure analysis take only a single argument — the object — whereas traversers generated by the stack conversion take the object, the address of the garbage collection function (mark or repair) to use for the traversal,

and a pointer to the info block.

Unions are handled exactly as autotagged unions in the structure analysis. The appropriate items are marked as needing to be autotagged, and the traversal generation simply generates a `switch` statement querying the current tag value from the garbage collector.

4.4.5 Removing Stack Frames

The removal of a stack frame at the end of its scope is fairly straightforward. A first subpass runs through the function definition and saves information about the block level of call targets. The second pass adds the code to pop to the correct stack level by recurring over the function and observing stack frames as they are pushed, the ends of blocks, and nonlocal jumps.

At the end of a block, the global stack variable is reset to its value on entering the block. At a nonlocal jump, the stack is reset to the stack frame that should exist at that particular target. Nonlocal jumps include the following cases:

- `break`: The stack is reset to the stack frame that was in effect before the loop or `switch` statement started.
- `continue`: The stack is reset to the stack frame that should be in effect at the start of the loop. This is rarely different from the previous case.
- `goto`: The stack is reset to the stack frame that should be in effect at the `goto` target's block level.
- `return`: The stack is reset to its original value when execution entered the function.

Note that some simplification is performed to get rid of dead code created by these general rules. For example, the previous items insert stack ejections both at return statements and at the end of blocks, which leads to dead code after return statements. Further, considerable care is taken to handle GCC's extension allowing blocks in expression position, because the last statement in the block is the block's value.

4.5 Implementing Autotagging

Magpie automatically tags some unions to distinguish between union cases when traversing the appropriate object. It implements this feature by adding a simple software "write" barrier. I place the word "write" in quotation marks because Magpie also includes barrier code when the program takes the address of a union.

The barrier is added to the code using a basic type inference schema, which attempts to determine if a given field access (`foo.field`) references a particular case in a union. It then checks to see if the item occurs in certain situations in the code, and if so, generates a call to the collector giving the address of the union and the union case chosen.

The implementation of the autotagging conversion is a linear, combined analysis and conversion pass with a three-value return. The three values are the new internal form post-conversion, a list of inferred types for the form, and any calls to the collector's autotagging functions necessary within the sub form. The calls in this final value are placed at the nearest appropriate point, by saving the result of assignment or address-of operation in a temporary variable, calling the autotagging routine, and then returning the saved result.

The following are some of the exceptional cases, added for specificity. The arguments to the function (*autotag*) are, in order: the form being converted, information about the types being autotagged, and a boolean determining whether or not the form is in an lvalue.

$$\frac{\begin{array}{l} < exp'_{array} types_{array} calls >= autotag(exp_{array}, types, lval?) \\ < exp'_{size--} >= autotag(exp_{size}, types, FALSE) \\ rettypes = \{x \mid (array(x) \in types_{array}) \vee (ptr(x) \in types_{array})\} \\ exp' = exp : array\_acc(exp'_{array}, exp'_{size}) \end{array}}{autotag(exp : array\_acc(exp_{array}, expsize), types, lval?) =< exp' rettypes calls} \tag{4.9}$$

The interesting part of this conversion rule is that it essentially ignores any autotagging information found in exp_{size}. This is because the size subexpression in an array access is manifestly not an lvalue, so there is no need to be concerned

about any union field accesses found within it, even if the array access itself appears in an lvalue.

$$\frac{\begin{array}{l} < exp'_{lvalue}, infos_{lvalue}, exps_{lvalue} >= autotag(exp_{lvalue}, types, TRUE) \\ < exp'_{rvalue}, infos_{rvalue}, exps_{rvalue} >= autotag(exp_{rvalue}, types, lval?) \\ myexps = SHOULD\_AUTOTAG(exp'_{lvalue}, exp'_{rvalue}, infos_{lvalue}, infos_{rvalue}) \\ exps = append(exps_{lvalue}, expsrvalue, myexps) \\ NOT(NULL?(exps)) \\ res_{exp} = MAKE\_SEQUENCE(exp : assign(temp, exp_{rvalue}), exps, temp) \\ res_{infos} = infos_{lvalue} \cup infos_{rvalue} \end{array}}{autotag(exp : assign(exp_{lvalue}, exp_{rvalue}), types, lval?) =< res_{exp}, res_{info}, \epsilon >} \quad (4.10)$$

I include this case as an example of how the autotagging calls, once generated, are included into the final source. In this case, the assignment detects that autotagging calls are necessary around the assignment by observing the result of SHOULD_AUTOTAG. It adds these calls by creating an expression sequence; the sequence starts with a save of the original expression's value to a temporary, then executes the autotagging calls required, and then returns the saved value. The implementation of autotagging requires additional infrastructure (not shown in this rule) that creates the temporary value with the correct type.

As stated before, because this is a software write barrier, it will not catch writes to unions that use pointer arithmetic or occur outside the purview of the conversion process. Unfortunately, although a reentrant hardware write barrier — a write barrier that allows the exception-handling code to reset the write barrier after execution — would identify these writes correctly, it would have no information about the union case being selected, and thus would be of no benefit. Finally, the autotagging conversion will not catch cases where the program lies about what it is writing to the union case — writing a pointer to an integer case, for example.

4.6 Dealing with Shared Libraries

Shared libraries present no problem to Magpie, except in the case that a library function saves a reference internally (past the life of the call) or invokes a call that may allocate. The latter case effects the call analysis and call optimization, and

is discussed in Section 4.3. The former case is the subject of the immobility flags discussed in Section 2.2.2 and this section.

Cases where a library saves a reference into the garbage-collected heap cause problems for two separate reasons. The first is that the object might be incorrectly collected if the collector is not aware of this saved reference and there are no other reference paths from the observable roots. The second problem is that, if the collector moves the object during collection, the saved reference becomes invalid.

The ideal solution involves converting the library using Magpie, but conversion is not feasible in all cases. For example, converting the entire Gnome GUI library suite for a small application is prohibitively costly. In these cases, Magpie allows the use of annotations around either the function specifications in the library headers or the arguments to the function invocation in the program source.

In either case, the conversion adds a call into the garbage collector noting any pointers found within the annotation's scope. The collector then guarantees that the objects referenced by these pointers are never collected and never moved. The former case is overconservative if the library is guaranteed to throw away the stored value at some predictable point. The `libc` function `strtok` is an example of such a function. However, it is unclear how such invariants could be expressed easily in the general case, so the collector never resets immobile objects. Improving this behavior is an interesting avenue of future investigation.

The translation to add this code is trivial, simply checking for pointers within annotated calls and creating C expression sequences that tell the collector that a pointer in the given argument should be marked as an immobile root and then returning the original expression.

4.7 Implementing the Garbage Collector

I have attempted to make the garbage collection interface for Magpie as general as possible, in the hopes of allowing other collectors to function with Magpie-converted code. Some programs may require the responsiveness guarantees of an incremental collector, or perform much better with an older-object first collector.

Should Magpie be extended to handle multiple system threads, a parallel collector may be necessary.

In its current form, the garbage collector included with Magpie is a standard, two-generational, stop-the-world collector. The nursery is allocated as a single block, and objects allocated into the nursery are prepended with their tag. Objects surviving the nursery are copied into the older generation, which is collected using a mark-and-compact algorithm. The nursery is dynamically sized based on the amount of memory used by the system after garbage collection, and the collector attempts to interact will with other applications by giving memory pages back to the underlying operation system as soon as possible. This system is very similar to Dybvig, Eby and Bruggeman's BiBOP collector [19].

As implemented, the collector is portable over several operating systems, and support is being added to handle using the collector inside Linux kernel modules. The interface to the underlying system simply requires primitives to allocate, free and write-protect large blocks of memory. At the moment, Magpie supports Linux, FreeBSD, Solaris, Windows and Mac OS X.

Because C does not guarantee that pointer variables (or pointer fields) reference the start of an object, the collector uses an out-of-band bitmap alongside each garbage collector page. This bitmap includes information about where objects start and whether or not they are marked. In C programs where references always reference the head of an object, a per-object header field might be preferable for performance reasons. However, because this assumption cannot be made in the general case, the choice was forced. Currently, the bitmap adds an overhead of two bits per word. Information on whether or not an object has been moved is implicit and based on what page the object occurs on.

4.7.1 Implementing Autotagging

The collector tracks autotagging data using a derivative of a trie data structure. The basis of this structure is a 16-ary tree, with each subtree holding the key, the value associated with the key, and (potentially) pointers to child nodes. The lookup function checks to see if the current node's key is equal to the sought after key, and

returns the value if so. If not, the lookup function examines the least significant four bits of the sought after key, and uses that as the index for recurring to the child. The key value is shifted four bits to the right upon recursion.

The set and remove functions of this data structure are obvious. The only additional interesting note on the structure, as implemented by the collector, is that the array of pointers to subtrees is allocated on demand and removed when no longer necessary. This seems to have a positive effect on space efficiency in test cases.

The collector also notes words in memory as being autotagged using the off-line bitmap. This allows the mapped references to be easily updated during collection. After collection is complete, the structure is walked, and any mappings referencing collected objects are removed.

4.7.2 Implementing Immobility

The current implementation of object immobility in the collector is heavyweight. Essentially, a space is created to hold the object, the object is copied and marked as moved, and then the collector forces a garbage collection so that any objects referencing the object are updated. This means that every call to the immobilization procedure invokes a garbage collection, which has obvious performance consequences.

Improvements on this system are obvious, but have not been investigated due to time constraints.

4.7.3 Tuning the Garbage Collector

Although the basic functioning of the included garbage collector is fixed, several values can be tuned during the `gcgen` phase of the conversion process. These constants have to do with the sizes of various structures within the collector, but they can have noticeable effects on performance if tuned well (or poorly).

The first constant sets the size of collector page in the older generation. How to tune this size depends on several factors in the application, and the choice is probably best simply described as black magic, but general rules do hold. A smaller

page size increases overhead in both time and space for large heaps, because the collector tracks some amount of metadata for every page and must occasionally walk the list of active pages. Larger page sizes decrease this overhead, but increase the possibility of excessive fragmentation if there exists a type that is used infrequently. For example, if the collector is tuned to use 16kb pages and the application allocates and retains a single object of type foo, that retention will cause the collector to maintain an entire page just for that object.

The second constant tunes the initial nursery size for the application. This constant is simpler to generalize about: large applications should use large values, small applications should use small values. The trade-offs here involve the cost of allocating a large chunk of memory as one of the application's first instructions, versus potentially triggering several collections during program start up. In general, the goal of tuning this value is to attempt to set it so the fewest number of collections occur during the initialization of the program.

The final constants tune what size the collector sets aside for the new nursery after every garbage collection. This size is computed via the following function:

$$new\_size = (grow\_factor * current\_heap\_size) + grow\_addition$$

The two tunable constants are $grow\_factor$ and $grow\_addition$. Tuning these two constants is similar to tuning the initial nursery size; tuning them so the function generates too small a size causes too many garbage collections, but tuning them so the function generates to large a size may cause excessive page allocation slowdowns. These values, however, are considerably harder to tune, as they are based on the dynamic behavior of the program. In general, the built-in constants should be left alone, but those working to get every possible microsecond of performance out of the program may find these constants helpful.

4.8 Threads and Magpie

As described and currently implemented, Magpie functions only over single-threaded code. Extending Magpie to handle multithreaded code ranges in difficulty

from moderately difficult to extremely difficult depending on the level of parallelism desired.

As it stands, Magpie uses a single, global shadow frame stack. To handle multiple threads, each thread would need its own stack, and the code would have to be able to quickly and easily update the stack for each particular thread. Because OS thread libraries are not easily changed, this would involve the creation of an additional data structure mapping thread identities to their appropriate stack. Instead of simply referencing or setting a single global variable, the stack conversion would have to reference or set the current value in this table. Further, Magpie would need some way to identify instances of any thread-local data as roots for garbage collection.

Secondly, for the collector to function, it must reach safe points in every running thread before starting a garbage collection. On the bright side, it is easy to find a simple safe point: the point after new shadow stacks have been added. The conversion could then add barrier notification code at each of these points, and the collector could simply block until all garbage collected threads have reached such a point.

The disadvantages of this simplistic conversion are twofold. First, barriers are not a cheap mutual exclusion tool, and the cost of using them may be unacceptable *a priori*. Second, there is no way of guaranteeing that these safe points appear at any regular interval, particularly with the optimizations turned on. So it would be possible — in edge cases — for a thread that does allocations only during initialization to cause a system deadlock by never reaching a safe point after the initialization.

CHAPTER 5

THE COST OF CONVERSION

This chapter reports on the costs of using Magpie. The costs of Magpie come in two areas: costs in time for converting the program using Magpie, and costs created by modifying the program. I assert that the more important cost is the former. The goal of Magpie is to save the programmer time and effort in dealing with memory management in their large, old applications. Therefore, the ease with which she can convert a program with Magpie is of primary importance.

The latter costs — the costs in program execution created by the conversion process — are of lesser importance. Magpie strives not to have too major an influence on the time and space behavior of the converted program. However, it is my belief that minor changes in the space requirements or time efficiency matter little in comparison to increased programmer efficiency and a decreased chance of bugs.

The chapter begins with an overview of the benchmarks used to evaluate the performance of Magpie. It continues with a comparison between converting the program to use garbage collection with the conservative, non-moving, Boehm collector versus using Magpie. It then reports on the space and performance impact of using Magpie, and concludes with some observations on the causes of the space and performance changes.

5.1 An Overview of the Benchmarks

Table 5.1 outlines the benchmarks used in evaluating Magpie. These benchmarks include integer and floating point benchmarks from SPEC2000, and represent an interesting variety of programs.

Table 5.1. An overview of the size of the various benchmarks used. All preprocessed files generated on Mac OS/X 10.4.6.

Program	# Files	LOC	Preprocessed Size	Decrufted Size
164.gzip	20	8,605	640KB	144KB
175.vpr	41	17,729	1.3MB	524KB
176.gcc	127	229,693	7.7MB	5.5MB
177.mesa	122	61,754	7.9MB	4.0MB
179.art	1	1,270	60KB	28KB
181.mcf	25	2,412	624KB	64KB
183.equake	1	1,513	68KB	36KB
186.crafty	43	21,150	2.1MB	616KB
188.ammp	31	13,483	1.3MB	380KB
197.parser	18	11,391	804KB	348KB
254.gap	63	71,363	2.8MB	1.6MB
256.bzip2	3	4,649	140KB	76KB
300.twolf	85	20,459	2.3MB	804KB

The largest, GCC, is representative of many common C programs in its memory use, performing allocations and deallocations throughout its lifetime. Internally, GCC serves as a useful benchmark of Magpie's tolerance for edge cases in C programs; GCC uses most GCC extensions and rarely-used C forms, as well as many questionable programming practices. For example, GCC frequently invokes functions without declaring the function in any way. The `300.twolf` benchmark has a similar allocation/deallocation profile.

Benchmarks such as `186.crafty` and `254.gap` behave in the opposite way. They quickly allocate a large block of memory during an initialization phase, and then allocate little memory during program execution. These programs serve as examples of the cost of the code transformations performed by Magpie.

The `197.parser` benchmark is particularly interesting, because it defines its own implementation of `malloc` and `free`. The conversion process uses the call into this subsystem as its allocator and removes calls to the custom deallocator, effectively making the custom allocation/deallocation system dead code. Thus, although the original benchmark allocates a large fixed block and uses it for the

lifetime of the program, the Magpie converted program varies its heap size.

The rest of the programs fill the space between these examples. Most allocate a large part of their memory in an initialization phase, and then allocate and deallocate smaller amounts during the main execution phase. The `177.mesa` benchmark exists only in a PPC version, because I could not get any version of the benchmark to compile under FreeBSD.

Finally, preprocessing the original files adds considerable cruft, in the way of unused variable, function and type declarations. Magpie's front end removes much of this cruft before the input is fed to any of the other analyses or conversions. The size of each benchmark after this decrufting is reported in the last column of Table 5.1. Thus, the execution time of Magpie's front end is a function of the column $PreprocessedSize$, but the execution time of the remainder of the system is a function of the column $DecruftedSize$.

5.2 Converting the Benchmarks

Theoretically, using the Boehm collector should be strictly easier than using Magpie. After all, theoretically all that is required to use the Boehm collector with an existing program is relinking the program. In my experience converting the benchmark programs, however, the opposite was true: converting programs using Magpie was strictly easier than using the Boehm collector.

5.2.1 Using Boehm with the Benchmarks

In the ideal case, converting a program to use the Boehm collector requires relinking the program and/or a mechanical search and replace over the program source. All calls to `malloc` are replaced with `GC_malloc`, and so on. The Boehm collector even provides a mechanism to perform the conversion by relinking the file; the linker relinks the `malloc` calls to `GC_malloc`.

In practice, this was not true for all of the benchmarks on either FreeBSD or Mac OS X. On FreeBSD, most of the benchmarks required only a mechanical search and replace, changing `malloc` to `GC_malloc` and removing `free`. Under Mac OS/X,

the opposite was true. I had to intervene with most benchmarks under OS X to get them to work with the Boehm collector.

A large part of this additional work involves identifying global variables used as program roots and informing the Boehm collector of them; this may be a bug in the Boehm collector on the PPC platform. Having Magpie made this considerably easier for me than it would be for someone without Magpie. Magpie identified all the global roots for its own conversion, and I simply copied what it found. Still, since the conventions to inform the collector of roots were different, this took some amount of time. How long this would take for someone without Magpie is unknown, particularly given the questionable programming practices of some of the SPEC benchmarks.

The conversion to Boehm for `197.parser` required even more work. Since the benchmark uses its own custom allocator and deallocator, converting the program to the Boehm collector required changing all the custom calls to use Boehm.

Finally, after several days, I gave up trying to figure out how to convert `176.gcc` to use Boehm.

These failures may be indicative a bug in the Boehm collector, as it is supposed to find roots within the program. However, that the Boehm collector – a mature, well-tested piece of software – fails to find these roots may also be indicative that finding these roots is extremely difficult in the general case.

5.2.2 Magpie

The most difficult part of converting the benchmarks with Magpie is finding what functions to pass with the `allocators` argument. Most SPEC benchmarks use facade functions or macros in place of direct calls to `malloc`, `calloc` and `realloc`. However, a simple search for the string "alloc" found all the facade functions quickly; I believe I spent less than two hours finding all the facade functions in all of the benchmarks.

After that, using Magpie was easier than using the Boehm collector in every case. Optimistically, I tried running all the analyses with the `--auto-run` flag.

Since there were no errors using the defaults found by the analyses, I never required the GUI. The most difficult part of the conversion process was writing the Makefile.

Table 5.2 shows the cost to the programmer for performing the allocation analysis. The majority of the time spent in the allocation analysis is in parsing the program. Unfortunately, because this is the first pass of the conversion process, the only way to speed this phase up would be to improve the performance of the parser. Times in this figure are approximate; they were generated using only a single run on a 1.5GHz G4 PowerPC processor running Apple's Mac OS X 10.4.6.

Table 5.3 shows the cost of the structure analysis. Again, the structure analysis was correct on all items, and could be run automatically. In the case of `177.mesa`, the OpenGL structures trigger an unfortunate case in the structure analysis, so questions for one or two large structures are asked multiple times. This, in turn, slows down the execution of the analysis, as the structure analysis must dynamically check to see if the user's answers require it to ask questions about additional

Table 5.2. The cost of the allocation analysis for each of the benchmark programs. Parse time is the time spent in parsing and massaging the source into the correct internal formats. User time is the amount of time the programmer spends answering questions. All times approximate.

Program	Questions	# Wrong	Total Time	Parse Time
164.gzip	5	0	0m18s	0m17s
175.vpr	104	0	0m48s	0m47s
176.gcc	64	0	3m44s	3m43s
177.mesa	67	0	3m24s	3m23s
179.art	11	0	0m04s	0m04s
181.mcf	4	0	0m15s	0m15s
183.equake	29	0	0m05s	0m05s
186.crafty	12	0	0m52s	0m52s
188.ammp	37	0	0m37s	0m37s
197.parser	110	0	0m25s	0m24s
254.gap	2	0	1m26s	1m26s
256.bzip2	10	0	0m07s	0m7s
300.twolf	188	0	1m03s	1m01s

Table 5.3. The cost of the structure analysis for each of the benchmark programs. Parse time is the time spend in parsing and massaging the source in the correct internal formats. User time is the amount of time the programmer spends answering questions. All times approximate.

Program	Questions	# Wrong	Total Time	Parse Time
164.gzip	5	0	0m09s	0m08s
175.vpr	87	0	0m28s	0m26s
176.gcc	218	0	4m08s	3m38s
177.mesa	1557	0	8m22s	1m36s
179.art	10	0	0m05s	0m04s
181.mcf	23	0	0m05s	0m04s
183.equake	0	0	0m02s	<0m01s
186.crafty	17	0	0m26s	0m25s
188.ammp	140	0	0m39s	0m36s
197.parser	93	0	0m25s	0m21s
254.gap	0	0	0m03s	<0m01s
256.bzip2	0	0	0m02s	<0m01s
300.twolf	181	0	1m14s	1m09s

structures. This behavior is the cause of the greatly increased time spent in the structure analysis phase, considering the behavior in the other cases and the size of the benchmark.

Again, the times given in the figure are approximate, based on a single run. Note that the analysis can cut down on parse times considerably compared to the allocation analysis. Because the structure analysis needs information on only a limited number of structures — those structures noted by the allocation analysis — it can avoid parsing files once it has all the information it needs. This can result in greatly reduced times in the structure analysis. For some of the benchmarks, reordering the list of files given to the structure analysis can lower the parse times even further than reported.

Finally, Table 5.4 reports the time spent in the other phases of the conversion. The "Call Analysis" column reports the time spent in the call analysis phase. As noted previously, this step can be skipped if optimizations are turned off in the

Table 5.4. The cost of the automatic conversions. Conversion time is the time spent by Magpie in the various analyses, transformations and additions required to take the original file and create the internal representation of the converted file. Total convert time includes parsing, unparsing and recompilation of the file.

Program	Call Analysis Time	Conversion Time	Total Convert Time
164.gzip	0m18s	0m06s (0m05s)	1m05s
175.vpr	0m37s	0m20s (0m18s)	2m05s
176.gcc	3m44s	5m04s (4m21s)	16m16s
177.mesa	3m20s	3m37s (3m21s)	11m01s
179.art	0m04s	0m01s (0m01s)	0m10s
181.mcf	0m15s	0m03s (0m02s)	0m49s
183.equake	0m05s	0m01s (0m01s)	0m12s
186.crafty	0m52s	0m19s (0m16s)	3m25s
188.ammp	0m36s	0m21s (0m17s)	2m24s
197.parser	0m25s	0m17s (0m16s)	1m40s
254.gap	1m31s	1m01s (0m53s)	5m20s
256.bzip2	0m07s	0m03s (0m02s)	0m19s
300.twolf	1m2s	0m40s (0m37s)	5m22s

conversion phase. The "Conversion Time" column reports on how long Magpie spends analyzing and rewriting the original source, and does not include time spent parsing or unparsing the source. The number in parenthesis is the time spent if optimizations are disabled. Finally, the "Total Convert Time" reports the amount of time spent doing the final conversion. This includes generating the garbage collector, parsing the source, converting the source, unparsing the source and compiling the converted source. Again, all times are approximate, using a single run on an Apple G4.

These times are significantly higher than compiling the original source. This slowdown is largely due to the cost of parsing the source. Again, a faster parser would have the largest benefit in speeding up the entire Magpie system. The unparser is similarly slow, although not as dramatically as the parser. However, the time spent in this process is computer time, not programmer time: the programmer is free to do other things while the conversion is running.

5.2.3 Unions in the Benchmarks

Several of the benchmarks use unions. Table 5.5 shows the unions found in the benchmarks, and how they are treated. The costs for these unions shows up in three places. First, the conversion engine must spend extra time adding in code to do the automatic tagging required for autotagged unions. Second, this autotagging has a cost in performance, because the autotagging code essentially creates a software write barrier on any instances of the union. This write barrier calls into the collector and performs an insert/update operation on a tree.

Finally, the data structure in the garbage collector adds both time and space overhead to garbage collection outside of the traversal function's lookup cost. The time component is spent checking whether moved items have autotagged words inside them, and transferring the autotagging data to the new location. The space is spent keeping track of the autotagging data. I put some effort into minimizing the space utilization and maximizing the performance of the relevant data structure, but considerable space is still required if many objects are autotagged.

Table 5.5. The number of unions in each of the benchmark programs, and how they are handled for the conversion.

Program	Unions	# Autotagged	# Not Autotagged
164.gzip	1	1	0
175.vpr	1	1	0
176.gcc	1	1	0
177.mesa	1	1	0
179.art	0	0	0
181.mcf	0	0	0
183.equake	0	0	0
186.crafty	0	0	0
188.ammp	1	1	0
197.parser	1	1	0
197.parser*	1	0	1
254.gap	0	0	0
256.bzip2	0	0	0
300.twolf	0	0	0

The union in the `197.parser` benchmark is a simple union of pointer types. Magpie does not notice this, however, and the automatic analysis configures the union to use autotagging. To show the costs of autotagging, I created a second version, `197.parser*` in which I wrote my own case distinguishing code for the union, which simply always chooses the first union case. I cannot provide principled results on how long this took, since I did this second conversion after I did the original conversion. Thus, I knew that all the results generated by the allocation and structure analyses were correct, and the second conversion involved me clicking "Next" unreasonably quickly.

5.2.4 Executable Size

Table 5.6 reports on Magpie's impact on executable size. Since Magpie adds additional statements and expressions to the source code, as well as the additional traversal procedures and a garbage collector, it is not surprising that the size of the Magpie-converted executables are uniformly larger than in the base or Boehm versions. Both the Magpie and Boehm collectors are statically linked to the executable.

Table 5.6. The impact of the Magpie conversion on executable sizes.

Program	Base	Boehm	Magpie
164.gzip	128KB	128KB (1.00)	192KB (1.51)
175.vpr	432KB	704KB (1.63)	656KB (1.52)
176.gcc	3.3MB	3.6MB (1.08)	6.6MB (1.99)
177.mesa	3.5MB	3.5MB (1.00)	4.0MB (1.32)
179.art	52KB	328KB (6.56)	108KB (2.11)
181.mcf	64KB	340KB (5.28)	120KB (1.84)
183.equake	60KB	340KB (5.73)	116KB (1.92)
186.crafty	488KB	764KB (1.56)	596KB (1.27)
188.ammp	364KB	640KB (1.76)	552KB (1.52)
197.parser	300KB	580KB (1.93)	504KB (1.68)
254.gap	1.1MB	1.1MB (1.00)	2.0MB (1.82)
256.bzip2	92KB	92KB (1.00)	148KB (1.60)
300.twolf	628KB	908KB (1.45)	808KB (1.29)

5.3 The Cost in Time

Table 5.7 shows the cost of adding garbage collection to the benchmarks. The table provides information for a register-poor machine (FreeBSD x86) and a comparatively register-rich machine (OS X PPC). Because the stack conversion code saves roots on the stack by taking the addresses of local variables, it may have a major impact on register allocation. Thus, it is informative to look at the difference in results between a register-poor machine and a register-rich machine.

The table reports results on comparing five different versions of each benchmark. The *Base* version is the original version of the program. The column "Base Time" reports on the average execution time of this program. All the following columns contain values normalized to this number.

The *Boehm* column reports the change in execution time created by linking the original program to the Boehm collector. As noted earlier, the mechanism for converting the program to use the Boehm collector varied between the programs. Some of the programs required only relinking or a mechanical search and replace, others required extensive patching. In all cases, an effort was made to maintain the basic allocation patterns of the original program. There are no results for the Boehm collector in the case of `176.gcc`, because I could not get a Boehm-collected version to work.

The *NoGC* column reports on the difference in performance found when performing the conversion, but not using a garbage collector. These files are generated by performing the normal Magpie conversion, with two modifications. The first replaces the garbage collector with a simple facade to `malloc`. Thus, a call to `GC_malloc` simply calls `malloc`, ignoring any tagging information. Similarly, calls to the autotagging operations do nothing. Second, in the conversion phase, I generate the converted program using a random string for the `--deallocators` argument. Thus, deallocations in the program are not removed, but all other calls into the collector are maintained. Doing this gives a rough approximation of the cost of the conversion, not including any garbage collection costs.

One benchmark, `197.parser`, defines its own allocation and deallocation sub-

Table 5.7. The performance impact of garbage collection on the benchmarks.

Program	Invoc.	Base Time	Base	Boehm	NoGC	NoOpt	Magpie
Apple OS X PPC							
164.gzip	5	3m59.5s	1.00	0.99	1.00	1.09	1.07
175.vpr	1	3m33.9s	1.00	1.00	1.05	1.07	1.04
176.gcc	5	3m13.8s	1.00	N/A	1.22	1.24	1.14
177.mesa	1	4m16.3s	1.00	0.98	0.99	1.30	1.11
179.art	2	15m38.7s	1.00	0.96	1.00	1.01	1.00
181.mcf	1	12m03.7s	1.00	0.99	1.02	1.02	1.02
183.equake	1	7m17.4s	1.00	0.98	1.00	0.90	0.90
186.crafty	1	3m08.9s	1.00	1.00	1.02	1.03	1.00
188.ammp	1	17m00.0s	1.00	1.07	1.15	1.11	1.08
197.parser	1	0m12.6s	1.00	1.49	N/A	5.09	4.80
197.parser*	1	0m12.6s	1.00	1.49	N/A	3.68	3.39
254.gap	1	3m56.1s	1.00	1.00	1.57	1.59	1.47
256.bzip2	3	6m49.5s	1.00	1.01	1.00	1.00	0.99
300.twolf	1	11m56.8s	1.00	0.95	1.02	0.83	0.83
FreeBSD x86							
164.gzip	5	4m34.5s	1.00	1.03	1.24	1.31	1.09
175.vpr	1	4m04.3s	1.00	0.98	1.04	1.02	0.96
176.gcc	5	3m03.1s	1.00	N/A	1.60	1.61	1.34
179.art	2	8m28.5s	1.00	1.00	0.99	0.94	0.94
181.mcf	1	6m24.5s	1.00	1.00	1.15	1.15	1.00
183.equake	1	2m58.5s	1.00	1.00	0.99	1.02	0.99
186.crafty	1	3m27.2s	1.00	0.99	1.10	1.09	1.02
188.ammp	1	14m31.8s	1.00	0.90	1.16	0.96	0.96
197.parser	1	0m10.1s	1.00	1.44	N/A	6.89	5.35
197.parser*	1	0m10.1s	1.00	1.44	N/A	5.22	3.52
254.gap	1	3m08.7s	1.00	1.01	2.48	2.46	2.39
256.bzip2	3	5m54.9s	1.00	1.00	0.99	0.99	0.99
300.twolf	1	13m02.9s	1.00	0.99	1.08	0.89	0.88

system. Unfortunately, that means the *NoGC* conversion uses `malloc` for its allocations but uses the custom deallocation routine for deallocations. This causes the *NoGC* version to fail unpredictably for `197.parser`, so that data is not available.

The *NoOpt* column reports on the performance of the Magpie conversion with optimizations turned off. Otherwise, the program is converted normally, and includes the default collector. The final column, *Magpie*, reports on the performance of the normal Magpie conversion, using optimizations and the default collector.

Both *NoOpt* and *Magpie* make use of an untuned collector; all the options to the `gcgen` phase are left with their default values. It is possible that tuning these values further would improve performance. For completeness, the default collector uses 16KB pages and a 2MB initial nursery size.

Every version of every benchmark is run six times on each platform, comprising of a priming run and five timed runs. Some benchmarks are defined in terms of multiple program executions; these cases are noted in the "Invoc." column of Table 5.7. The numbers reflect the average of these five runs. Standard deviations were trivial for all programs, generally 1% or less of the total execution time. The machine for x86 runs is a 1.8GHz Pentium IV CPU running FreeBSD 4.11-STABLE with 256MB of memory. The machine for PPC runs is a 1.5Ghz PowerPC G4 running OS X 10.4.6 with 1.25GB of memory.

5.3.1 Comparing *Base* and *NoGC*

The difference between *Base* and *NoGC* is entirely due to the conversion of the source code. In most cases, the converted versions run at the same speed or slower than the original program. In many cases, this slowdown is less than 25%. In a few cases, however, the slowdown is drastic: `254.gap` runs significantly slower, and is discussed in Section 5.3.6. In four cases the conversion speeds up the program very slightly (1%).

The conversion performs several changes on the original source code. Obviously, the conversion adds code to create and remove shadow stack frames. However, the Magpie front-end changes the code in many other ways, essentially by forcing evaluation order on some expression types. For example, calls contained within the

arguments of other calls are lifted and converted to a sequence of calls. Magpie also changes many assignments to use a temporary variable.[1]

These modifications affect the ability of GCC to analyze the source code. In most cases, the modifications create small differences in the performance of the final executable. However, in some cases it appears that the modifications have a considerably larger effect on the final performance of the executable.

5.3.2 Comparing *NoGC* and *NoOpt*

The *NoGC* benchmark is generated exactly as the *NoOpt* benchmark, with two modifications. First, the "collector" used is just a simple facade over `malloc`. Second, no calls to `free` are removed. Thus, *NoGC* shows the cost of the Magpie conversion, without including the cost difference in using a different memory management style.

Whether garbage collection or manual memory management is faster depends strongly on the program and underlying platform. On the PPC, three benchmarks have better performance with *NoOpt* than with *NoGC*: `183.equake`, `188.ammp` and `300.twolf`. On the x86, considerably more benchmarks are faster using garbage collection: `175.vpr`, `179.art`, `188.ammp`, `254.gap` and `300.twolf`.

5.3.3 Comparing *NoOpt* and *Magpie*

As expected, using optimizations never slows the converted code down. This is unsurprising, since the goal of the optimizations is simply to remove cases where variables need to be saved. Since these optimizations do not involve transforming the code to lower the performance of the generated program, there is little chance that the optimizations will accidentally convert the code so that the program is slowed down.

---

[1]This is necessary due to GCC producing unpredictable code for assignments. For example, in the expression `array[i][j] = foo(bar)`, GCC will sometimes produce code that partially or fully computes the address of the array accesses before invoking the call. This creates problems when the call triggers a garbage collection, as the collection may have moved the array.

On the OS X PPC platform, the optimizations had no impact on performance in some cases. `181.mcf`, `183.equake`, `256.bzip2`, and `300.twolf` gain no performance improvement with the optimizations turned on. In most cases, the advantage of using the optimizations was minimal: 3% or less. In a few cases, however, using the optimizations created notable performance benefits.

On the FreeBSD x86 platform, the advantage of using the optimizations were considerably more noticeable. This is surprising. I would have expected the effects on a comparatively register-poor machine to be much less significant. There are several possible reasons for this. First, the OS X benchmarks use a version of GCC version 4, while the FreeBSD benchmarks use version 2.59.4. So it is possible that the earlier version is more strongly affected by the stack frame code.

Another possible reason is that the internals of the Pentium IV CPU are less forgiving of the converted code than the internals of the PowerPC. Modern x86 architectures translate the input x86 binary code into an internal instruction set; generally, a RISC-based instruction set with many more registers. It is possible that the modifications made by Magpie inhibit the efficiency of either the translation or the execution of the translated instructions. Since the details of both the translation and internal implementation are trade secrets, it is difficult to discover if either is the case.

5.3.4 The Cost of Autotagging

The only difference between the two versions of `197.parser` is autotagging. `197.parser` uses autotagging, whereas I wrote my own code to distinguish the union cases in the `197.parser*` case. Viewing these two benchmarks alone, autotagging seems to have a significantly negative effect on program performance. On the PowerPC platform, the autotagged version performs 480% worse than the *Base* version, as compared to the nonautotagged version's slowdown of 339%. On the x86 platform, the autotagged version performs 352% worse whereas the nonautotagged version performs 239$ worse.

However, in the other five programs that use autotagging — `164.gzip`, `175.vpr`, `176.gcc`, `177.mesa` and `188.ammp` — the performance effect of Magpie is consid-

erably less noticeable, and it is unclear how much of a factor autotagging is in these numbers. On one benchmark — `188.ammp` on FreeBSD — the converted, autotagged *Magpie* version even runs faster than *Base*.

5.3.5 Comparing *Base*, *Boehm* and *Magpie*

Ignoring the `197.parser` and `254.gap` benchmarks, Magpie performs reasonably well compared to both *Base* and *Boehm*. On the PowerPC, in all cases but the two `197.parser` versions and the `254.gap` benchmark, the performance of Magpie-converted code with optimizations is within 15% of the original program and Boehm. Interestingly, the results for the Boehm collector are not strongly predictive of the results for *Magpie*. In the ten cases in which the *Boehm* version is faster than the *Base* version, *Magpie* is also faster than the *Base* version in only five. On the other hand, in the eight cases in which the *Boehm* version is slower than the *Base* version, *Magpie* is also slower in seven of them.

On the PowerPC, Magpie is largely slower than the base version, usually by less than 15%. On two benchmarks, `183.equake` and `300.twolf`, *Magpie* is clearly faster than the *Base* version. On another, `256.bzip2`, Magpie is only slightly faster (a total difference of only a few seconds).

The results are considerably more varied on the x86. Six of the thirteen programs show slight to moderate performance improvements using Magpie. The `300.twolf` benchmark shows a considerable performance improvement of 12%. On the other hand, `176.gcc` performs 34% worse and `254.gap` performs 146% worse, as compared to 14% and 47% on the PowerPC.

5.3.6 Examining `197.parser` and `254.gap`

`197.parser` and `254.gap` perform badly compared to the other benchmarks. `197.parser` behaves particularly badly, slowing down by a factor of 3.3 times (the non-autotagged version under OS X) to 5.4 times (the autotagged version under FreeBSD).

Most of the slowdown in the `197.parser` benchmark comes from not using the custom allocator used in the base version. As noted in Section 5.1, the base

version of the benchmark eschews the system's `malloc` and `free` for its own version, which makes an automatic translation to the *NoGC* version impossible. However, I modified the output of the automatic translation by hand, and discovered that this hand-generated version of *NoGC* slowed down by a factor of 1.44 on OS X. This number is slightly better than the Boehm results on the same platform.

For `254.gap`, some of the slowdown comes from the program's extensive use of function pointers. As noted in Section 4.3, the call graph analysis used in Magpie makes no attempt to resolve call targets through function pointers. This inability inhibits Magpie from making use of many optimization opportunities.

Further examination with a profiler does not lead to a strong indication of one single factor causing the remaining slowdown. However, profiling data does suggest that taking the address of variables in tight loops may be causing some of the slowdown.

5.3.7 Possible Shadow Variable Optimization

One possible way to avoid taking the address of important variables is to use temporary variables to shield the them from Magpie taking their addresses. Thus, instead of saving the variable *foo* directly, Magpie might convert the code to use a temporary variable, *bar*, surrounding the call. For example, such a conversion might change the following:

```
{
   ...
   x = allocating_call(foo);
   ...
}
```

into

```
{
   ...
   bar = foo;
```

```
    x = allocating_call(bar);
    foo = bar;
    ...
}
```

In performing this conversion, Magpie would not need to transmit information about *foo* to the collector. Instead, *bar* would be transmitted to the collector (and its address taken), and *foo* would be left alone. The disadvantage of this method is that it may increase the stack use of the program if the compiler cannot optimize the potential additional space, and the extra assignment costs may negate any performance gained by not taking the address of the primary variable.

I have implemented a prototype version of this optimization, to see if this optimization might be of use in future versions of Magpie. This optimization is placed after the other three optimizations, but before any other modifications have been made to the program, and inserts the assignments and variable declarations required. It only functions for simple stack frame types; see Section 4.4.2 for more information on simple forms. I tested the results of the benchmark under OS X.

For the modified `197.parser` benchmark, this optimization had a negative effect, slowing the benchmark down to 1.64 times the speed of the base version — a difference of 0.20. The optimization had only a small negative effect on `254.gap`: the Magpie version with the optimization ran 1.49 times slower than the base version, a difference of 0.02.

I tested the optimization two other benchmarks, `177.mesa` and `186.crafty`, and achieved similar results. `177.mesa` slowed down to 1.33 times the speed of the original, or an additional 22%. `186.crafty` showed a minimal improvement, running at 0.99 times the speed of the original, or a 1% improvement.

It is possible that more selective use of this optimization would result in better performance improvements. However, this would require either user intervention or a complicated analysis to determine which variables to use with the optimization.

5.4 Space Usage

To compute the space use of the original and modified programs, I modified the base versions of each benchmark. The modifications entailed the addition of a call into a space-tracking library function, inserted as the first and last action of `main` as well as after every allocation. This function computed the total space used by the program via a system, and saved the data to a file.[2] Figures 5.1 through 5.12 show the memory behavior for the *Base*, *Boehm* and *Magpie* versions of each benchmark.

The choice to use a system call to fetch the entire size of the program is an important one. First, it includes the size of the code segment. Magpie-converted versions will have larger code sizes due to the conversion process adding information for the collector. Second, the standard implementations of `malloc` hides information about the metadata used in their memory management systems.

In most cases, the Magpie-converted program requires more space than the original or Boehm versions. However, much of this space comes from pre-allocated (2MB) nursery space. Neither the original nor Boehm versions pre-allocate space for new allocations, but the collector included with Magpie does. In many cases, tuning the collector more carefully would shrink these differences to nearly zero; essentially, the programmer would rely on the fact that the programs allocate large blocks of memory during initialization, and tune the constants so that post-collection nursery sizes were near zero.

`164.gzip`, `175.vpr`, `176.gcc`, `188.ammp` and both versions of `300.twolf` use autotagging (Figures 5.1, 5.2, 5.3, 5.8 and 5.12, respectively). In these benchmarks, the gradual increases in total program size — the ones not reflected in the other versions — are mostly likely the increased space cost of autotagging. As the converted program autotags objects, more data are dynamically added to the total space cost of the execution.

`164.gzip` is a particularly good example of this effect. The benchmark uses unions in the definition of its Huffman encoding trees. Thus, the compression

[2]Many thanks to Jed Davis, who told me how to do this.

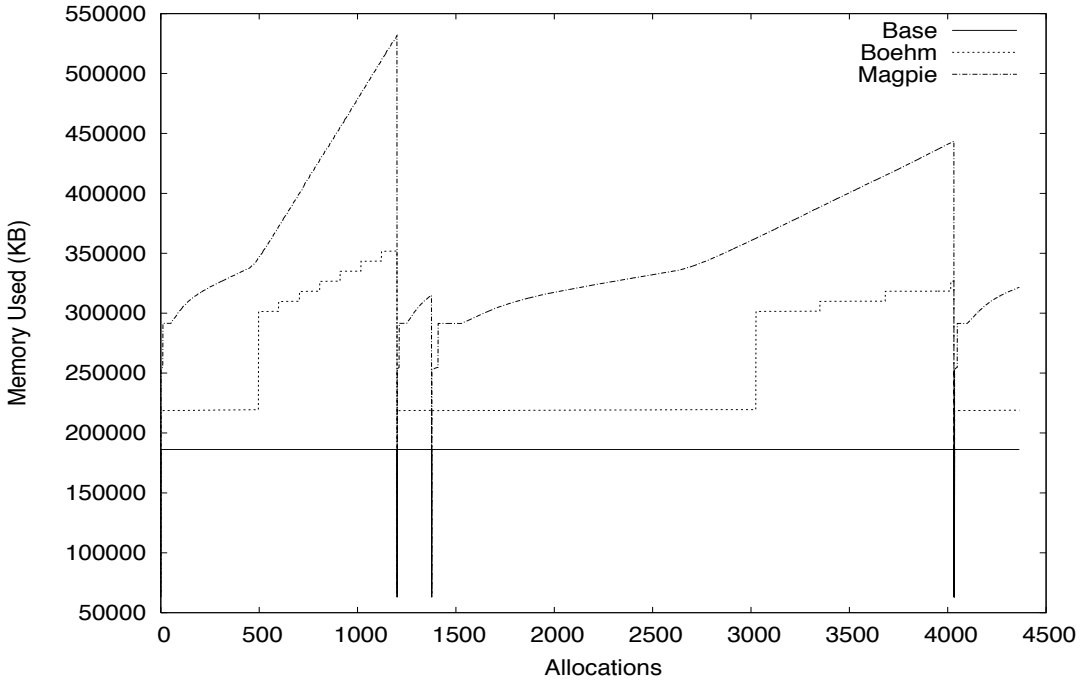

Figure 5.1. The memory behavior of the `164.gzip` benchmark.

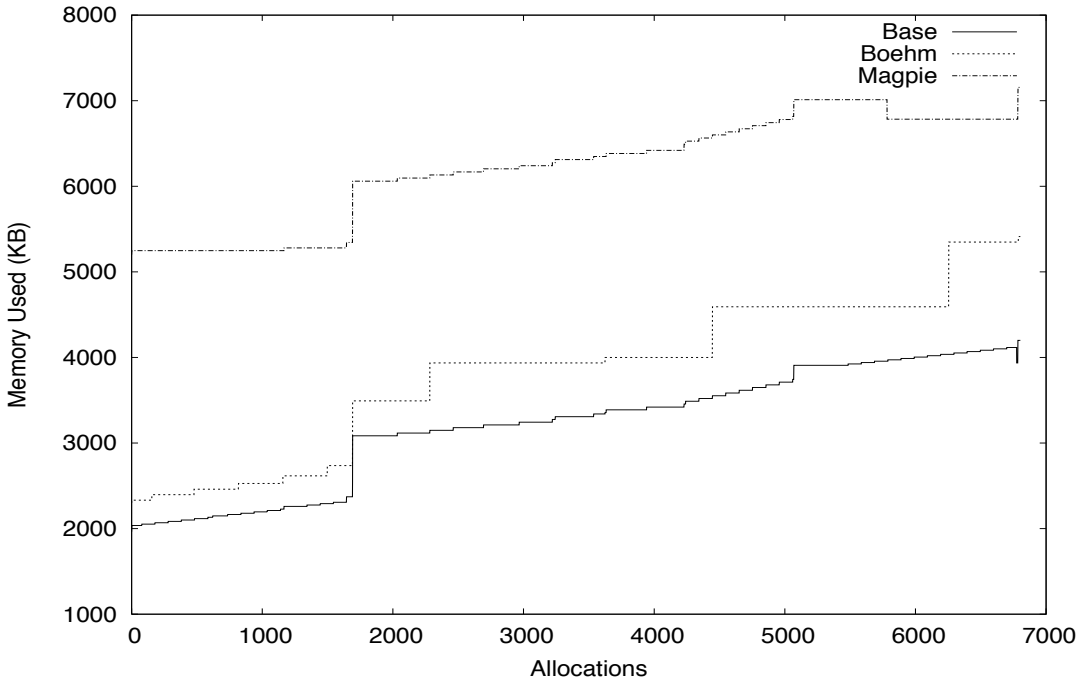

Figure 5.2. The memory behavior of the `175.vpr` benchmark.

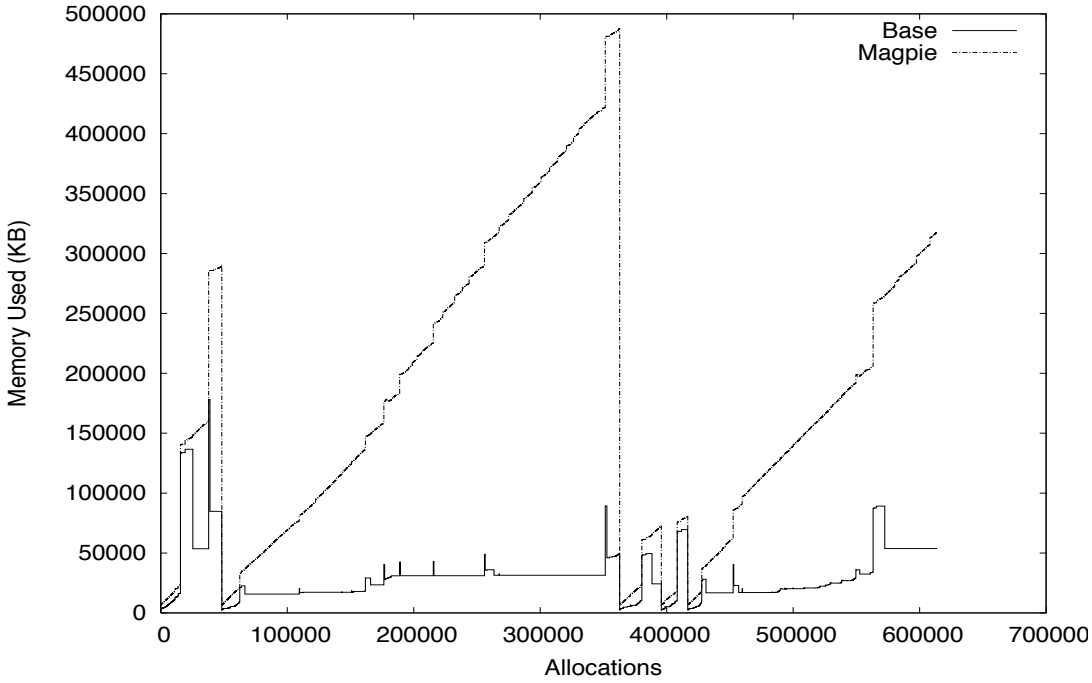

Figure 5.3. The memory behavior of the `176.gcc` benchmark.

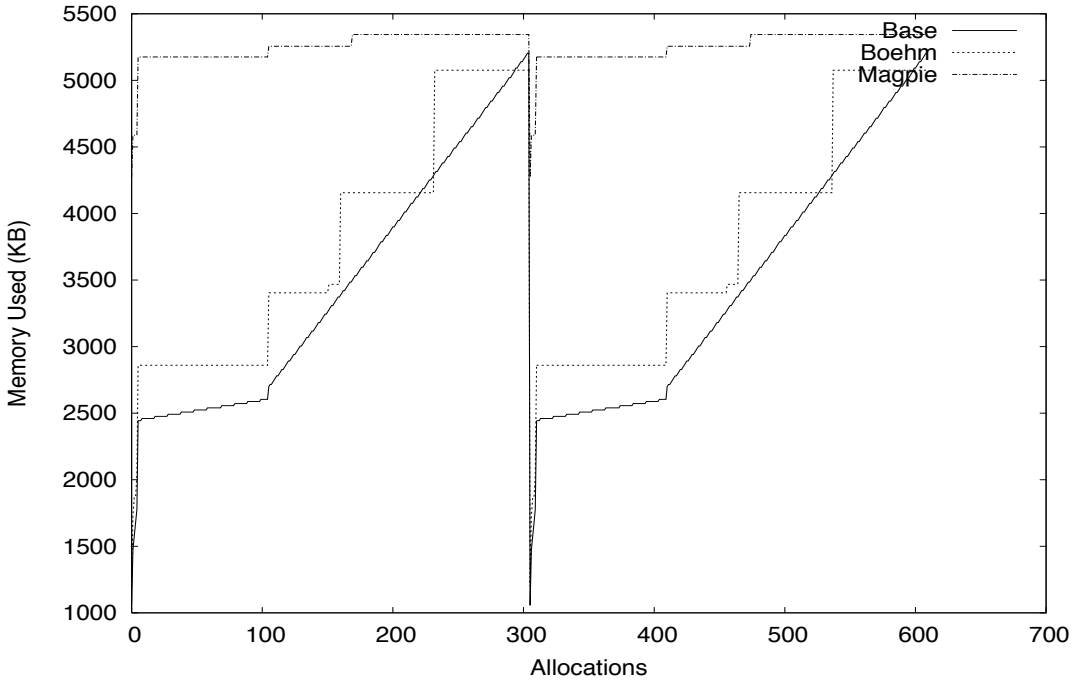

Figure 5.4. The memory behavior of the `179.art` benchmark.

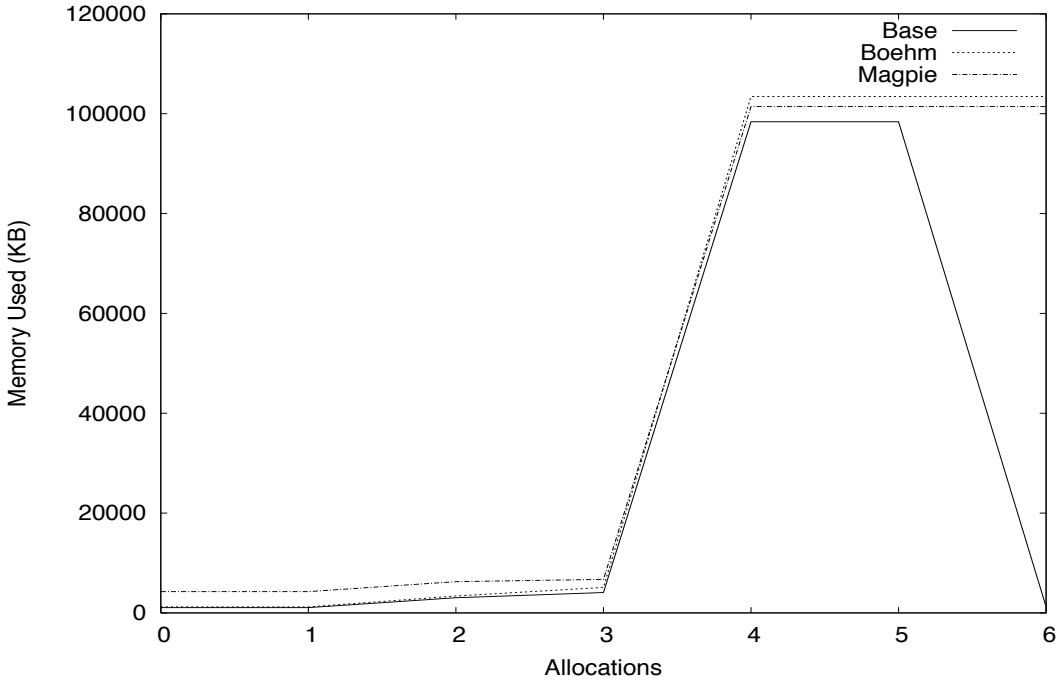

Figure 5.5. The memory behavior of the `181.mcf` benchmark.

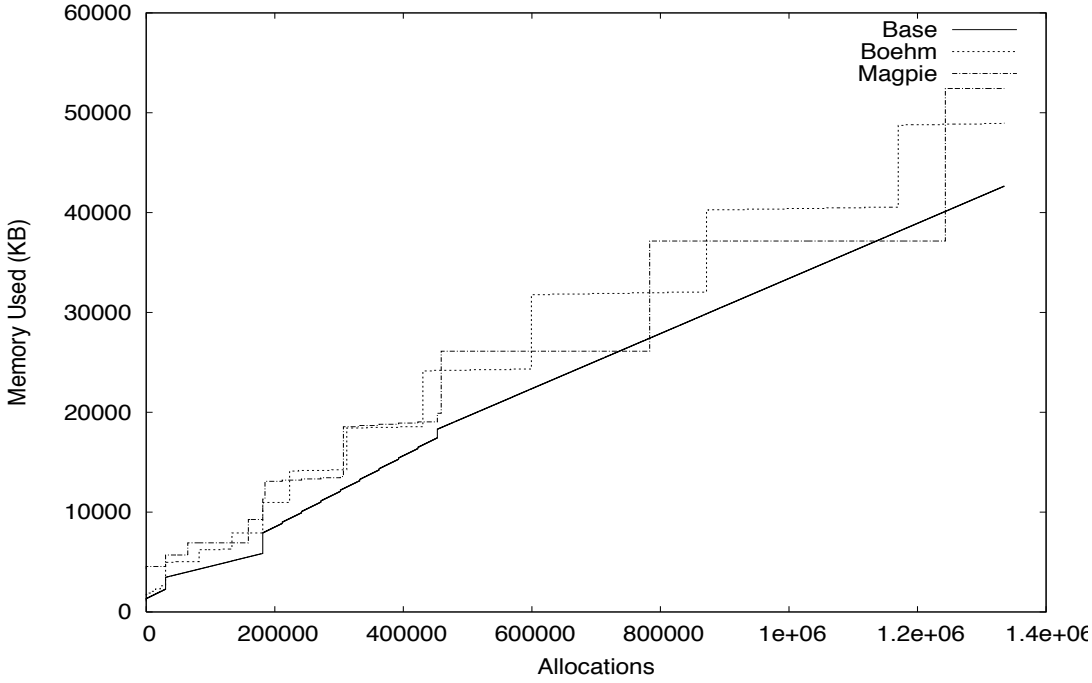

Figure 5.6. The memory behavior of the `183.equake` benchmark.

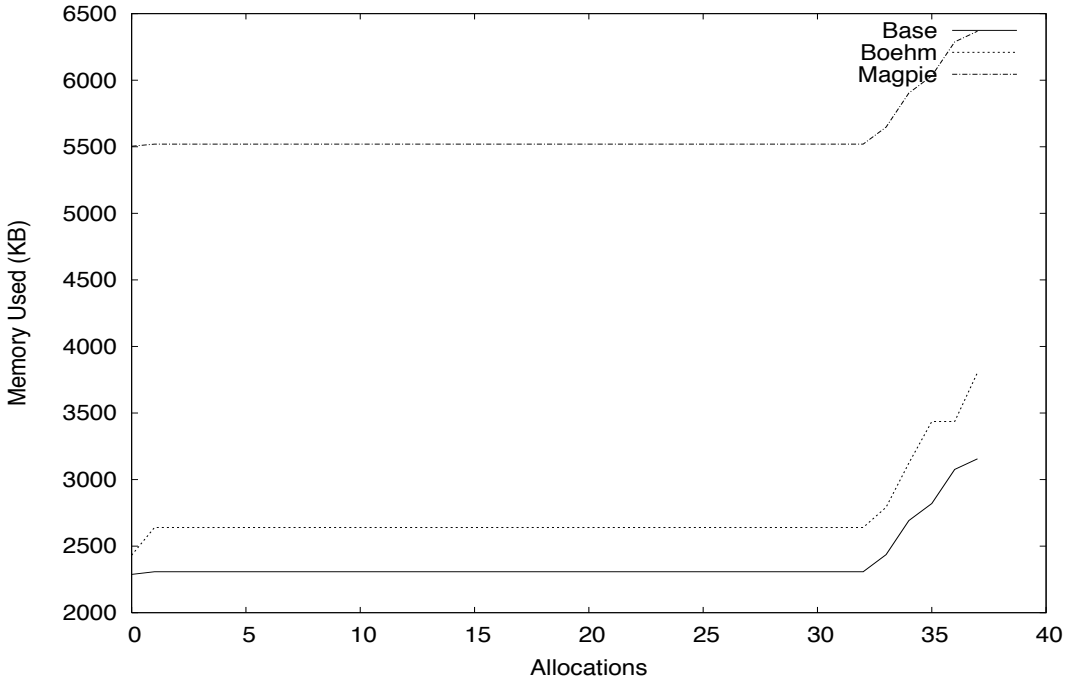

Figure 5.7. The memory behavior of the `186.crafty` benchmark.

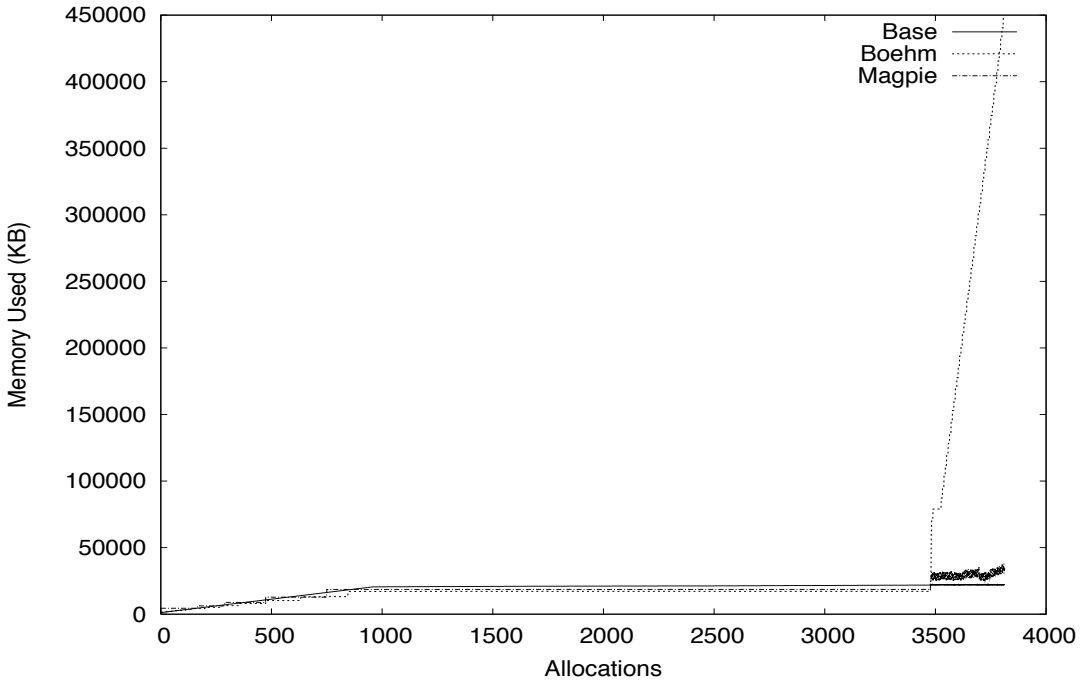

Figure 5.8. The memory behavior of the `188.ammp` benchmark.

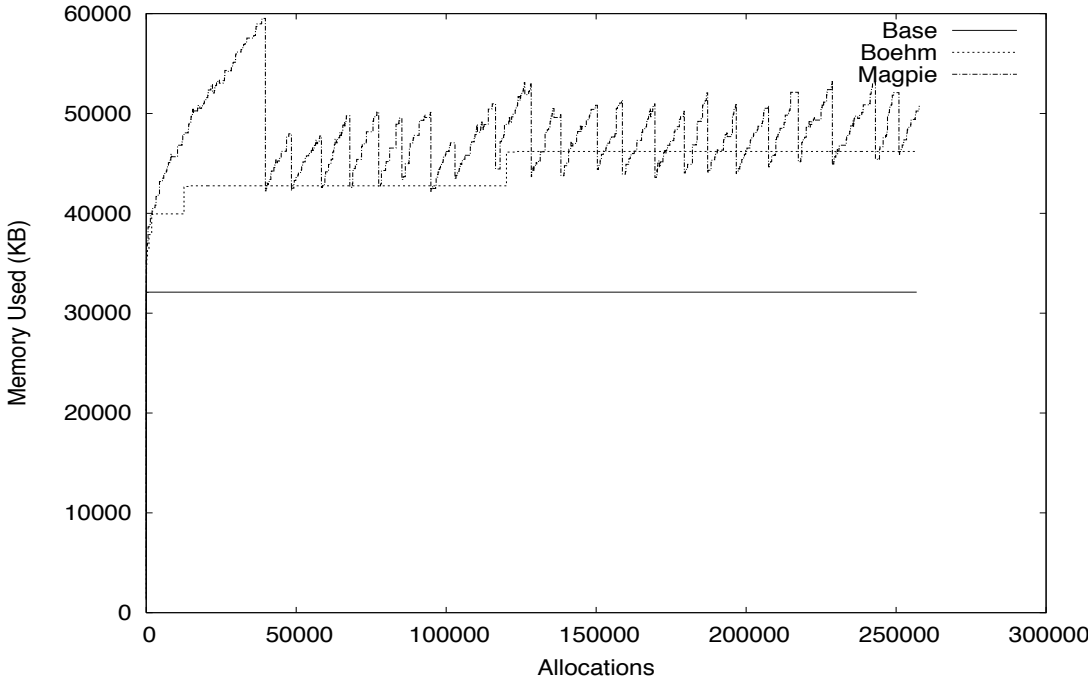

Figure 5.9. The memory behavior of the `197.parser` benchmark.

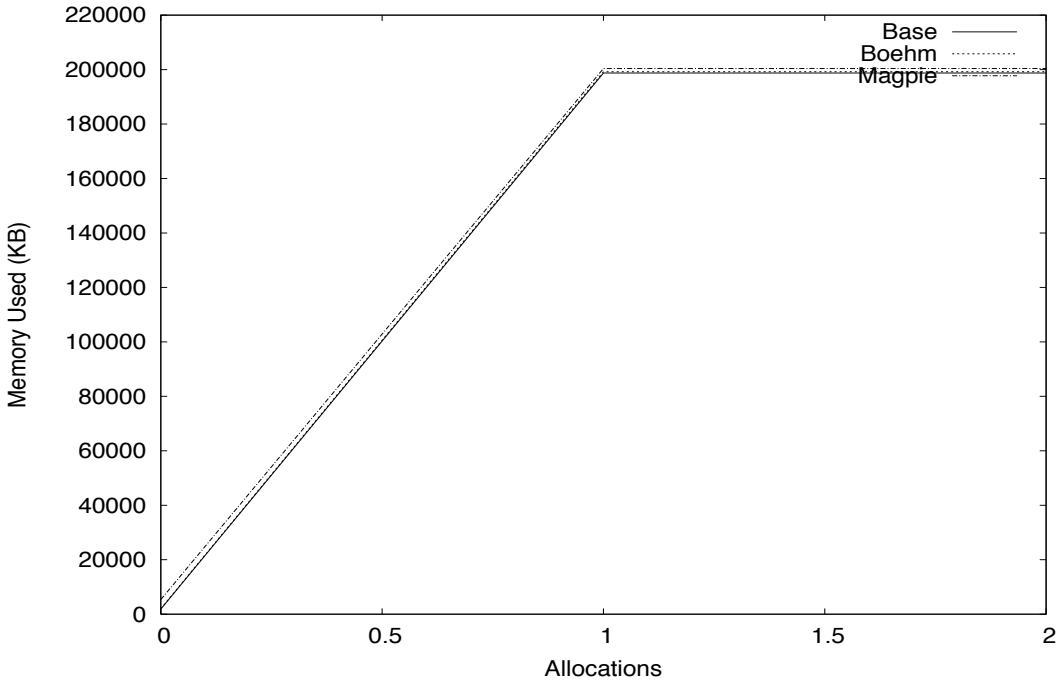

Figure 5.10. The memory behavior of the `254.gap` benchmark.

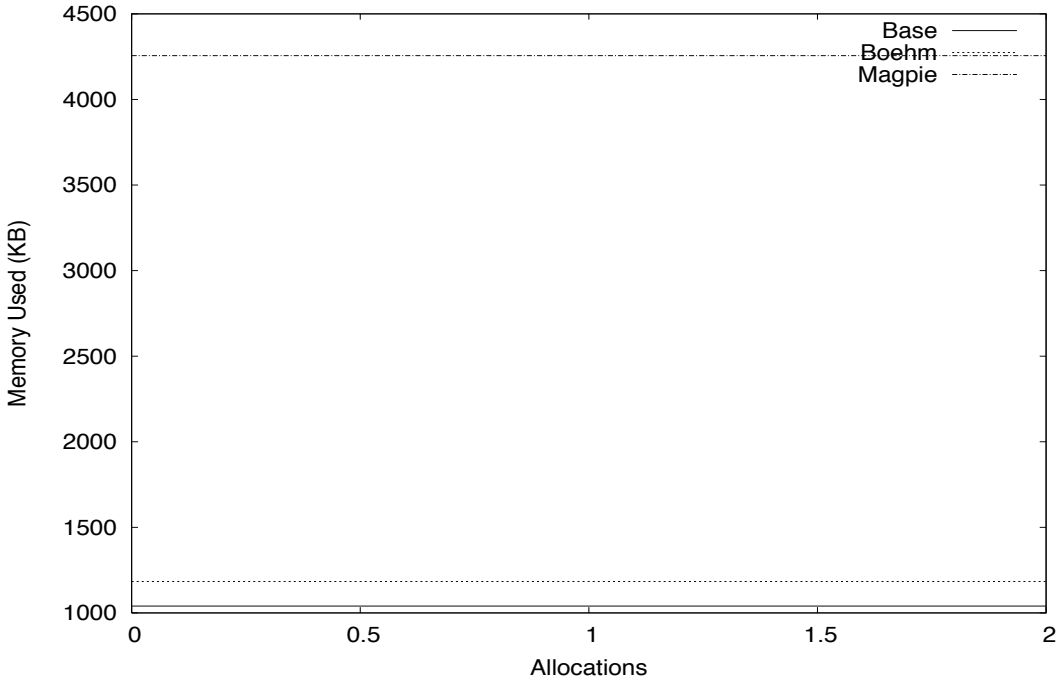

Figure 5.11. The memory behavior of the `256.bzip2` benchmark.

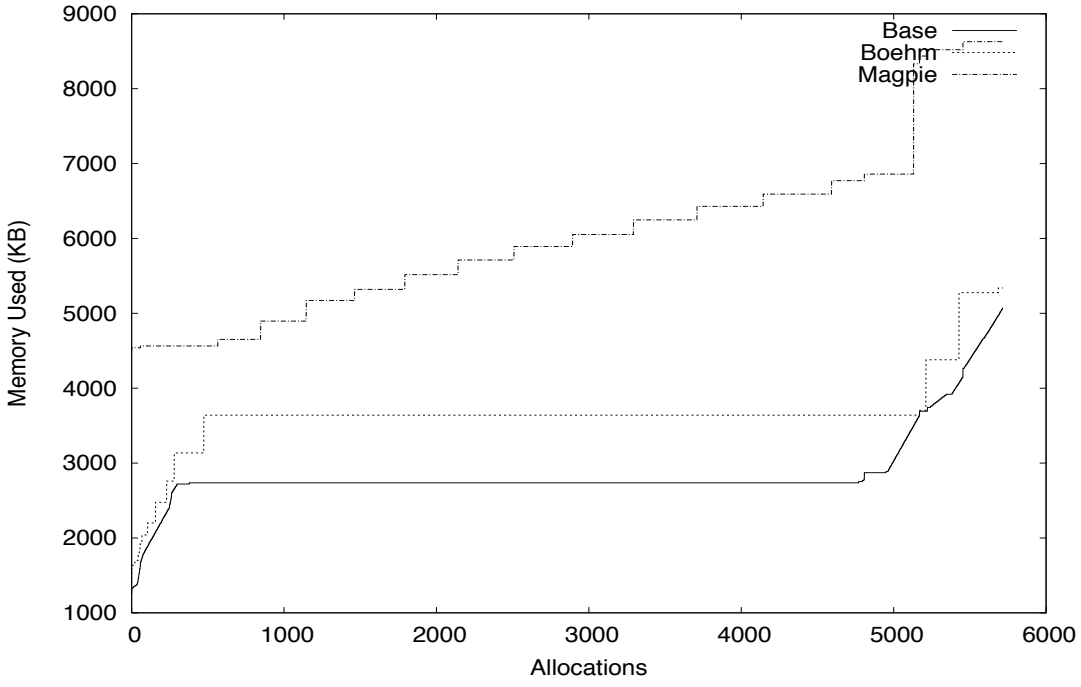

Figure 5.12. The memory behavior of the `300.twolf` benchmark.

and decompression phases show sharper increases in space use than the other two versions, as the converted program adds much of the encoding tree to the collector's autotag data structure.

5.5 Final Discussions on Space and Time

Garbage collection can be a win in space and time due to three factors:

- A different model for object deallocation costs.
- Faster allocation.
- Smaller object sizes and tighter object placement.

5.5.1 Object Deallocation Costs

The first is most noticeable in pure copying collectors. In such collectors, garbage collection time is strictly a function of reachable memory; the collector never iterates over deallocated objects. This is in contrast to manually managed programs, which must call a deallocation function on each deallocated object. In most cases, this deallocation function performs other operations beyond deallocating the block of memory. For example, if it discovers two adjacent deallocated blocks, it may combine them into a single unused block.

If the program mostly allocates short-lived objects, the garbage collector will have less heap space to traverse during garbage collection without having to call a deallocation function on each dead object. This can be a significant win in program performance.

However, most collectors are not pure copying collectors, and may need to perform some operations on dead objects in the heap. Whether this cost is higher or lower than the deallocation function for the manually managed version depends on the collector. Further, if the original program allocates mostly long-lived objects, then the original will spend little time in deallocation functions and the converted program will spend more time traversing live heap space.

5.5.2 Faster Allocation

Both BSD-based systems (such as Apple's OS/X) and GNU `libc`-based systems use variants of the Doug Lea allocator [29]. The basic structure behind this algorithm is a set of bins. Each bin holds a list of free memory blocks of a particular size; what sizes are used depends on the specific implementation. To perform allocation, the Lea allocator takes the requested object size, adds two words (see Section 5.5.3), and rounds to the nearest bin size.

If there exists a free block in the selected bin, the allocator removes the free block from the bin and uses it as the return value. If there is not, the allocator looks for a free block in bins with larger sizes. When it finds one, it splits the block into two free blocks; one of the correct size, and one of the original size minus the correct size. The left-over section of the block is added to the bin of that size.

In contrast, a garbage collector using a pre-allocated, fixed size nursery (such as the default Magpie collector) performs considerably less work. Since the allocator can rely on the garbage collector to deal with deallocation and fragmentation concerns, allocators in such systems simply return successive blocks of memory for each allocation. Thus, in the ideal case, an allocator for a garbage collected system uses only a few global variables (the start, end and first unallocated address in the nursery), a few mathematical operations, and a single conditional (to check if there is sufficient space left in the nursery) to perform an allocation.

Magpie's allocator does not reach this ideal speed. First, it allocates large objects outside the pre-allocated nursery, in order to lower copying costs during garbage collection. This requires an additional check. It also performs a two additional checks based on object size: one to check for zero-sized allocations, and one to enforce a minimum allocation size. Finally, the Magpie collector keeps metadata on the start and end of every block, and performs several loads and stores to track this information. This metadata is only required to deal with programs that create pointers into the interior of heap-allocated objects. Ideally, the garbage collector could be extended to remove this information — and the associated function calls — if the program uses only pointers to the first word of

every object.

Testing with a microbenchmark that allocates fifty million eight-byte objects in a tight loop suggests that the allocator with the default garbage collector is actually slower than `libc`'s malloc, by a factor of roughly two (14.18 seconds, compared to `libc`'s 7.04 seconds). Removing the operations noting the beginning and end of the object improves the performance of the collector considerably; 3.66 seconds for the allocator without these operations, again compared to `libc`'s 7.04 seconds.

5.5.3 Smaller Object Sizes

As noted in Section 5.5.2, derivatives of the Doug Lea allocator add two words (placed before and after the object) to the size of the object before selecting a bin. In contrast, the Magpie allocator adds a single additional word to the size of the object and does not use bins. The single word is used to associate the tag information with the object. If the object survives its first garbage collection and moves to the older generation, this word is removed. As noted previously, the older generation segregates objects by tag, so the tag information is associated with an object's page rather than the object itself. Finally, all implementations of `malloc` that I have encountered have a minimum allocation size of 16 bytes. Magpie enforces a minimum object size of eight bytes in the older generation and twelve bytes in the nursery.

The removal of the extra word(s) and avoidance of object allocation bins results in tighter placement of objects in the heap. This, in turn, can result in better cache behavior, since it is more likely that a cache access to one object will also cache all or part of the previous or succeeding object.

However, while Magpie's garbage collector places objects nearer to each other, it does so with a considerably increased metadata cost. Some of these costs are fixed; for example, Magpie uses a fixed-size hash table mapping pointers to their associated page information structures. Other costs are functions of the heap size. Magpie uses a structure holding metadata for each page in the heap, as well as using a bitmap to store information about each word in the heap. Currently, the former requires 32 bytes per page in the heap, while the latter uses two bits per

word in the heap.

CHAPTER 6

EXPLOITING PRECISE GC: MEMORY ACCOUNTING

As applications grow increasingly complex, they are increasingly organized into smaller subprograms. For example, web browsers invoke external programs to display images and movies, and spreadsheets frequently execute user-defined scripts. The more subtasks that an application invokes the more things can go wrong, and the more it becomes useful to control the subtasks. In this section, I concentrate on the problem of constraining memory use.

Applications currently restrict memory use by partitioning data and then limiting the memory use of the partitions. Traditional operating systems partition memory into completely separate heaps for each process, disallowing references between them. This strict partitioning makes interprocess communication difficult, requiring the marshaling and demarshaling of data through pipes, sockets, channels or similar structures. In some cases, marshaling important data proves infeasible, leading to the use of complicated protocol programming.

More recent work provides hard resource boundaries within a single virtual machine. Systems in this class, such as the KaffeOS virtual machine [3], JSR-121 [47] or .NET application domains [32], still partition data, but without the separate address space. Generally, the programmer explicitly creates a shared partition and may freely allocate and reference objects in it. However, these systems place restrictions on inter-partition references. For example, KaffeOS disallows references from its shared heap to private heap space. This less strict form of partitioning only partially alleviates the burden on the programmer. Although the program may now pass shared values via simple references, the programmer must explicitly manage

the shared region. In the case where one process wants to transfer data completely to another process, the transfer may require two deep copies: one to transfer the data into shared space, and the other to transfer it out. In short, the programmer must manually manage accounting in much the same way a C programmer manages memory with *malloc*() and *free*().

My system of *partition-free* memory accounting provides the accounting ability of partitions without unnecessary work by programmers. Further, as a *consumer-based* system, programmers simply allocate and pass objects around as they please, and the current holder of the object is charged, rather than the allocator. Thus, data migration and sharing require no additional work on the part of the programmer: no marshaling, no complex communications, and no explicit management of partitions. By leveraging an existing garbage collector and the thread hierarchy, the system is flexible enough to handle most memory accounting demands, and is fast enough to be used in production quality software. Finally, the system exports simple but reliable guarantees, which are easy for the programmer to reason about.

Although Price et al. [41] address memory accounting in a way similar to my system, they do not build on a process hierarchy, which is a cornerstone of my work. I believe that a consumer-based accounting mechanism must be tied to a process hierarchy to provide useful guarantees to the programmer. My practical applications—as well as my comparison to other accounting mechanisms—both depend crucially on parent–child relationships among processes.

I begin the discussion of this work in Section 6.1 with example applications where partition-free, consumer-based memory accounting saves valuable programmer time and energy, and present the details of the accounting system in Section 6.2. Section 6.3 describes how I use this system in the examples in section 6.1. Section 6.4 outlines how the system works over large, general classes of applications. I conclude with the implementation and performance effects of the system in Section 6.5 and a comparison of this system with other work in Section 6.6.

6.1 Motivating Applications

Conventional partitioning or producer-based accounting techniques can be used to solve most accounting problems. However, these techniques often make programming difficult for no reason. I present three applications in this section where using consumer-based, partition-free memory accounting greatly simplifies our implementation task.

6.1.1 DrScheme

The DrScheme programming environment consists of one or more windows, each split into a top and bottom half. The top half provides standard program editing tools to the user, while the bottom half presents an interactive Scheme interpreter. Normally, users edit a program in the top half and then test in the bottom half.

As a development environment, DrScheme frequently executes incorrect user code, which must not make DrScheme crash. Language safety protects DrScheme from many classes of errors, but a user program can also consume all available memory. Thus, DrScheme must account for memory used by the user program, and DrScheme must terminate the program if it uses too much.

As previously noted, the usual way to account for memory use and enforce memory limits is to run the interpreter in a completely separate heap space. However, this separation requires significant programmer effort. DrScheme would have to communicate with its child processes through pseudovalues rather than actual values and function calls, much like a UNIX kernel communicates with processes through file descriptors rather than actual file handles. This approach becomes particularly difficult when writing DrScheme tools that interact with both the program in the editor and the state of the interpreter loop, such as debuggers and profilers.

DrScheme could partition values without resorting to separate heaps, but partitioning only solves some problems. The programmer can either keep the entire system within one partition, or split the system into many partitions. Neither approach works well in practice:

- Allocating all objects into a single, shared partition makes communication simple, as references may be passed freely. However, DrScheme frequently invokes two or more interpreter loops. Because every object resides in the same partition, the accounting system could give no information about which interpreter DrScheme should kill if too many objects are allocated.
- Separating DrScheme into several partitions leaves no natural place for shared libraries. Shared libraries would either need to be duplicated for every partition, or must be told into which partition to allocate at any given time. The first — duplication — has obvious problems. The second amounts to manual memory management, requiring unnecessary time and effort on the part of the programmer.

A producer-based accounting scheme suffers from similar problems. To write simple APIs, the program in test often invokes parts of DrScheme to perform complex operations. Some of these operations allocate data, and then return this data to the child process. In this case, a producer-based system would either not honor the data hand-off or would require the parent process to communicate to the accounting system that it is allocating the data on behalf of another process. In the latter case, security becomes an additional problem.

Instead of developing complex protocols to deal with separate address spaces or partitions, the system allows a direct style of programming, where a small addition to DrScheme provides safety from overallocating programs.

6.1.2 Assignment Hand-In Server

Students in CS2010 at the University of Utah submit homework via a handin server. The server then tests the student's program against a series of teacher-defined tests.

This server clearly requires memory constraints. First, an incorrect program that allocates too much memory on a test input may kill the entire handin server. Second, and unlike DrScheme itself, a malicious student might attempt to bring down the server by intentionally writing a program that allocates too much memory.

Running the student program interpreter in the same process as the server saves a great deal of programming work, and saves the course instructor time by not requiring test case input and results to be marshaled. Further, we avoid problems duplicating and reloading libraries. Duplication, for example, creates a problem for test cases that use library data types, because the types in a student program would not match test cases generated by the testing process. Reloading becomes a further problem for advanced student projects, which typically require large support libraries. Reloading these libraries for every test may stress the CPU beyond acceptable levels, particularly near assignment due dates.

Partitioned accounting schemes solve some of these problems, but not all of them. As in DrScheme, shared libraries are problematic, requiring either a great loss of information or a form of manual memory management. Again, as in DrScheme, producer-based accounting schemes fail when shared libraries allocate memory on behalf of a student program.

Instead of managing such details, the server uses partition-free, consumer-based accounting and works with no special protocols. The server requires neither copying code nor protocol code, and shared libraries are loaded once for the entire system.

6.1.3 SirMail

SirMail began as a modest mail client, and was gradually extended with a few additional features, including HTML rendering and attachment processing. These two extensions, however, introduce a memory security problem.

The HTML engine renders HTML email in the normal way. In doing so, however, it may download images from unknown, untrusted servers. By creating a small email that requires the display of an arbitrarily large graphic, an attacker (or spammer) could easily bring the whole system to a grinding halt. Attachment processing also presents a problem, as a huge attachment may arbitrarily pause the whole mail client for the duration of processing.

Partitioned accounting systems solve both these problems but create new ones. To function the HTML rendering engine must draw to the main SirMail window. Similarly, SirMail must pass the data to be processed to the attachment proces-

sor, and then receive the result. Partitioning schemes cause problems for both interactions. Because shared memory regions may not contain pointers to private memory regions, for the HTML engine to draw to the screen either SirMail must place the entire GUI (and all callbacks and other references) into a shared region or the engine must communicate to SirMail using some drawing protocol. In the case of the attachment processor, SirMail would have to copy the entire attachment into and then out of the subprocess's heap for accounting to make sense.

Using partition-free accounting solves both these problems. SirMail simply passes a direct reference to the GUI to the rendering engine, which can then interact with it in the usual ways. Similarly, SirMail may simply call the attachment processor with a reference to the data, and have a reference to the result returned to it, requiring no copying.

6.2 Consumer-Based Accounting

Without sharing, accounting is a simple extension of garbage collection. Sharing complicates matters, Because the system must provide guarantees as to which process or processes the shared objects will be accounted. In other words, if thread A and thread B both share a reference to x, the accounting system must provide some guarantee as to how the charge for x will be assigned.

Certain policy decisions would provide useful guarantees, but drastically reduce performance. One such policy is to charge the use of x to both A and B, and another might split the charge equally between A and B. Although these policies provide reliable, intuitive guarantees, they both suffer from a performance problem: a worst-case execution time of $C * R$, where C is the number of threads and R is the number of reachable objects. Experience suggests that this extra factor of C scales poorly in real systems, and provides a new opportunity for denial of service attacks (i.e., by creating many threads).

My approach makes one simple guarantee to the programmer that nevertheless provides useful information. I guarantee the charge of x to A, and not B, if A descends (hierarchically) from B. Due to the hierarchical nature of threads,

these charges bubble up to the parents, regardless, so assigning the charge to the child provides more useful accounting information. In other words, because B is responsible for the behavior of A, B eventually becomes charged for A's usage. In the case of unrelated sharing threads, the charge is assigned arbitrarily. For reasons that we discuss at length in the following section, I have not found the arbitrary assignment a problem. In fact, I find that this simple approach applies well to a large class of potential applications.

Finally, the policy must describe what it means for a thread to reference an object. The system considers an object reachable by a thread if it is reachable from any thread that thread manages, with a few exceptions. First, because weak references do not cause an object to be retained past garbage collection, a thread holding an object only through a weak reference is not charged for it. Second, many threads hold references to other threads. However, these references are opaque, so the original thread cannot then reference anything in the referenced thread. Therefore, if an object x is reachable by thread C only through a reference to some other thread, then C is not charged for x.

Given these guarantees and policies, we export the accounting API as follows:

- `thread-limit-memory(thread1, limit-k, thread2)` installs a limit on the memory charged to the thread `thread1`. If ever `thread1` uses more than `limit-k` bytes, then `thread2` is shut down.

  Typically, `thread1` and `thread2` are the same thread, and the parent thread uses the child thread in question for both arguments. Distinguishing the accounting center from the cost center, however, can be useful when `thread1` is the parent of `thread2` or vice-versa.

Although `thread-limit-memory` is useful in simple settings, it does not compose well. For example, if a parent process has 100 MB to work with and its child processes typically use 1 MB but sometimes 20 MB, should the parent limit itself to the worst case by running at most 5 children? And how does the parent know that it has 100 MB to work with in the case of parent-siblings with varying memory consumption?

To address the needs of a parent more directly and in a more easily composed form, we introduce a second interface:

- `thread-require-memory(thread1,need-k,thread2)` installs a request for `need-k` bytes to be available for thread `thread1`. If it is ever the case that `thread1` cannot allocate `need-k` bytes, then `thread2` is shut down.

Using `thread-require-memory`, a parent process can declare a safety cushion for its own operation but otherwise allow each child process to consume as much memory as is available. A parent can also combine `thread-require-memory` and `thread-limit-memory` to declare its own cushion and also prevent children from using more than 20 MB without limiting the total number of children to 5.

All these procedures register constraints with the accounting system separately. Thus, a child processes cannot raise a limit on itself by simply reinvoking *thread-limit-memory* as both limits remain in effect. Furthermore, note that killing a thread simply closes all its ports, stops all its threads, and so on, but does not necessarily explicitly deallocate memory. Because of this, memory shared between a thread being shutdown a surviving thread will not be deallocated by the shutdown process.

In addition to the two memory-monitoring procedures, I also export a function that reports a given thread's current charges:

- `current-memory-use(thread)` returns the number of allocated bytes currently charged to thread `thread`.

These procedures, in combination, provide simple mechanisms for constraining the memory use of subprocesses. In most cases, extending an existing application to use memory constraints requires only a few additional lines of code.

6.3 Accounting in the Examples

Using the new accounting mechanisms, I easily solved the accounting problems described in Section 6.1. In this section, I report briefly on our experience.

6.3.1 DrScheme

To support a stop (break) button, DrScheme runs every interpreter window in a separate thread. These threads descend from a single, parent thread for the system. Because the accounting mechanism charges shared memory to the child, rather than the parent, none of the existing interfaces required updating. DrScheme simply allocates complex objects and transfers them directly to the child. Additionally, the DrScheme process has direct access to the closures, environment, and other interpreter state of a user program.

The initial pass to add accounting to DrScheme required only four lines of code. Specifically, the four lines locally bind the new thread, set the memory limit, and proceeded as normal.

However, a problem quickly became evident. The system, as originally constructed, contained reference links from the child thread to the parent (through ports, GUI objects and so forth passed down through the API), but also links from the parent down to each of the children. The accounting pass, then, picked one of the children to account to first, and that child was charged for most of the objects allocated in the entire system. Because the child can reach up to the parent's object space and then back down to all its siblings' object spaces, it can reach (and thus is charged for) all these objects.

Initially, we attempted to break all the links from the child to the parent. However, doing so creates many of the same problems as the old operating system process solution. For example, instead of handing out I/O ports directly, a file handle system must be used.

Rather than rewriting a huge chunk of DrScheme, we investigated breaking the links from the parent to the child. This modification turned out to be quite simple, involving only a few hours of programmer time to find the links from parent to child, plus another half hour to remove these links. In all, the task required the changing of five references: two were changed to weak links, one was pushed down into the child space and the final two were extraneous and simply removed.

6.3.2 Hand-In Server

To make the hand-in server shut down overconsuming test threads, a single line was needed. Another programmer then added an additional feature to report to students when the test was terminated. Even this extra feature proved fairly simple, with the entire change comprising roughly 25 lines.

6.3.3 SirMail

Modifying the existing code to limit the memory use of the HTML rendering engine required about 45 minutes of time from a programmer unfamiliar with the SirMail code base, and about 20 lines of code. Most of this additional code detects and reports when the accounting system shuts down the rendering thread.

The MIME processing modifications turned out to be easier, requiring approximately 10 minutes of time and an additional 5 lines of code. These five lines implement a pattern for libraries and callbacks that is described in 6.4.4.

6.4 Accounting Paradigms

In this section, I describe several common paradigms for multiprocess programs, and show how my system easily handles most of them. Despite the apparently weak guarantee for shared-object accounting, in many cases the accounting system improves on existing mechanisms. Figure 6.1 shows a graphical depiction of three different communication paradigms.

6.4.1 Noncommunicating Processes

In some cases, a program must offload some work and does not care about the results. In such examples, the parent spawns the child with some initial data set, and then largely ignores the child unless it raises an error. Examples include print spoolers, nonquerying database transactions (inserts, updates, etc.), and logging utilities. This protocol roughly matches traditional operating system processes.

Conventional process accounting suffices in such tasks, due to the small amount of one-way communication between the two processes, but my system works equally

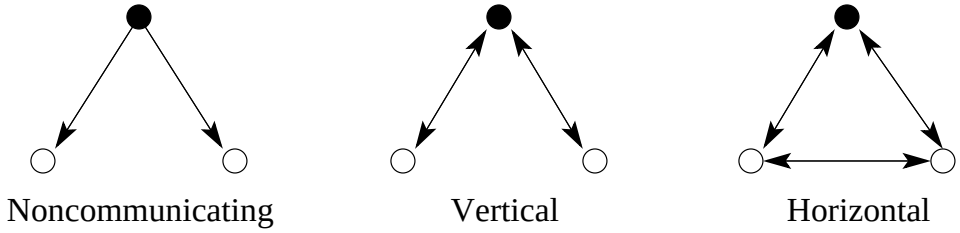

Figure 6.1. The three interprocess communication patterns. The filled circle represents the parent process, with the hollow circles representing child processes. The arrows represent directions of communication.

well. Because the data in the subprocess never escapes that subprocess, any data that the subprocess uses is charged to it.

6.4.2 Vertically Communicating Processes

Another common paradigm involves two-way communication between a parent process and a child process, but not between child processes. Examples include web browsers, programming language IDEs, file processors, database queries and so on. In these cases, the parent process may create an arbitrary number of children, but these children communicate only with the parent and not each other.

Such purely vertical communication paths represent a large subset of concurrent, communicating programs involving untrusted processes. Generally, the parent program runs the untrusted process and communicates with it directly. Communication between other subprocesses and the untrusted process usually pass through the parent program. Meanwhile, the parent and each child must cooperate closely, and this cooperation is most easily implemented by sharing closures and objects.

My algorithm for accounting memory clearly covers this case. As my accounting mechanism always accounts shared objects to the child, information on the memory usage of the child remains exact. Thus, by leveraging the thread hierarchy, I create a framework providing exactly what the programmer wants and needs. In general, I find the system allows applications in this class to restrict the memory use of their subprocesses with few to no changes in the way they allocate and share data.

6.4.3 Horizontally Communicating Processes

Sometimes, a parent spawns multiple children that communicate directly. Examples include AI blackboard systems and parallel sorting algorithms. In such cases, the children work collaboratively to solve some problem.

When two sibling processes share data, my algorithm guarantees only that one will be charged, but does not guarantee which. Therefore, on one garbage collection, assignment of the charge may go to one process and on some other during the next. This uncertainty in charging reflects that, typically, individual charges make little sense and that killing one process will not allow others to continue. In that case, children can be grouped under a grouping thread, which lies between the parent thread and the child threads in the hierarchy. The parent then sets limits on the set of children, and shuts them down as a group if they violate any memory constraints.

Another possibility is that the children may be working on disjoint sets of data, so charges make sense for each process and, presumably, individual children can proceed even if others die. In these cases, a limit on the children as a group makes little sense. However, the parent may set a memory requirement, to guarantee that a certain amount of memory is held in reserve for itself. When a violation of this requirement occurs, the program simply kills off the subprocesses in some static or heuristic order.

6.4.4 Libraries and Callbacks

The previous sections concentrate on concurrent subprocesses, where consumer-based accounting makes sense. In some cases, this assumption does not apply. The first case that I consider involves untrusted libraries or callbacks installed by untrusted code. The second case involves situations (concurrent or not), where the application requires producer-based accounting.

Some applications link (statically or dynamically) to untrusted libraries. Further, some concurrent applications install callbacks from child processes into the parent process. In these cases, the desired granularity for accounting more closely resembles a function call than a thread or process. These cases require wrappers to the API described previously.

In most cases, an additional function call or the introduction of a macro suffices. These convert the original function call into a series of steps. First, the new code creates a new thread to execute the call and sets the appropriate limit upon it. In short, a function call is converted into a short-lived subthread with appropriate constraints.

A disadvantage of this approach involves the creation of the temporary thread, which involves an obvious performance penalty. However, it seems unlikely that converted calls will appear in tight loops. (Calls in tight loops typically execute quickly, and thus are unlikely to have behavior requiring memory constraints.)

Even if speed is not an issue, functions requiring thread identity will not work using this method. For example, security mechanisms might grant privileges only to one particular thread, but not subthreads of that thread. By running the function in a new thread, the system loses any important information stored implicitly in the original thread.

6.4.5 Producer-Based Accounting

The only situation in which our system does not clearly subsume existing technologies is when producer-based accounting is required. In other words, process A provides data to process B, and charges for the data should always go to A, regardless of the relationship between the two.

I have not encountered in practice an example where producer-based accounting makes sense, and I conjecture that they are rare. Even so, this problem might often reduce to a rate-limiting problem rather than a memory usage problem. In other words, B does not so much wish to constrain the memory use of A as much as it wants to constrain the amount of data that it receives at one time. Suitable rate-limiting protocols should suffice for this case. Further, using a weak reference allows a program to hold data without being charged for it. Because weak references do not cause the data contained in them to be retained by the collector, the accounting system does not charge their holders with their data.

Another possible scenario involves the use of untrusted libraries for creating and maintaining data structures. I can imagine using an off-the-shelf library to

hold complex program data, rather than writing the data structures from scratch. In using such a library, I can imagine wanting to ensure that its structures do not grow unreasonably. Again, this situation seems unlikely. Programs typically do not trust important data to untrusted data structure libraries. Further, it is unclear how the program would continue to run after a constraint violation kills its own data structures.

6.5 Implementation

Implementing consumer-based accounting requires only straightforward modifications to a typical garbage collector. My approach requires the same mark functions, root sets and traversal procedures that exist in the collector. The primary change is in the organization and ordering of roots. A second involves the method of marking roots, and the final involves a slight change in the mark procedure.

First and foremost, accounting requires a particular organization of roots. Before accounting, the roots must be ordered so that root A appears before root B if the thread responsible for A descends from the thread responsible for B. Second, the accounting system marks and fully propagates each individual root before moving on to the next one. Figure 6.2 outlines these steps.

By placing this partial order on the roots and forcing full propagation before moving on to the next root, the system provides the full guarantee described previously. If object x is reachable by roots from threads A and B, where B is a descendent of A, then the mark propagation will select B's root first due to the partial order. When the root or roots associated with A come around, x has already been marked and no further accounting information is gathered for it.

Minor modifications to the mark procedure alleviate one further potential problem. These stop mark propagation when threads and weak boxes are reached to match the behavior outlined in Section 6.2.

Doing this process alongside the collector allows one pass to perform both accounting and collection, and works in many cases. My first implementation of memory accounting used this methodology. However, in cases where the set of collectable objects include threads, creating appropriate orderings becomes dif-

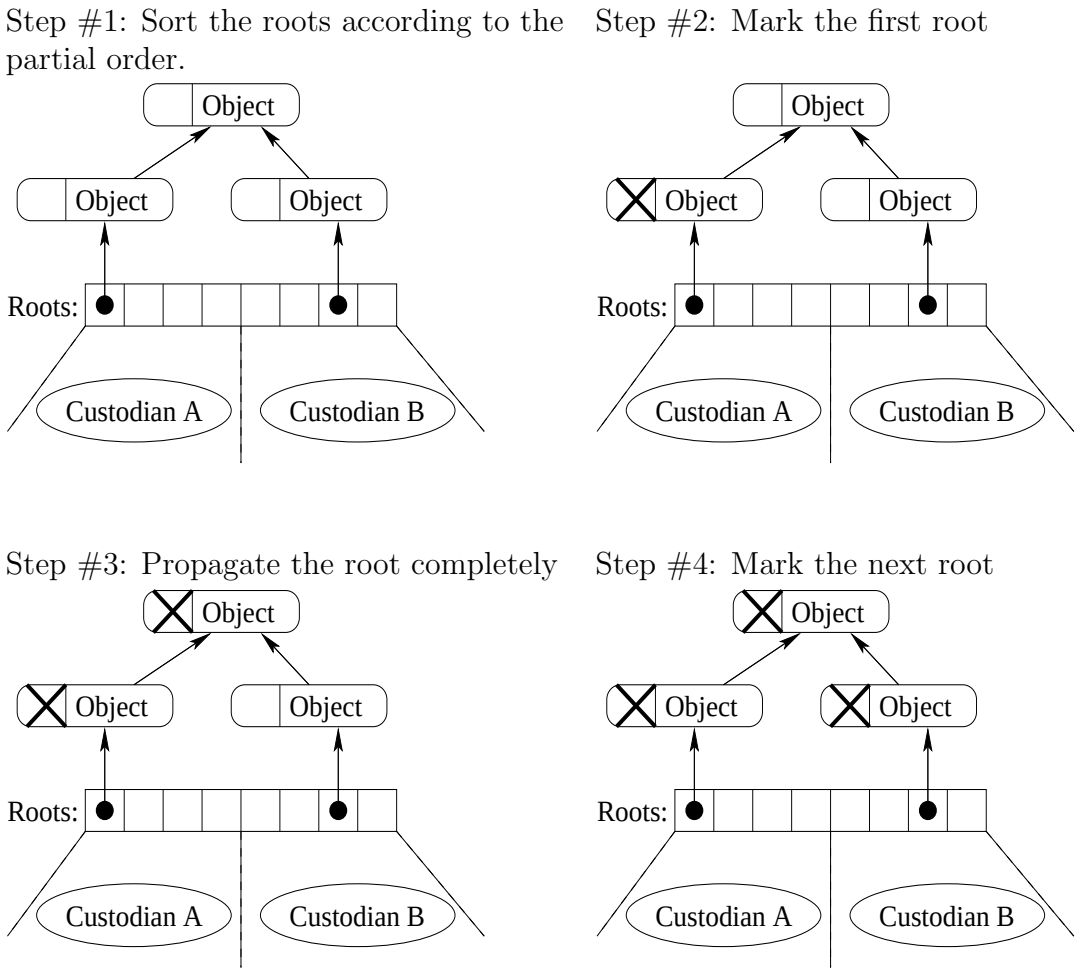

Figure 6.2. The four steps of the accounting procedure.

ficult. Because the accounting mechanisms requires the marking of threads and the collector does not know which threads are live, determining how to proceed becomes tricky.

In such cases, a second pass implementing memory accounting suggests itself. This second pass occurs after garbage collection but before control returns to the mutator. My implementation of such a system suggests that the additional effort required above that of the all-in-one solution remained minimal. Again, using much of the existing collector scaffolding saves considerable amounts of work and time. This two-pass style may increase slowdowns noticeably in some cases, however.

6.5.1 Incremental Collection

My implementation of memory accounting builds on an existing precise, stop-the-world garbage collector. To see the problem in combining incremental and accounting collectors, consider the example in Figure 6.3. In this case, we have two threads (A and B) and four objects (w, x, y and z), and two of the objects — w and x — have been marked by the collector. If the collector now turns control over to the mutator, the mutator may then modify the heap so that the link from y to z is destroyed and a new link from x to z is created. At this point, an unmodified incremental collector will account z to B, which violates the guarantee in Section 6.2.

In incremental collectors, the collector uses either read barriers or write barriers to prevent the mutator from putting references from unscanned (*white*) objects to scanned (*gray* or *black*) objects [1, 54]. Using a write-barrier technique, the collector must either *gray* the referring object (scheduling it for repropagation once the collector resumes) or gray the referred-to object (forcing it into scanned space).

In both cases, any references by the newly-grayed object will be placed in the collector's mark queue. To support accounting, the garbage collector requires two modifications. First, the trap routine must place these objects at the head of the mark queue, causing them to be marked immediately once the collector resumes.

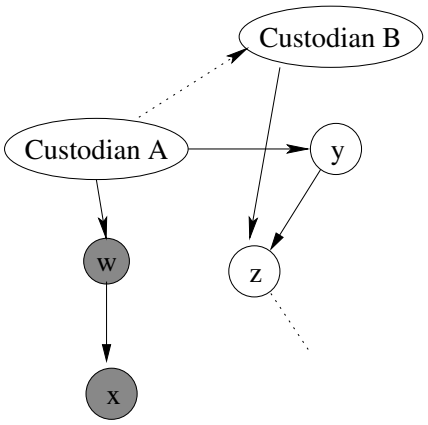

Figure 6.3. A potential heap layout, mid-collection. The grayed objects have been marked by the collector.

Second, the trap routine must annotate these objects with the thread to which the referring object was accounted. By combining these, the collector accounts these objects to the appropriate thread immediately after it resumes execution.

Because these objects are put at the head of the queue, there is no chance of a parent reaching these objects before its child, thus insuring the guarantee described in Section 6.2. Because the modification simply modifies the order of objects added to the queue in the trap function, a modified version of an incremental algorithm will halt if the unmodified version halts.

6.5.2 Performance

An implementation of this memory accounting system was created for Mz-Scheme [20], an interpreter for the programming language Scheme. Performance tests on microbenchmarks resulted in a worst-case execution hit of 13%, with most programs slowing down by a factor of 4% to 8%. Experience with larger programs suggests similar slowdowns. [51]

6.6 Comparisons to Existing Systems

I previously reported preliminary results for our accounting system [52] and followed that with final results in a later report [51]. This section of the document is a detailed overview of the latter.

Other recent research focuses on providing hard resource boundaries between applications to prevent denial of service attacks. The KaffeOS [3] for Java provides the ability to precisely account for memory consumption by applications. MVM [13], Alta [4], and J-SEAL2 [7] all provide similar solutions, as do JSR-121 [47] and .NET application domains [32], but in all cases these boundaries constrain interprocess communication. In highly cooperative applications, or in situations requiring some amount of dynamic flexibility in sharing patterns, these systems may present significant barriers to simple development.

Generally, the existing work on resource controls — including JREs [14] and research on accounting principals in operating systems address only resource appli-

cation, which does not adequately provide the full range of tools we believe modern programmers require.

Price et al. [41] present the only other consumer-based, partition-free accounting system, which was developed in parallel with my system. They present no results for practical applications. Moreover, their policy for shared objects is to rotate the mark order for roots, which would not provide a sufficient guarantee for any of our motivating applications. Price et al. also present a concept of "unaccountable references." These references effectively block the marking phase of the accounting mechanism. The rationale for these objects is to block a malicious process A from passing an inordinately large object to process B in an attempt to get it killed. It is unclear, however, what advantages unaccountable references have over normal weak references.

CHAPTER 7

CONCLUSIONS

The principle contribution of this work is the design and implementation of an easy-to-use, highly applicable tool for transforming C to use precise garbage collection that does not impose extraordinary penalties in programmer time, execution time or space utilization. In most cases, Magpie performs within 20% (faster or slower) than the original C code and requires no more effort than the existing Boehm collector. The memory use of Magpie-converted code similarly tracks the Boehm collector, and in some cases removes memory spikes created by Boehm.

Magpie reaches fast post-parsing compilation speeds by using only a series of linear analyses and conversions. Except for the call graph analysis, none of the analyses or conversions are global, which allows for the use of Magpie on programs during development.

Magpie achieves reasonable execution times due to several factors. First, Magpie transforms the original source only to do garbage collection; it does not introduce bounds checks, cast safety checks, or similar tests traditionally performed in type-safe languages. Second, Magpie's garbage collector streamlines object allocation, and attempts to layout objects as compactly as possible. This allows for better cache utilization than the default manual memory management system in C language runtimes. Finally, Magpie performs several optimizations in order to minimize the effects of the transformation.

Magpie inhibits space leaks by transferring the responsibility for memory management to the computer. Thus, incorrect deallocations and many memory leaks will not occur due to programmer error in the creation or utilization of memory management protocols. Further, since Magpie transforms the C to use precise

garbage collection, Magpie-converted code will not introduce space leaks due to conservative approximations of what is or is not a pointer or root.

7.1 Magpie for C/VM Interfaces

The primary focus of this dissertation is on the use of Magpie for application programs. However, Magpie presents a major opportunity for virtual machine authors. Many virtual machines — Microsoft's CLR [32], the Java Virtual Machine [31], the PLT [20] and Chez [18] Scheme implementations, the Python [17] virtual machine, and many more — allow programmers to create arbitrary virtual machine extensions using C. While these *native interfaces* vary widely, all of them allow programmers to use a high-level language for most of the program, but still use C if their program requires tight coupling with a low-level library, or complete control over program execution or the layout of some data structure.

However, the ability to write such extensions creates problems if the virtual machine uses garbage collection. As with Magpie when dealing with foreign libraries, these virtual machines must be careful when allowing references to objects to pass between the virtual machine and the C extension. When an extension passes an object back to the virtual machine, the virtual machine must ensure that it can identify the type of that object and discover exactly which words in the object are references. Perhaps the more difficult problem occurs when the virtual machine passes an object to the extension. In this case, the garbage collector must be aware of any references saved within the extension, so that it can ensure that the object is not collected and update references should the object move.

Existing implementations of interfaces to unsafe code either provide no protection against garbage-collector corruption by the C extensions, or use one of several heavyweight techniques to ensure that the collector behaves correctly. In the latter case, the techniques used can create an excessive amount of work for the programmer, create significant efficiency and implementation requirements for garbage collectors, or both.

For example, the native interface for Java (JNI) uses a layer of indirection

between C extensions and objects in the garbage-collected heap. C code wishing to access fields and methods within an object must use accessor functions, rather than referencing the fields and methods directly. In the case of method calls this can be particularly obnoxious, as it requires the use of a primitive to obtain the method signature and then the use of a second primitive to invoke the function. For example, if the C code wishes to invoke method `f` on object `o`, which takes a string as input and produces a `double`, the C code cannot simply invoke the function as follows:

```
result = o->f(str);
```

Instead, the C must perform two, sequential statements [48], where `cls` is the class of the object `o`.:

```
jmethodID mi = env->GetMethodID(cls, "f", "(ILjava/lang/String;)D");
result = env->CallDoubleMethod(o, mi, 10, str);
```

In some cases, the JNI requires *object pinning*. Like the object immobility in Magpie (see Section 4.6), object pinning inhibits the collector from moving an object in the heap. This creates additional complications in implementing the collector, and may inhibit the use of certain garbage collection styles that require object mobility. Microsoft's Common Language Runtime, which allows the mingling of safe and unsafe code within the same virtual machine, uses a similar technique. In C#, when programmers wish to access garbage-collected objects from unsafe code, they must first pin the objects using the `fixed` construct. In all other cases, C# forbids the use of pointer variables referencing managed (garbage collected) objects [33].

Magpie could simplify these interfaces by providing a simple mechanism to inform the virtual machine's garbage collector of references within the unsafe code. In its conversion, Magpie generates all the information required to identify objects referenced within the unsafe extension, as well as identifying all the references within an object created within the extension. Thus, programmers could more

naturally reference garbage-collected objects or create new objects for use within the virtual machine.

The only facility not provided by Magpie is type safety. In such a system, programmers could write pointers to integer fields within an object, or, in general, use object fields in ways disallowed by their stated types. To what degree this functionality is desirable depends on the virtual machine's security model. However, some virtual machines disallow the use of unsafe extensions *a priori* when security is of paramount importance. Further, it may be possible to extend Magpie to restrict some unsafe behavior should the security model demand it.

7.2 Annotations vs. GUIs

For the most part, Magpie uses GUIs to gather information about the program from the user. It then saves the information gathered into a separate persistent store, outside of the program source. This is a choice of interface, rather than an issue at the core of the Magpie process. Magpie could have used an annotation system, instead, with no changes to the core Magpie algorithms described in Chapters 2, 3 and 4.

Using a GUI and an offline store solves four problems: parsing annotations placed in the original C source, learning the syntax of the annotations, dealing with system libraries, and discovering where in the code to place the annotations.

In Magpie, annotations in the source would be required at variable declarations, field declarations and allocation points. In the first two cases, parsing the annotations would be simple. The annotations could simply placed in the same place as additional flags on a type (such as "unsigned"), without extensive modifications to an existing parser. The last case, however, may be difficult. Programs typically use facades over the underlying `malloc` function to perform additional error handling. If these facades are functions, then it may be possible to use an extended cast form to annotate the function calls. However, if the facades are macros, then they may expand to statement forms rather than expression forms. Parsing these cases may require extensive modifications to an existing C parser.

Secondly, Magpie would require a robust annotation syntax. While, in some cases a simple flag would suffice (`__is_pointer__` or `__is_array_of_pointers__`, for example), in many cases Magpie requires significantly more information: array sizes and type information, for example. In more involved cases, Magpie may require arbitrary user code. Learning such a robust syntax may be daunting to users. However, since Magpie uses a GUI to interact with the user, these details are hidden from the user.

Thirdly, in some cases, Magpie requires information about structures defined in system header files. Typically, users cannot edit these files. In Magpie, information about structures in these files is kept offline, in a separate file, and thus Magpie has no problem converting read-only files. An annotation-based system, would either require that system administrators allow users to edit these files, or require the programmer to copy the files, edit them in user-writable space, and modify all references to the files.

Finally, annotation-based systems typically generate warnings or errors when the system determines that it requires an annotation that is not present. In the case of Magpie's allocation analysis (see Section 4.1), this would not present a problem. Each allocation point is independent of all other allocation points, so a single analysis pass can determine every point requiring annotations.

For Magpie's structure analysis (see Section 4.2), however, whether or not an annotation is required may depend on the existence of other annotations. For example, if field foo is annotated as a structure of type bar, then annotations are required for all the fields of bar. If, however, foo is annotated as a simple pointer, then annotations may not be required for the fields of bar. In an annotation-based system, this would require repeated invocations of the analysis to ensure that exactly the required annotations exist, with the associated reparsing and reanalysis costs. Since Magpie's GUI interface is interactive, however, this information can be gathered by need without reparsing or reanalyzing the entire file.

Unfortunately, using an offline data store complicates the handling of incremental changes in the program source. Annotations will survive this process easily, and

interact well with existing source repository tools. An offline data store, however, is difficult to manage with source repository tools, as well as requiring additional code to reassociate gathered information with the source should the programmer modify the program.

A combination of the two approaches may be ideal. In such a system, an interactive GUI would find where annotations were required, ask the user for information about those annotations, generate the annotations, and then place the annotations in the original source. This solves the problems of finding places that require annotations, learning the annotation syntax, and dealing with incremental updates. If human readability is not required, it may also solve the problem of parsing. It does not, however, solve the problem of read-only files.

7.3 Future Work

Magpie is a work in progress. The current version is a functional prototype, meant to function on sufficient programs to provide an understanding of its behavior and to provide a platform for adding future enhancements. However, this dissertation includes most of the interesting research questions surrounding Magpie.

In the future, I would like to add the following to Magpie:

- *Support for C++.* Internally, Magpie contains most of the infrastructure required for C++. The parser requires some work to handle `namespace` declarations, but the internal libraries already support namespaces. Exceptions require only simple additions to the parser and stack conversion pass. The front-end of Magpie already includes an overloading analysis, so no additional work is required for overloading. From an implementation point of view, templates require the most work, as parsing and expanding templates is a considerable amount of work. From a research perspective, the `new` expression has properties that Magpie would have to handle. A `new` expression causes an allocation, but also may invoke arbitrary code. Magpie would have to split these steps up, so that the shadow stack was updated properly at each point.
- *Better Instrumentation*: Currently, tuning or measuring the performance of

Magpie-converted programs is extremely difficult. As Magpie not only modifies the cost of allocation and deallocation, but also restructures memory in ways that can affect caching, virtual memory systems, and the hardware TLB, it can be difficult to separate out exactly what is causing a program to slow down or speed up. It would be extremely helpful to have better measurement tools for tuning the garbage collector and/or the traversal procedures.

- *Miscellaneous Internal Improvements*: Currently, the parser for Magpie is a major performance bottleneck. This could be fixed in one of two ways: by improving the parser's performance or by decreasing the amount of work the parser performs. The latter could be done by parsing the original C source rather than preprocessed source; this allows the use of precompiled headers and memoization. Magpie's parser also uses a LALR(1) parser generator implemented in Scheme. Additional performance improvements could be found either by increasing the efficiency of the generated parser, by using C and the highly efficient `bison` parser, or by using a different parsing technology so that the parse tree more accurately reflects the semantics of the source. Finally, several of the analyses — the structure analysis in particular — use naive subroutines and data structures, and their performance could be dramatically improved with a little work.
- *Handling Incremental Updates*: Although I still think that using off-line persistent storage to hold information about the program is a better idea than annotations, annotations do make incremental updates considerably easier. At the moment, contextual information is stored for structures and allocation sites. Handling incremental updates would require the implementation of a semantic patching pass, which would be complicated. Another approach might involve the analyses writing the results back to the original files as annotations, allowing easier patching and revision control. However, as noted previously, macros complicate the implementation of such a system.
- *Concurrency*: The complications surrounding concurrency are described in the previous section. However, there is already strong interest in using Magpie

as a final phase of a compiler for a parallel language, so this may happen quite soon [21].

REFERENCES

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 11–20. ACM Press, 1988.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. Technical report, University of Wisconsin-Madison, December 1993.

[3] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000. USENIX.

[4] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. In *Proceedings of the USENIX 2000 Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298. ACM Press, 2003.

[6] S. J. Beaty. A technique for tracing memory leaks in C++. *SIGPLAN OOPS Mess.*, 5(3):17–26, June 1994.

[7] W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in java: The J-SEAL2 approach. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 139–155, 2001.

[8] H.-J. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 197–206. ACM Press, 1993.

[9] H.-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the Second International Symposium on Memory Management*, pages 59–64. ACM Press, 2000.

[10] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–100. ACM Press, 2002.

[11] D. Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. Foreword By-Grady Booch and Foreword By-Charlie Kindel.

[12] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the First International Symposium on Memory Management*, pages 37–48. ACM Press, 1998.

[13] G. Czajkowski and L. Daynès. Multitasking without compromise: a virtual machine evolution. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 125–138, 2001.

[14] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, 1998.

[15] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 69–80. ACM Press, 2003.

[16] C. Ding and Y. Zhong. Compiler-directed run-time monitoring of program data access. In *Proceedings of the Workshop on Memory System Performance*, pages 1–12. ACM Press, 2002.

[17] F. L. Drake, Jr., editor. *Python Reference Manual*. Python Software Foundation, 2.4.3 edition, March 2006.

[18] R. K. Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, Bloomington, IN, 2005.

[19] R. K. Dybvig, D. Eby, and C. Bruggeman. Don't stop the BiBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana Computer Science Department, March 1994.

[20] M. Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, PLT Scheme Inc., 2005. `http://www.plt-scheme.org/techreports/`.

[21] C. Grothoff. Personal conversation, 2006.

[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX 2002 Technical Conference*, pages 125–138, San Francisco, CA, Winter 1992.

[23] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the Third International Symposium on Memory Management*, pages 150–156. ACM Press, 2002.

[24] A. L. Hosking, J. E. B. Moss, and D. Stefanovic. A comparative performance evaluation of write barrier implementation. In *Proceedings of the 7th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109. ACM Press, 1992.

[25] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363, New York, NY, USA, 1986. ACM Press.

[26] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report 91-47, University of Massachusetts, Object Oriented Systems Laboratory, Department of Comp. and Info. Science, Amherst, MA, 01003, 1991.

[27] G. Insolvibile. Garbage collection in C programs. *Linux Journal*, 2003(113):7, 2003.

[28] S. P. Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999.

[29] D. Lea. A memory allocator, April 2000. `http://g.oswego.edu/dl/html/malloc.html`.

[30] C.-W. Lermen and D. Maurer. A protocol for distributed reference counting. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 343–350, New York, NY, USA, 1986. ACM Press.

[31] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, Reading, MA, 2nd edition, 1999.

[32] E. Meijer and J. Gough. Technical overview of the common language runtime. `http://citeseer.nj.nec.com/meijer00technical.html`.

[33] Microsoft Corporation. *C# Language Specification*, page 320. Microsoft Corporation, Redmond, WA, 2003.

[34] T. Mott. *Learning Carbon*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[35] S. S. Muchnick. *Advanced Compiler Design & Implementation*, pages 443–446. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[36] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139. ACM Press, 2002.

[37] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89 No. 2, 2003.

[38] S. Owicki. Making the world safe for garbage collection. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–86. ACM Press, 1981.

[39] Y. G. Park and B. Goldberg. Reference escape analysis: optimizing reference counting based on the lifetime of references. In *PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189, New York, NY, USA, 1991. ACM Press.

[40] S. M. Pike, B. W. Weide, and J. E. Hollingsworth. Checkmate: cornering C++ dynamic memory errors with checked pointers. In *Proceedings of the thirty-first SIGCSE technical Symposium on Computer science education*, pages 352–356. ACM Press, 2000.

[41] D. W. Price, A. Rudys, and D. S. Wallach. Garbage collector memory accounting in language-based systems. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2003.

[42] J. Regehr. Personal conversation, 2004.

[43] G. Rodriguez-Rivera, M. Spertus, and C. Fiterman. A non-fragmenting non-moving, garbage collector. In *Proceedings of the First International Symposium on Memory Management*, pages 79–85. ACM Press, 1998.

[44] M. P. Rogers. How sweet it is! A course in Cocoa. *J. Comput. Small Coll.*, 18(4):295–307, 2003.

[45] D. J. Roth and D. S. Wise. One-bit counts between unique and sticky. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 49–56, New York, NY, USA, 1998. ACM Press.

[46] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient java. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 104–113. ACM Press, 2001.

[47] Soper, P., specification lead. JSR 121: Application isolation API specification, 2003. `http://www.jcp.org/`.

[48] Sun Microsystems. *Java Native Interface Specification*, chapter 2. 2003. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/design.html#wp17074`.

[49] D. Tarditi. Compact garbage collection tables. In *Proceedings of the Second International Symposium on Memory Management*, pages 50–58. ACM Press, 2000.

[50] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til: a type-directed optimizing compiler for ml. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 181–192, New York, NY, USA, 1996. ACM Press.

[51] A. Wick and M. Flatt. Memory accounting without partitions. In *International Symposium on Memory Management*, 2004.

[52] A. Wick, M. Flatt, and W. Hsieh. Reachability-based memory accounting. In *Proceedings of the 2002 Scheme Workshop*, Pittsburgh, Pennsylvania, October 2002.

[53] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42. Springer-Verlag, 1992.

[54] T. Yuasa. Realtime garbage collection on general-purpose machines. *Journal Of Systems And Software*, 11:181–198, 1990.