Local Variable Allocation For Accurate Garbage Collection of C++

Ravichandran Ganesan
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Programming Languages and Compilers Commons, and the Systems Architecture Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/83

This Article is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Local Variable Allocation For Accurate Garbage Collection of C++

TR94-12
Ravichandran Ganesan

November 8, 1994

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
Local Variable Allocation For Accurate Garbage Collection of C++

Ravichandran Ganesan

Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011

ABSTRACT

Accurate garbage collection of C++ requires that every memory location and every register be known to contain either a pointer or a non-pointer. In order to minimize the run-time overhead of tagging memory locations and registers, techniques for partitioning memory and registers into separate classes dedicated independently to the representation of pointers and non-pointers respectively have been developed. This paper describes the implementation and performance of a specially designed activation frame targeted to the SPARC architecture, as implemented in a customized version of the GNU g++ compiler.

1. Introduction

Accurate garbage collection of C++ improves programmer productivity, simplifies the development of reusable software components, and offers the potential of supporting more reliable and predictable dynamic memory management performance than is possible using traditional C++ dynamic memory management techniques. But previous implementations of accurate C++ garbage collectors require special hardware and incur a run-time overhead of up to 30%, even with the aid of the special hardware support [11].

Schmidt and Nilsen have identified the management of stack activation frames to be the primary source of run-time overhead in an existing hardware-assisted accurate garbage collection system for C++ [9, 11]. The challenge lies in properly identifying precisely which words within every activation frame contain pointers. Schmidt studied several alternative strategies:

1. Upon entry to a function, the stack expands to represent the new activation frame, and special code is executed to tag each word within the newly allocated activation frame as either a pointer or a non-pointer word. Preparatory to building an argument list for a function to be called from the constructed activation frame, the parameter passing area (found within the current activation frame) is tagged to represent the types of the parameters to be passed.

2. Same as 1, except separate parameter passing areas are maintained for each function called from the current activation frame. Thus, there is no need to retag the memory set aside to represent the parameter passing area.

3. Heap allocate all activation frames, tagging the parameter passing area prior to every function call.

1 Portions of this paper were excerpted from Code Generation to Support Efficient Accurate Garbage Collection of C++ on Stock Hardware, a paper currently being prepared for publication by Kelvin Nilsen, Ravichandran Ganesan, Satish Guggilla, Satish Kumar, and Kannan Narasimhan.
4. Heap allocate all activation frames, dedicating separate parameter passing areas to each function called from the allocated activation frame.

All of these techniques demonstrated very poor performance. For the benchmark applications that were studied, the overhead in comparison with traditional activation frame implementations was a minimum of 150% of the traditional execution costs! The second technique exhibited slightly better performance than the first, and the fourth was slightly better than the third. The first two techniques exhibited measurably better performance than the last two. See reference 9 for a more detailed analysis.

In response to these observations, Nilsen and Schmidt designed a more efficient activation frame protocol for C++ programs [8, 9]. In this revised design, all activation frames are heap allocated as in the fourth alternative described above. However, rather than discarding activation frames in order for the garbage collector to reclaim them, the new design recycles activation frames for each function independently. Thus, if the leaf function \( f \) is called a hundred times, the overhead of allocating and tagging the contents of the activation frame is seen only the first time the function is called. The other 99 function calls reuse the previously allocated activation frame.

The success of this scheme is highly dependent on the activation frame hit rate, defined as the ratio of the number of function activations whose frames are allocated from the free lists to the total number of function activations. Measurements of this ratio on two real programs found activation frame hit rates of 0.9995 and 0.9998 [9, 10]. This is the activation frame allocation scheme that was measured to incur overheads of up to 30% on program execution times. The reason that the overhead is so high, even though an extremely low percentage of function calls incur the overhead of allocating and tagging the activation frame, is because the effort required to maintain free lists of previously used activation frames adds nine instructions to the code representing the prologue and epilogue of every function in the system. The purpose of this research project is to evaluate a proposed technique designed to eliminate this 9-instruction overhead for a high percentage of the function calls in a typical application.

**Related Work**

Most of the previous research on code generation techniques to support garbage collection has focused on languages designed to incorporate garbage collection. In these languages, clean semantic models and cooperative implementation techniques make the implementation of efficient garbage collection much easier than in languages such as C and C++ which were not originally designed to make use of automatic garbage collection. As this paper describes garbage collection of C++, our survey of related work concentrates on code generation techniques for uncooperative languages.

Existing literature describes only two alternative techniques for garbage collection of the full C++ language. Conservative garbage collection, designed by Boehm and his colleagues, works, for the most part, with traditional C and C++ compilers by assuming that every word of memory may contain a pointer. It treats as a pointer any register or memory cell containing a value that is a valid heap memory address [1]. Conservative garbage collection has been shown to provide good time and space performance on a wide variety of real-world applications [13]. The alternative available garbage collection technique for C++ is accurate garbage collection, as described in references 8 and 7. Accurate garbage collection requires cooperation from the compiler’s code generator in order to distinguish pointers from non-pointers within memory and registers.

Though the design of conservative garbage collection was intended to avoid the need for modifications to compiler code generators, a number of traditional code optimization techniques have been found to be incompatible with conservative garbage collection. For example, certain induction variable optimizations may convert the only pointer to a particular heap object into an integer offset relative to some other pointer (which refers to a different heap object). If garbage collection is triggered while the loop is executing, the garbage collector will not recognize the referenced object as live. Another example of problematic code generation results from addition of a pointer variable and a large integer constant. This may translate into a two-instruction sequence, the first instruction of which adds a constant to the high-
order bits of the pointer so that the resulting value points beyond the borders of the referent object. Bohem and Chase have studied a number of different compilers and have characterized the sorts of problems that their code generation strategies present to the conservative garbage collection technique. They have addressed these problems in reference 2.

Since conservative garbage collectors are never sure of whether a particular value is a pointer or a non-pointer, they are not able to relocate objects in order to reduce memory fragmentation. This is because a relocating garbage collector needs to update all pointers to reflect the new locations of the objects they refer to. If a particular word that resembles a pointer is being used by the program as a pointer, then it would be acceptable for the garbage collector to move the referent object and modify the pointer. But suppose the memory really represents an integer which just happens to hold a value that resembles a heap address. If the conservative garbage collector modifies this word as a consequence of relocating the referenced object, then the integer’s value becomes corrupted. Thus, relocating garbage collectors need to be accurate garbage collectors, in the sense that they must have full knowledge of exactly which memory cells and registers represent pointers.

In reference 3, Diwan, Moss, and Hudson of the Object Systems Laboratory at the University of Massachusetts at Amherst describe the modifications they made to the GNU code generator in order to support relocating garbage collection of Modula-3. Modula-3, unlike C++, was designed to cooperate with automatic garbage collection. Thus, Diwan’s techniques do not generalize directly to garbage collection of C++. Nevertheless, it is instructive to consider the methods Diwan has developed. Diwan’s garbage collector requires a base pointer to the beginning of every heap object. Pointers that refer to internal fields of heap objects must be converted into base pointers in order for the garbage collector to process them. Like Boehm and Chase, Diwan points out that standard compiler optimizations may result in pointers that do not point directly to the bases of objects, and in pointers that refer to addresses beyond the boundaries of the objects that they belong to. Diwan calls these derived pointers, because they are derived from the base pointers by some sequence of arithmetic operations. Unlike Boehm and Chase, Diwan needs to be able to accurately identify every pointer in the system, and must be able to map every pointer to the base address from which it was derived.

Diwan’s technique is to select gc-points within each function, which identify control points at which garbage collection might begin. At each gc-point, the compiler builds a number of tables that characterize the contents of all registers and activation frame slots at that execution point. Two of these tables identify which registers hold pointers, and which activation frame slots hold pointers, respectively. A third table provides formulas for calculating base pointers from the derived pointers held in certain registers and stack locations. In order to enable these calculations, it is occasionally necessary to extend the lifetimes of certain variables beyond the point at which a traditional register allocator would consider them dead. The increased register pressure may result in worse generated code under some circumstances, but Diwan reports that they found no differences between the optimized code generated by the traditional compiler and the code generated by the revised system for the benchmark applications that were evaluated in reference 3.

Previous work by Schmidt and Nilsen differs from Diwan’s work in that the target language is lower level than Modula-3 [8, 9]. Since the target language is C++, there is no hope of tracking base pointers for all heap-allocated objects. In C++, derived pointers may be assigned to programmer-defined variables and passed as arguments to other functions. Therefore, the garbage collector (rather than the compiler) takes responsibility for computing base pointers from derived pointers. Special circuitry provides the functionality in the hardware-assisted real-time garbage collection system [6]. In the stock-hardware garbage collection system, this functionality is implemented in software [5]. Thus, Schmidt’s compiler did not need to preserve base pointers, nor did it need to find ways to compute base addresses of objects from derived pointers. Another difference between Schmidt’s compiler and the other two studies cited in this section is

---

2 The GNU code generator serves as a portable back-end for a number of different languages, including C, C++, Smalltalk, and Modula-3.
that Schmidt’s compiler did not support optimizations.

Terminology

In the discussion that follows, we use the term descriptor to denote a pointer. By pointing to objects allocated elsewhere, each descriptor is capable of “describing” all conceivable kinds of information. We use the adjective terminal to characterize memory locations known not to contain pointers. If all live memory is represented as a directed graph in which nodes represent dynamically allocated objects and directed edges represent pointers from one object to another, the terminal nodes are those from which no directed edges emanate. The source nodes in this directed graph are pointers residing outside the garbage-collected heap. These source pointers, which are under direct control of the CPU, are called root descriptors.

2. A Survey of Local Variable Needs

Since our goal is to improve the efficiency of function entry and exit, we first undertook to better understand the activation frame needs of a typical function. In particular, we sought to understand how many pointers and non-pointers needed to be stored within activation frames, and what sort of structure was typically imposed on the internal organization of activation frames by local structure and array declarations. We studied four C++ applications:

cfrac:
    Cfrac is a program to factor large integers using the continued fraction method.

cham:
    Chameleon is a N-level channel router for multi-level printed circuit design.

espr:
    Espresso is a logic optimization program.

gawk:
    Gnu version of awk – pattern scanning and processing language.

These programs were proposed by Zorn as standard benchmarks for evaluation of applications that make extensive use of dynamic memory [12]. Gao converted these programs, originally written in C, to C++ so that they could be analyzed by C++ compilers [4].

We instrumented the GNU C++ compiler to report the number of scalar pointers and the number of scalar non-pointers found in the activation frames of each function in the four experimental workloads. These distributions are illustrated in figures 2.1 through 2.4, shown below. Note that the graphs reveal a different style of programming in each of the applications. Most cfrac (Fig. 2.1) functions, for example, require considerably more descriptors than terminals. Nevertheless, the large majority of cfrac functions use no more than 25 descriptors. The cham functions (Fig. 2.2) are more evenly distributed. It appears that approximately equal fractions use more terminals than descriptors and vice-versa. Note also that the large majority of cham functions use no more than five descriptors and ten terminals.

Table 2.1 summarizes the deviation between the number of terminals and the number of descriptors in the activation frames for the functions that comprise each of the four experimental workloads. Note that a large majority of the function activation frames utilize approximately equal numbers of descriptor and terminal data words. Missing from the graphs is a report of the numbers and internal organizations of locally declared structures and arrays. We call these aggregates. Table 2.2 summarizes the numbers of functions within each experimental workload that make use of aggregate variables. Note that only a small fraction of the total number of functions studied use local arrays and structures, and that none of the studied functions makes use of any local unions. Table 2.3 reports these observations in terms of percentages.

Based on these observations, we set out to design an activation frame that would perform very efficiently for typical functions (approximately equal numbers of descriptors and terminals, and no aggregates), and
would perform reasonably well for atypical functions.
3. Traditional Stack Variable Allocation

We considered the traditional C++ activation frame design to represent the target efficiency. Using traditional implementation techniques, the cost of allocating an activation frame is simply the cost of
Table 2.1: Deviation between descriptor and terminal words

<table>
<thead>
<tr>
<th>Program</th>
<th># of functions</th>
<th># of functions with descriptors − terminals</th>
<th>&gt; 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>60</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>cham</td>
<td>109</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>espr</td>
<td>358</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>gawk</td>
<td>106</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Aggregates

<table>
<thead>
<tr>
<th>Program</th>
<th># of functions with local arrays</th>
<th># of functions with local structs</th>
<th># of functions with local unions</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cham</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>espr</td>
<td>14</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>gawk</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.3: Percentage Aggregates

<table>
<thead>
<tr>
<th>Program</th>
<th># of functions</th>
<th>Percentage with local aggregates</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>cham</td>
<td>109</td>
<td>5.5</td>
</tr>
<tr>
<td>espr</td>
<td>358</td>
<td>8</td>
</tr>
<tr>
<td>gawk</td>
<td>106</td>
<td>13.2</td>
</tr>
</tbody>
</table>

decrementing the %sp (stack pointer) register by the size of the new activation frame. And the cost of deallocating the activation frame is the the cost of incrementing the %sp register by the same amount.

3.1. Register Windows

Since our implementation targets the SPARC architecture, it is necessary to understand the use of register windows. SPARC machines contain from 40 to 520 general-purpose 32-bit r registers. These registers are partitioned into 8 global registers, plus an implementation-dependent number of 16-register sets. Each register set is further partitioned into 8 in registers and 8 local registers. The in registers of each window overlap with the out registers of the caller’s window. A register window comprises 8 in and 8 local registers of a particular register set, together with 8 in registers of an adjacent register set, which are addressable from the current window as out registers (see Figure 3.1.1). The SPARC instruction encoding reserves five bits to address registers. Register numbers are mapped according to the following table:

<table>
<thead>
<tr>
<th>Windowed Register Name</th>
<th>Absolute Register Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>global[0] − global[7]</td>
<td>r0 − r7</td>
</tr>
<tr>
<td>out[0] − out[7]</td>
<td>r8 − r15</td>
</tr>
<tr>
<td>local[0] − local[7]</td>
<td>r16 − r23</td>
</tr>
<tr>
<td>in[0] − in[7]</td>
<td>r24 − r31</td>
</tr>
</tbody>
</table>

Thus, at a given time, an instruction may access any of the the 8 global registers, any of the 16 registers in the register set dedicated to the current function, and any of the 8 in registers of the next register set. The in and out registers are used primarily for passing parameters to subroutines, and for receiving the results from called functions, and for keeping track of the run-time stack.
When a procedure is called, the caller stores the parameters in its *out* registers and the current window pointer is decremented to point to the adjacent window. The callee accesses its parameters as *in* registers. Only the first six *out* registers (%o0 – %o5) are available for passing parameters. By convention, register %o6 serves as the system stack pointer (%sp is an alias for %o6), and %o7 holds the return address for a called function. Note that %i6 of the callee is the same as %o6 of the caller. %i6, which represents the height of the stack at the time the current function was called, serves as the frame pointer (%fp is an alias for %i6). If a particular function has more than six scalar parameters, the additional parameters are passed in stack locations. The callee stores its return value in its %i0, which is accessed by the caller as %o0.

![Figure 3.1.1: SPARC Register Windows](image)

**3.2. The SPARC Activation Frame**

The traditional SPARC activation frame consists of the following:

- 16 words always starting at %sp for saving the procedure’s *in* and *local* registers, should a register window overflow occur.
- One word is always reserved for passing a “hidden” parameter. Whenever a function is declared to return a structure, the caller of the function has the responsibility of allocating the memory into which the structure will be stored. The address of the allocated memory is passed as this hidden parameter. Note that this word of the activation frame is unused whenever the current function calls a subroutine that returns a scalar argument.
• Though the first six parameters are passed in registers, each of the six parameter-passing registers is assigned a corresponding memory location by the caller. The callee may decide to copy the parameter registers into these memory locations if it desires to spill the register, or if source code endeavors to make use of the parameter’s address.

The following additional fields are allocated only as needed within the stack frame.
• Space for outgoing parameters beyond the sixth scalar parameter.
• All automatic arrays, automatic aggregates, automatic scalars which must be addressable, and automatic scalars for which there is no room in registers.
• Compiler-generated temporary values.
• Floating-point registers saved across calls.

An automatic variable in the stack is addressed as a negative offset from the frame pointer (%fp). A typical stack frame is shown in Figure 4.1.

4. Design of the Activation-Frame

It is important for the garbage collector to identify which words in the stack frame are descriptors and which are terminals. So we need a mechanism to give this information to the garbage collector. Schmidt’s implementation [9] uses a technique called **Pointer Location Descriptions** (PLD for short) to identify the descriptors and terminals in a run-time stack. In Schmidt’s system, each activation frame is accompanied by a PLD, which specifies which words in the activation frame are descriptors and which are not. The PLD contains a bit map within which each bit represents a different word of the activation frame. The bit is set if and only if the corresponding word contains a descriptor.

In the revised activation frame design, it is still necessary to differentiate between descriptors and terminals. However, the new design reduces the run-time overhead of tagging activation frame words, while also reducing the memory needs related to representation of tags.

The traditional activation frame for a function call in a SPARC machine is shown in Figure 4.1. In the revised design, we divide the stack into chunks of 4 words, and impose the restriction that **all the descriptor data in a function must occupy the odd numbered chunks and all the terminals must occupy the even numbered chunks**. With this restriction, it is straightforward for the garbage collector to distinguish between the descriptors and terminals. Based on the revised design, the new activation frame for a function call is illustrated in Figure 4.2.

The example below illustrates clearly the difference between the existing activation frame and the new-activation frame. Consider the following function declaration:

```c
foo() {
    int t1, t2;
    int *d1;
    int t3, t4, t5;
    int *d2;

    function body
}
```

In the existing activation frame design, space for the variables is allocated in the same order as the variables’ declarations are processed by the compiler. This is shown in Figure 4.3 (A). In the revised design, space for t1 and t2 are allocated as in the original activation frame, but d1 is placed in a different chunk, because d1 is a descriptor. This is shown in Figure 4.3 (B). t3 and t4 occupy the third and fourth words of the first chunk of terminals, and d2 is allocated to the second word of the first descriptor chunk. The compiler places t5 in the first word of the second chunk of terminals. The difference between the original
activation frame and the new design is clearly portrayed in Fig. 4.3. In this revised system, allocation and deallocation of function activation frames consists simply of incrementing or decrementing %sp by the
size of the activation frame, which is always an even multiple of the chunk size.
The main additional challenge is the handling of aggregate data types, such as arrays and structures. Note
that many aggregates do not naturally align within the constraints imposed by the alternating chunks that
comprise the run-time stack. Consider, for example, the declaration:

```c
int a[100];
```
This variable requires one hundred consecutive terminal words. These are not available on the run-time
stack.

We have designed a run-time type identification system based on signatures. A signature is a bit stream
associated with an area of memory. It contains one bit corresponding to each word of the corresponding
area. The bit is set if and only if the corresponding location contains a descriptor. The following example
shows several declarations and the corresponding signatures.

```c
int i;  signature: 0
```
int *k; \hspace{1em} \text{signature: 1}
float f; \hspace{1em} \text{signature: 0}
double d; \hspace{1em} \text{signature: 00}
double *dp; \hspace{1em} \text{signature: 1}

The signature of int i is 0, since i is terminal data occupying one word. The signature of int *k is 1, since k is descriptor data occupying one word. The signature of an aggregate data object is the catenation of the signatures of its constituent data types.

For example, the variable ab, declared below:

\begin{verbatim}
struct a {
    int *i;
    int *j;
    int k;
} ab;
\end{verbatim}

has the signature 011, and

\begin{verbatim}
int a[10];
\end{verbatim}

has the signature 0000000000. Therefore, the variable baz in the following declaration

\begin{verbatim}
struct b {
    struct a s1;
    int x[10];
} baz;
\end{verbatim}

has the signature 0000000000011, obtained by catenating the signatures of the individual structure elements.

In the actual implementation, the signature is represented by the following structure:

\begin{verbatim}
struct sig {
    int nwords; // Number of words represented by the ptrmap
    int scancnt; // Number of words that the garbage collector needs
        // to scan – i.e. the offset of the last non-zero
        // bit within ptrmap
    unsigned int ptrmap[0]; // variable length array containing pointer bitmaps.
};
\end{verbatim}

Signatures of structures and arrays are normally constant throughout program execution. However, signatures of unions that contain both descriptor and terminal variants are updated dynamically. For constant signatures, scancnt represents the word offset of the last descriptor within the object whose type is described by the signature. For dynamic signatures, scancnt represents the word offset of the last word within the corresponding object that might contain a descriptor. Thus, the signature for the variable baz declared above is represented by the three-tuple \{13, 2, 0x3\}. This means baz occupies 13 words, and all of the descriptors within baz are found within the first two words. The value 0x3 signifies that both of the two first words contain descriptors.

Each function that has local aggregate variables is allocated a heap-allocated aggregate area in which to store all of its aggregate locals. This is illustrated in Fig. 4.4. Consider the local variables declared within function foo. t1, t2, and d1 are allocated slots on the activation frame stack. arr and a are allocated positions within the aggregate area.
foo() {
    int t1, t2;
    int *d1;
    int arr[5];
    struct {
        int *i;
        int j;
    } a;

    function body
}

Figure 4.4: Activation Frame for Function with Aggregate Data

When the compiler encounters a local aggregate, it begins construction of the corresponding aggregate area, and creates a local variable (usually represented by a register) which points to the base of the aggregate area. Call this variable the aggregate area pointer. Local scalar variables are accessed using integer offsets from the %fp register, and local aggregate variables are accessed using integer offsets from the aggregate area pointer. The compiler constructs a signature for the aggregate area, so that the garbage collector can identify which words within the aggregate area hold pointers. With this new set-up the descriptors in the stack can be easily identified as they occupy the odd chunks, and the descriptors in the aggregate area can also be easily identified using the signature associated with the aggregate area.

In order to reduce the costs of function entry and exit, aggregate areas are recycled for use by subsequent invocations of the same function. On exit from a function that uses an aggregate area, the function’s
epilogue links the aggregate area onto a list representing all of the previously allocated aggregate areas of this particular size and signature. Separate linked lists of available aggregate areas are kept for each different size and type of aggregate area. The list heads are maintained in an array called the activationFrameCache, which is illustrated in Fig. 4.5.

Note that each entry in this array is a pointer to a list of activation frames. Every function that uses an aggregate area is assigned a unique integer index within the activationFrameCache array. Thus, each slot within this array represents the cache of recycled aggregate areas of the size and type that are appropriate for each individual function that makes use of aggregate areas. When one of these functions begins to execute, it consults the appropriate slot of the activationFrameCache to see if a previously allocated aggregate area is available. If so, it uses it, unlinking the recycled aggregate area from its free list. When the function exits, it replaces the aggregate area onto the corresponding free list.

Only recursive functions will have caches of more than a single available aggregate area. Consider the program fragmented displayed below:

---

\[ As a future optimization, we intend to map multiple functions that use identical aggregate areas to the same free list. Currently, every function has a unique list. \]
main()
{
    Before_f_1:
    f(1);
    Before_g:
    g();
    Before_h:
    h();
    Before_f_2:
    f(2);
    After_f_2:
}

f(int i) {
    struct foo baz;

    if (i == 1) {
        // do something
    }
    else {
        Before_Recursion:
        f(i - 1);
        After_Recursion:
    }
}

f is a recursive function with local aggregates. Assume g, which is not shown, is a function with local aggregates. And assume h, also not shown, is a function with no local aggregates. Figure 4.6 displays the contents of the activationFrameCache at several different execution points, as identified by the labels in the above code. Note that the first slot in activationFrameCache represents the list of available aggregate areas for the function f, and the second slot corresponds to the function g.

Initially, both of the lists are empty, as shown in Figure 4.6 (A). On exit from the function f, the allocated aggregate area is added to the list corresponding to the function f, as illustrated. After returning from the invocation f(1), there is one available aggregate area in the first slot of activationFrameCache. g performs similarly. Since h has no local aggregates, the activationFrameCache is unaffected by its execution. When f is called again after Before_f_2: the compiler-generated prologue finds that there is an aggregate area in the entry corresponding to f, and hence uses that aggregate area. This is reflected in in Figure 4.6 (e). When f is entered recursively, the corresponding entry within the activationFrameCache is now empty. Thus, f’s prologue allocates a new aggregate area, which is placed onto the appropriate free list at a later time by f’s epilogue.

Though we have not yet measured the performance of the revised system on real workloads, we expect to see activation frame reuse rates that are similar to those reported by Schmidt [9]. The main benefit, however, in comparison with Schmidt’s technique is that over 90% of function invocations do not even have to bother with the management of aggregate areas, since they have no local aggregate variables.

5. Implementation

5.1. The Basic Idea
To implement the proposed design, we modified the existing GNU C++ compiler. Among other things, we modified the compiler’s code generation functions that allocate stack memory for local variables. We also
modified the generation of code for function prologues and epilogues, in order to manage the cache of aggregate areas.

5.2. assign_stack_temp

The function assign_stack_temp takes responsibility for allocating memory for a local variable within the activation frame. In the original system design, assign_stack_temp did not need to know the type of the variable to be allocated. Thus, this information was not generally available within the body of assign_stack_temp. In the revised system, assign_stack_temp must know the type of the variable in order to decide where to place it within the activation frame: scalar terminals in even-numbered chunks, scalar descriptrs in odd-numbered chunks, and aggregates in the aggregate area. In order to satisfy this need, we added a fourth argument to assign_stack_temp to represent the signature of the variable to be allocated. This required that we search for all invocations of assign_stack_temp and supply the extra argument. In several cases, obtaining the signature of the variable to be allocated required a significant amount of work.
assign_stack_temp maintains a record of all the slots that have been assigned to the local variables of the function that is currently being translated. The record is maintained as a list of temp_slot structures:

```c
struct temp_slot {
    struct temp_slot *next; // points to next available slot
    rtx slot; // an rtx that represents address of this slot
    int size; // size, in bytes, of the slot
    char in_use; // non-zero if this slot is currently in use
    int level; // static nesting level of this slot
    int keep; // non-zero if this should be preserved
                // beyond call to free_temp_slots
    struct sig *sig; // signature of the slot
};
```

For each allocated slot, assign_stack_temp remembers the size of the slot (size), the signature of the slot (sig), whether the slot is currently in use (in_use), and the intermediate code that represents the address of the slot (slot). The GNU compiler’s intermediate code is called register transfer language (rtl). An rtl expression is abbreviated rtx. Note that the only distinction between temporary slots allocated on the stack and temporary slots allocated within the aggregate area is the rtx that describes the address of the slot. assign_stack_temp allocates slots for programmer-declared function variables, block variables, and compiler-generated temporaries. Temporary variables which are used only within particular compound statements are allocated with keep equal to zero. The current static block nesting level is maintained in a global variable. At the time that a temporary slot is allocated, the level field is initialized by copying from this global variable. Within the temp_slot data structure, keep is zero only for slots assigned to local variables whose lifetime is shorter than the entire function body. At the next invocation of free_temp_slots(), all slots that are currently in use (in_use) and have keep equal to zero are marked as no longer in use.

assign_stack_temp’s arguments are:

- **mode**: For scalar variables, SImode signifies terminal data and PSImode signifies descriptor data. For aggregate data, the mode parameter is ignored.

- **size**: The size of the temporary to be allocated, measured in bytes.

- **keep**: The value to be assigned to the temp_slot field by the same name.

- **sig**: For aggregate variables, sig represents the signature of the aggregate object. For scalar variables, sig is NULL.

The pseudo-code for assign_stack_temp is provided below.
rtx assign_stack_temp(mode, size, keep, sig) {

    Search the free list for an available slot exactly the desired size and
    signature tag that is not in-use
    if (slot is found)
        mark the slot as in-use and return its rtx
    else {
        search the free list for the smallest available slot that is at least
        as large as the desired size and matches the signature tag and
        is not in-use
        if (slot is found)
            mark the slot as in-use and return its rtx
        else {
            allocate a new temporary slot (P).
            Add it to the list of temp_slots.
            P->slot = assign_stack_local(mode, size, 0, sig).
            Initialize other fields.
            return P->slot
        }
    }
}

Note that assign_stack_temp calls assign_stack_local if it is unable to find an available slot amongst
the slots that have already been allocated (and deallocated) for the current function. assign_stack_local
finds space for the requested temporary, either in the stack frame or the aggregate area and returns an rtx
that describes the address of the allocated space. Note that assign_stack_local may expand the stack
frame or the aggregate area as a side effect of allocating the local space.

5.3. assign_stack_local

assign_stack_local expands the pool of slots within the activation frame in order to make room for a new
temporary variable. The arguments to assign_stack_local are:

mode:
    For scalar variables, PSImode signifies descriptor data. For aggregate data, the mode parameter is
    ignored.

size:
    The size of the requested temporary, measured in bytes.

align:
    align specifies the alignment restrictions for the newly allocated slots. A value of 0 means align
    according to mode. A value of −1 means align according to BIGGEST_ALIGNMENT, a machine-
    specific constant that characterizes the most constrained alignment appropriate for the target hard-
    ware (if align equals −1, the size is rounded up to the nearest multiple of
    BIGGEST_ALIGNMENT). Other positive values for align represent the desired alignment, specified
    in bits. For example, if align equals 64, align on a two-word (64-bit) boundary.

sig
    For aggregate variables, sig represents the signature of the aggregate object. For scalar variables, sig
    is NULL.

The implementation of assign_stack_local makes use of three global variables, which represent the off-
sets within different allocation regions. desc_frame_offset and term_frame_offset are the offsets within
the stack area of the next available descriptor and terminal data slots respectively. aggregate_frame_offset
is the current size of the aggregate area. Note that if sig is non NULL, the system
allocates space in the aggregate area, if sig is NULL if the mode is PSlmode, the system allocates a descriptor slot otherwise alloc_stack_local allocates a slot for terminal data. The variable pres_fun_aggregate_sig, which is updated each time the aggregate area is expanded, represents the signature of the function’s aggregate area.

Pseudo-code for assign_stack_local is provided below:

```c
rtx assign_stack_local(mode, size, align, sig) {
    if (sig is not NULL) {
        // allocate within the aggregate data area
        if (this is the first aggregate allocation for this function) {
            allocate a descriptor from within the descriptor frame to point
            to the aggregate data area
            adjust desc_frame_offset appropriately
        }
        align the allocation and size
        update the structure referenced by pres_fun_aggregate_sig
        update aggregate_frame_offset appropriately
    } else if (mode == PSlmode) {
        // allocate a descriptor slot
        update desc_frame_offset appropriately
    } else {
        // allocate a terminal slot
        update term_frame_offset appropriately
    }
}
```

5.4. Code Generation of Function Prologues

We treat functions having aggregate locals differently than the ones that do not have local aggregates. When generating code, we examine pres_fun_aggregate_sig to determine whether a function makes use of local aggregates. pres_fun_aggregate_sig is non-NULL only if the function being translated makes use of local aggregates.
compute the frame_size;
generate code to initialize the stack pointer by subtracting the frame size
from the old value of the stack pointer;
// Note that the new_function will have a different register window
// and hence a different stack_pointer. The old stack_pointer
// becomes the new frame_pointer

if (pres_fun_aggregate_sig != NULL) {
generate code to check whether a previously allocate aggregate frame
is available;
the generated code uses the existing frame if available, removing the
frame from its corresponding free list;
otherwise, the generated code calls gc_alloc(size, sig);
}

The following display compares the prologue of a function that uses an aggregate area with the prologue of
a function that does not:

<table>
<thead>
<tr>
<th>Prologue, using aggregate area</th>
<th>Traditional Prologue</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>main:</td>
</tr>
<tr>
<td>!#PROLOGUE# 0!</td>
<td>#PROLOGUE# 0</td>
</tr>
<tr>
<td>save %sp, -144,%sp</td>
<td>save %sp, -144,%sp</td>
</tr>
<tr>
<td>!#PROLOGUE FOR AGG AREA</td>
<td>!#PROLOGUE# 1</td>
</tr>
<tr>
<td>ld [addr], %i4</td>
<td>0</td>
</tr>
<tr>
<td>ld 0, %i3</td>
<td>1</td>
</tr>
<tr>
<td>ld [%i4 + %i3], %i5</td>
<td>2</td>
</tr>
<tr>
<td>andcc %i5, %i5, %i5</td>
<td>3</td>
</tr>
<tr>
<td>bnz @_main_c.c@_act_L1:</td>
<td>4</td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>mov 48 , %o0</td>
<td>A</td>
</tr>
<tr>
<td>sethi %hi(NewSignature.1), %o1</td>
<td>B</td>
</tr>
<tr>
<td>or %o1, %lo(NewSignature.1),%o1</td>
<td>C</td>
</tr>
<tr>
<td>mov 1, %o2</td>
<td>D</td>
</tr>
<tr>
<td>call __GC_alloc, 0</td>
<td>E</td>
</tr>
<tr>
<td>nop</td>
<td>F</td>
</tr>
<tr>
<td>mov %o0, %i5</td>
<td>G</td>
</tr>
<tr>
<td>ba @_main_c.c@_Lab:</td>
<td>H</td>
</tr>
<tr>
<td>nop</td>
<td>I</td>
</tr>
</tbody>
</table>

@_main_c.c@_act_L1:
    ld [%i5], %i6 ! 7
    ld %i6, [%i4 + %i3] ! 8
    !#PROLOGUE FOR AGG AREA
@_main_c.c@_Lab:
    !#PROLOGUE# 1

In this code, we have added an italicized label to each of the additional assembly statements that is required
in the customized version of the function prologue. In statement 1, addr represents the base address of the
activationFrameCache. Statement 2 initializes register i3 to the offset corresponding to the main pro-
gram within the activationFrameCache. The offset happens to be zero. Statements 3, 4 and 5 test to see
whether the corresponding activationFrameCache entry is NULL. If so we need to allocate a new aggregate area. Instruction 6 is the branch delay slot instruction. Statements 7 and 8 unlink the previously allocated aggregate area from its free list. Studies performed previously by Schmidt suggest that over 99% of the function prologue executions take this path through the prologue. Fewer than 1% of the prologue invocations execute statements A, B, C, D, E, F, G, H, and I, which take responsibility for heap allocating and initializing a new activation frame. Note that __GC_alloc takes three arguments which represent the size of the object to be allocated, the address of the signature to be associated with the object, and a flag which specifies whether the allocated object contains any unions that would necessitate a dynamic signature. The typical path through the code on the left executes the eight more instructions (the ones that are numbered) than the code on the right.

5.5. Code Generation of Function Epilogues

In the function epilogue, any function that makes use of local aggregates is required to place its aggregate area onto the appropriate free list. The following pseudo-code describes the code generators responsibilities in generating function epilogues:

```
if (pres_fun_aggregate_sig != NULL)
    generate code to return the aggregate area to the appropriate free list
    generate code that returns to the calling function
```

The following display compares the epilogue of a function that uses an aggregate area with the epilogue of a function that does not:

<table>
<thead>
<tr>
<th>Epilogue, using aggregate area</th>
<th>Traditional Epilogue</th>
</tr>
</thead>
<tbody>
<tr>
<td>!#EPILOGUE FOR AGG AREA !0</td>
<td>ret</td>
</tr>
<tr>
<td>ld [addr], %i4</td>
<td>!1 ret</td>
</tr>
<tr>
<td>ld 0, %i3</td>
<td>!2 restore</td>
</tr>
<tr>
<td>ld [%i4 + %i3], %i6</td>
<td>!3</td>
</tr>
<tr>
<td>st %i6, [%i5]</td>
<td>!4</td>
</tr>
<tr>
<td>st %i5, [%i3 + %i4]</td>
<td>!5</td>
</tr>
<tr>
<td>!#EPILOGUE FOR AGG AREA</td>
<td>ret</td>
</tr>
<tr>
<td>ret</td>
<td>restore</td>
</tr>
</tbody>
</table>

As with the code for the prologue, addr represents the base address of the activationFrameCache. This code, which consists of five extra instructions in each epilogue that makes use of local aggregate areas, places the aggregate area onto the appropriate free list. Note that the number of instructions could be reduced by retaining the address of the activationFrameCache slot in a dedicated register throughout execution of the function.

5.6. Dynamic Analysis of Aggregate Area Frequency

Table 2.3 reports the percentage of functions defined within each of the experimental workloads that makes use of aggregate areas. Table 5.6.1 reports the percentage of function invocations that require the use of an aggregate area for a sample execution of each of the experimental workloads. In general, we observe that the dynamic frequencies are even lower than the static frequencies. In other words, the functions that make use of aggregate areas are less likely to be invoked frequently than the functions that do not make use of aggregate areas.
Table 5.6.1: Dynamic Analysis

<table>
<thead>
<tr>
<th>Program</th>
<th># of calls in an execution</th>
<th># of calls to functions with aggregates</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfrac</td>
<td>2,239,234</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>cham</td>
<td>438,706</td>
<td>13,786</td>
<td>3.10</td>
</tr>
<tr>
<td>espr</td>
<td>279,389</td>
<td>143</td>
<td>0.05</td>
</tr>
<tr>
<td>gawk</td>
<td>439,379</td>
<td>40,489</td>
<td>9.20</td>
</tr>
</tbody>
</table>

Note that *cfrac* has no functions with local aggregates. These measurements suggest that the need to execute the more complicated versions of prologue and epilogue code that are required to manage the aggregate areas is relatively rare.

6. Conclusions and Suggestions for Future Work

All of the above-mentioned changes have been implemented and tested. This represents an important step in achieving the goal of fast and accurate garbage collection of C++, both for traditional systems running on stock hardware and for hard-real-time systems running on custom hardware.

This effort is part of a large project involving a number of researchers working on a variety of different aspects of the total system design. Other efforts are focusing on the identification of descriptors and terminals in the global data area, the machine registers, and the parameter passing area within each activation frame. Additionally, we are modifying the code optimizer so that it will avoid garbage-collection unsafe transformations [2, 3]. And we are investigating several different garbage collection techniques, both for stock hardware and for custom hardware.

We have not yet been able to evaluate the performance of an integrated system because parameter passing has not yet been implemented. Once the system has been integrated, we intend to perform extensive analysis of the generated code’s performance, and expect to find tune the compiler in response to the measured performance.

One space optimization that we intend to make is to merge multiple functions with identical aggregate areas into shared slots within the *activationFrameCache* array. This will reduce memory fragmentation and decrease the size of the *activationFrameCache* array. However, as discussed in section 2, the number of functions that use aggregate areas is only a small percentage of the total functions in the system. So the impact of this proposed optimization is not likely to be very significant.

7. Acknowledgements

I would like to express my gratitude to Dr. Kelvin Nilsen, my advisor, for his patience, guidance, resourcefulness and understanding throughout the course of this project. He has spent countless number of hours discussing my work and suggesting new ideas to explore. Working with him has been fun and rewarding.

I would also like to thank Dr. Suresh Kothari, Dr. Gary Leavens and Dr. Charlis Wright, my Program of Study Committee members, for their valuable time and co-operation.

I am grateful to the College of Liberal Arts and Sciences and the Computer Science Department at Iowa State University for financial support during this period. Finally, I am grateful to all my friends at Iowa State University for making my stay here fun and memorable.

8. References


5. S. K. Guggilla, Generational Garbage Collection for C++ Targeted to Sparc Architectures, Master’s Degree.


