Reprinted from the *13th International Conference on Compiler Construction*

Declarative Composition of Stack Frames

Christian Lindig[1] and Norman Ramsey[2]

[1] Universität des Saarlandes, 66 123 Saarbrücken, Germany
lindig@cs.uni-sb.de
[2] Division of Engineering and Applied Sciences
Harvard University, Cambridge, MA 02138, USA
nr@eecs.harvard.edu

Abstract. When a compiler translates a procedure, it must lay out the procedure's stack frame in a way that respects the calling convention. Many compilers respect the convention either by restricting the order in which slots can be allocated or by using different abstractions *ad hoc* for allocating in different regions of a frame. Such techniques cause the implementation of the layout to be spread over much of the compiler, making it difficult to maintain and verify. We have concentrated the implementation of layout into a single, unifying abstraction: the block. The block abstraction decouples layout from slot allocation. Stack-frame layout is specified in one central place, and even complex layouts are achieved by composing blocks using only two simple operators. Our implementation is used in the Quick C-- compiler to support multiple calling conventions on multiple architectures.

1 Introduction

In a compiled language, most of the information specific to one procedure is stored in that procedure's *stack frame*. The stack frame may hold such information as local variables, spilled temporaries, saved registers, and so on. The stack frame can even hold a source-language record or object, provided the object does not outlive the activation of its procedure. The advantage of allocating so much information into the stack frame is that everything can be addressed using only one pointer: the frame pointer. Each object or private datum is addressed by adding a different *offset* to the frame pointer.

Allocating memory on a stack performs so well that it is used in almost all compilers,[3] but it is nevertheless tricky:

– The offsets of some objects depend on the sizes of other objects, but the compiler must generate code before the size of every object is known. For example, the compiler might have to generate a reference to a record allocated on the stack before knowing how much stack space will be needed to hold callee-saves registers.

[3] For a notable exception see Appel (1992).

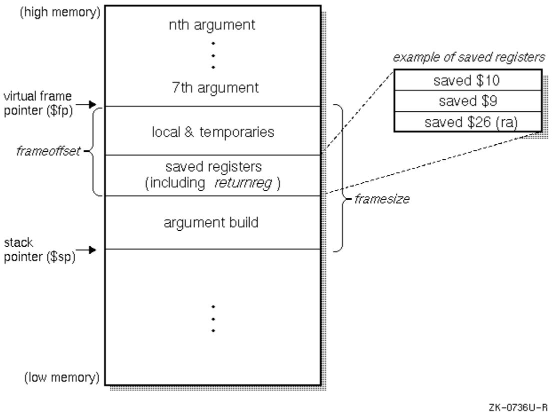

Fig. 1. Frame layout for the Tru64 Unix platform, from the online documentation of the *Assembly Language Programmer's Guide* (Compaq, 2000, Chapter 6).

- The compiler is not completely free to decide on the layout of a stack frame. To interoperate with other compilers or with debuggers, garbage collectors, and other tools that may walk the stack, the compiler must lay out the stack frame in a way that respects an architecture-specific calling convention (Compaq, 2000; Intel, 2003). Such a convention typically divides a frame into *regions*, each of which holds one kind of data: arguments or saved registers, for example.
- The compiler is not given the contents of all regions at once; instead, it must allocate stack space incrementally, one slot at a time. If the time at which a slot is allocated determines its position in the stack frame, the requirement to respect the calling convention places a heavy constraint on the order in which different parts of the compiler may execute.

The ideal compiler should be prepared to allocate any slot in any region at any time from any part of the compiler, all while making sure that the emerging frame layout respects the calling convention.

Today's compilers manage this problem, but at a cost in complexity. Compilers use different abstractions for different kinds of slots, and these abstractions are spread over the compiler. For example, the GCC compiler uses `temp_slot` values for temporaries and `arg_data` values for arguments, and these abstractions are defined in different modules (FSF, 2003). Similarly, the Objective Caml compiler OCAMLOPT uses different abstractions for slots for temporaries, incoming arguments, and outgoing arguments (Leroy et al., 2002).

The difficulty is understanding how these various abstractions interact as they form the frame. Frame layout is almost always specified by a diagram like that in Fig. 1, but in practice it is almost impossible to find the implementation of this diagram, because the implementation is spread over the compiler. Even given both the diagram and the compiler's source code, it can be quite difficult to verify that frame layout is correct—let alone change it. And such changes can be useful; for example, by adding a "canary" in front of a saved return address, one can protect against buffer overruns (Cowan et al., 1998). Other tasks that are unnecessarily difficult include arguing that the frame is as small as possible or experimenting with alternative layouts for improved performance (Davidson and Whalley, 1991). These difficulties are multiplied when a compiler supports multiple back ends or multiple calling conventions.

To address these concerns, we have developed a unifying abstraction that supports incremental allocation of stack slots in any order. The abstraction is based on declarative, hierarchical composition of memory blocks. It is unifying in that a slot, a region, and even the frame itself can each be represented as a block. The layout of the stack frame is specified using two block-composition operators: overlapping and concatenation. With this abstraction we have achieved the following results:

- Frame layout is specified using an algebraic description that corresponds directly to the sorts of diagrams found in architecture manuals (e.g., Fig. 1). The specification appears in one central place in the compiler.
- As a bonus, the layout can even be specified at compile time, instead of the more typical compile-compile time. Such flexibility facilitates experiments with frame layouts.
- Block creation, addressing, and composition are decoupled. Blocks can be created and composed in any order and any part of the compiler, thus enabling the compiler writer to run the parts of the compiler in any order, independent of the frame layout.

Our block-composition abstraction is implemented in the Quick C-- compiler, a new compiler for the portable assembly language C-- (Ramsey and Peyton Jones, 2000). The compiler produces code for Alpha, IA-32, IA-64, MIPS, and Power PC architectures. Its back end is controlled by the embedded scripting language Lua (Ierusalimschy et al., 1996; Ramsey, 2004), which allows a user to define frame layout as part of a calling convention.

2 Background

The layout of a stack frame must meet the requirements of the procedure calling convention. This convention is actually a contract among *four* parties: a calling procedure (the *caller*), a called procedure (the *callee*), a run-time system, and an operating system. All four parties must agree on how space will be allocated from the stack and how that space will be used: A procedure needs stack space for saved registers and private data, an operating system needs stack space to hold machine state ("signal context") when an interrupt occurs, and a run-time system needs to walk a stack to inspect and modify the state of a suspended computation. In addition to sharing the stack with the other parties, a caller and callee must agree on how to pass arguments and results. This paper focuses on the layout of a stack frame; a companion paper (Olinsky et al., 2004) addresses

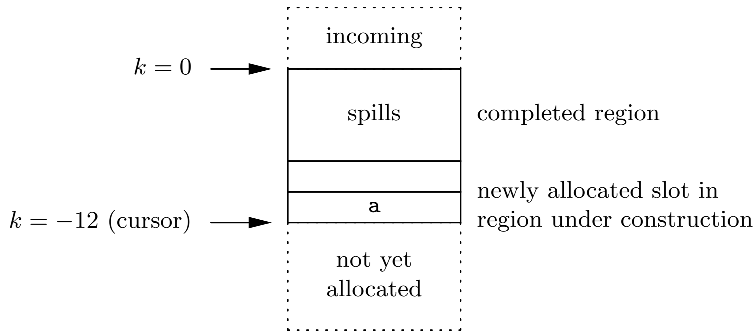

Fig. 2. Contiguous allocation of a stack frame using a cursor

the other big part of a calling convention: how procedures use registers to pass arguments and results.

For our purposes, then, the key fact about a calling convention is that it divides a stack frame into regions. A single region contains one kind of data, such as saved registers, arguments passed on the stack, source-language objects, and so on. A region is a sequence of *slots*, where each slot holds a single datum. The slot is the unit of allocation; for example, when the compiler discovers it needs to pass an argument on the stack, it allocates a slot to hold that argument.

At run time, a slot is read and written using an address of the form $p + k$, where p is typically either the stack pointer or the frame pointer, and k is a constant offset. A key question is when k is known. The answer is determined by the allocation strategy.

- One strategy is to allocate slots consecutively from the top of the frame to the bottom, as shown in Fig. 2. The compiler maintains a *cursor*, which keeps track of the amount of space allocated. To allocate a slot, the compiler advances the cursor by the size of the slot. Should the slot require some alignment, the compiler aligns the cursor beforehand.
  When slots are allocated using a cursor, k is known as soon as a slot is allocated. Knowing k immediately makes it easy to emit instructions that refer to the slot, but this allocation technique couples frame layout to the execution of the compiler: Allocations must be executed in an order consistent with the calling convention. This restriction can lead to contortions; for example, the LCC compiler requires the same compiler phases to execute in different orders depending on the target architecture (Fraser and Hanson, 1995).
- The other strategy is to allocate slots from different regions independently. In this strategy, k cannot be known until the sizes of earlier regions are known, i.e., not until allocation is complete. Not knowing k means that in an instruction that refers to the slot, k must be represented by a symbolic address, which must be replaced later. But this allocation technique makes frame layout independent of the execution of the compiler: Slots can be allocated in any order.

We use the second strategy, which maximizes the compiler writer's freedom. To make it easy to compute k, we introduce a new abstraction: the *block*.

3 Blocks and block composition

A block can represent a single slot, a region of the stack frame, a group of regions, or even the entire frame. Whatever it represents, a block is completely characterized by four properties:

– A *size* s, in bytes
– An *alignment* a, also in bytes
– A *base address* b, which is a symbolic expression
– A set of *constraints* c, where each constraint is a linear equation over base addresses

Size, alignment, and base address describe a region of memory. The base address is constrained by the alignment: $b \bmod a = 0$. After frame layout is determined, each base address will be equivalent to an addressing expression of the form $p + k$, e.g., $b = p + k_b$. The constraints express relationships, as described below, among base addresses of blocks that are in the same frame.

To form stack frames, blocks are composed in conventional ways, but ultimately convention is based on interference. Two blocks interfere if they are live at the same time, in which case they cannot share space. Interfering blocks are therefore *concatenated*. For example, source-language variables and spilled temporaries might be concatenated. If two blocks do not interfere, they are never live at the same time, in which case they can share space. Non-interfering blocks are therefore *overlapped*. For example, arguments passed on the stack at two different call sites might be overlapped. Both the concatenation and overlapping operations are designed so that a composed block contains its constituents, and correct alignment of the composed block implies correct alignment of the constituents. A composed block is as large as necessary, but no larger.

The ideas of concatenation and overlapping are simple, but some details are tricky. We therefore present both operations in detail, with formal semantics. In these presentations, we use two forms of notation. In short form, we write a block with size s, alignment a, and base address b as $s\mathtt{a}a@b$, pronounced "s bytes aligned a at b." For example, $12\mathtt{a}4@\mathtt{data}$ means a 16-byte block aligned on a 4-byte boundary at address `data`. We write constraints, if any, on the side. In long form, we use functions *size*, *base*, *align*, and *constr* to refer to the four properties of a block. The two forms are related by the equations $\mathit{size}(s\mathtt{a}a@b) = s$, $\mathit{align}(s\mathtt{a}a@b) = a$, and $\mathit{base}(s\mathtt{a}a@b) = b$.

As examples, we use blocks named after regions of a stack frame: block `in` holds incoming arguments, block `out` holds outgoing arguments, block `spills` holds spilled temporaries, block `conts` holds "continuations" (used to implement exceptions in C`--`), and block `data` holds user-allocated data.

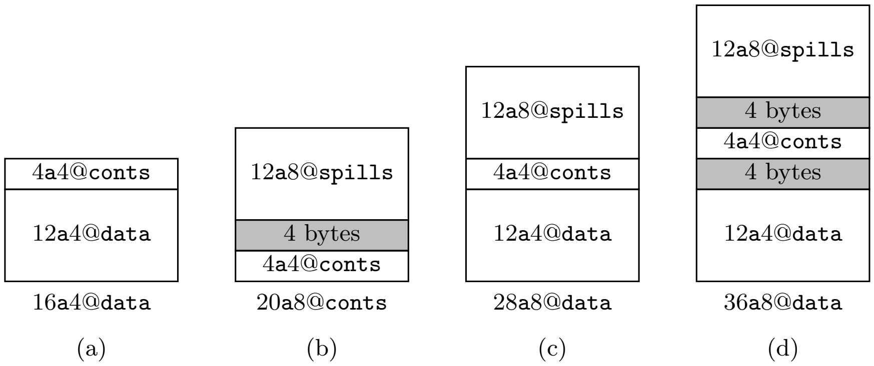

Fig. 3. Concatenation of memory blocks that form the private part of a frame. Notation 12a4@`data` means a block of size 12, aligned on a 4-byte boundary, at address `data`; constraints are not shown. Padding (shaded) is required to preserve alignment. Padding is sensitive to order; for example, composition (d) contains the same blocks as composition (c), yet is much larger.

3.1 Concatenation

We write concatenation using the infix operator $\oplus$; for example, we write the concatenation of 12a4@`data` and 4a4@`conts` as 12a4@`data` $\oplus$ 4a4@`conts`, and the second block begins where the first block ends (Fig. 3a). The address of the concatenation is the address of the first block, so 12a4@`data` $\oplus$ 4a4@`conts` is equivalent to 16a4@`data` with the constraint `conts` = `data` + 12. Because the size of 12a4@`data` is 12, a multiple of 4, base address `conts` is guaranteed to be 4-byte aligned provided address `data` is 4-byte aligned.

When concatenating blocks, we may need padding to guarantee the alignment of the right-hand block. For example, 4a4@`conts` $\oplus$ 12a8@`spills` = 20a4@`conts` with constraint `conts` + 8 = `spills`, where 4 bytes of padding are required (Fig. 3b). The padding makes `spills` − `conts` a multiple of 8.

As these examples show, proper alignment of a concatenation $lo \oplus hi$ guarantees proper alignment of lo and hi. The guarantee comes from two decisions: the alignment of the concatenation is the least common multiple of the alignments of lo and hi, and the distance between the base of lo and the base of hi is a multiple of the alignment of hi. The concatenation is therefore defined by the following equations:

$$\begin{aligned}
base(lo \oplus hi) &= base(lo)\\
size(lo \oplus hi) &= round_up(size(lo), align(hi)) + size(hi)\\
align(lo \oplus hi) &= lcm(align(lo), align(hi))\\
constr(lo \oplus hi) &= constr(lo) \cup constr(hi) \cup\\
&\quad \{base(hi) = base(lo) + round_up(size(lo), align(hi))\}\\
round_up(x, n) &= n \times \lfloor (x + n - 1)/n \rfloor
\end{aligned}$$

Concatenation has one awkward property: When three or more blocks are concatenated, padding can depend on the order of composition. In other words, concatenation is not associative. For example, columns c and d of Fig. 3 show two different compositions of the blocks $12\mathtt{a}4@\mathtt{data}$, $4\mathtt{a}4@\mathtt{conts}$, and $12\mathtt{a}8@\mathtt{spills}$. The composition $(12\mathtt{a}4@\mathtt{data} \oplus 4\mathtt{a}4@\mathtt{conts}) \oplus 12\mathtt{a}8@\mathtt{spills} = 28\mathtt{a}8@\mathtt{data}$, with constraints $\mathtt{data} + 12 = \mathtt{conts}$ and $\mathtt{data} + 16 = \mathtt{spills}$, and the composition $12\mathtt{a}4@\mathtt{data} \oplus (4\mathtt{a}4@\mathtt{conts} \oplus 12\mathtt{a}8@\mathtt{spills}) = 36\mathtt{a}8@\mathtt{data}$, with constraints $\mathtt{conts}+8 = \mathtt{spills}$ and $\mathtt{data}+16 = \mathtt{conts}$. The first composition needs no padding, but the second needs 8 bytes. The reason is that the second composition uses a concatenation on the right: The alignment of the right-hand block determines the padding, and a composite block has an alignment constraint at least as large as the constraints of its constituents.

Luckily, the nonassociativity of concatenation presents no problems in practice. In our compiler, we specify frame layout using an operation that concatenates a *list* of blocks. This operation applies the binary operator $\oplus$ in a left-associative way, which minimizes padding. Associativity is not an issue in cursor-based stack allocation because there is no choice of order; padding is added on demand for each slot, and the results are equivalent to our implementation.

3.2 Overlapping

When two blocks are never live at the same time, they can overlap, sharing memory. When two overlapping blocks are of different sizes, it may be possible to place the smaller block anywhere within the larger block, but in practice, the smaller block is placed at the bottom of the larger block, so that the two base addresses are equal. We call this operation *overlap low*. For completeness, we also discuss the *overlap high* operation, in which the smaller block is placed at the top of the larger block.

The most common use of the overlap-low operation is to create an argument-build area, which holds arguments that are to be passed to a callee on the stack. The area must be large enough to hold the arguments for the most complicated call in the procedure's body. Such an area can be created by allocating each call site's arguments into a different block, then overlapping all such blocks.

For example, on the Tru64 Unix platform the first six arguments are passed in registers and later arguments on the stack. Let us assume a procedure body with two call sites, where the first passes one argument on the stack, and the second passes two arguments on the stack. Each argument is 64 bits, i.e., 8 bytes. According to the ABI (Compaq, 2000), a region holding arguments must be 16-byte aligned. Therefore, the first call reserves a block $8\mathtt{a}16@\mathtt{out}_1$, and the second a block $16\mathtt{a}16@\mathtt{out}_2$. When the compiler later assembles the frame, it overlaps these blocks at their low end and arrives at a $16\mathtt{a}16@\mathtt{out}_1$ block and constraint $\mathtt{out}_1 = \mathtt{out}_2$.

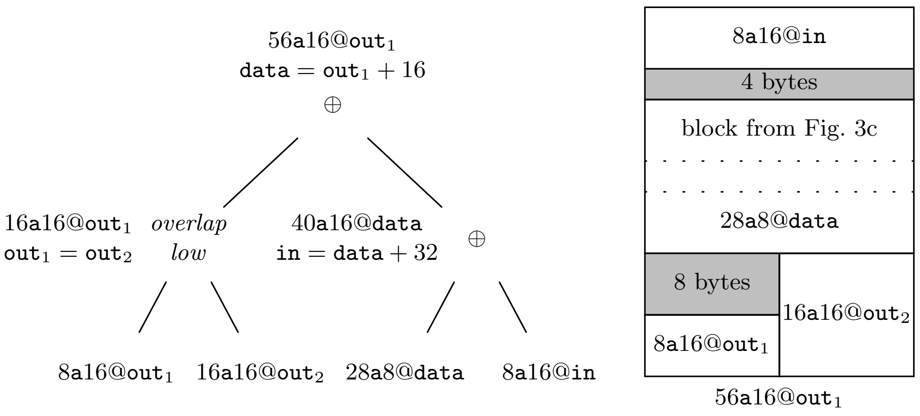

Fig. 4. Composition of four blocks into a frame: incoming arguments, private data (from Fig. 3c), and two blocks for outgoing arguments. The left side shows the tree-shaped construction, the right side the resulting block layout. Concatenation is denoted by $\oplus$, overlapping at the low end by *overlap_low*. Blocks 16a16@out$_1$ and 16a16@out$_2$ overlap such that they align on their lower end (higher addresses are towards the top of the diagram). To guarantee the alignment of `in`, padding was added.

Overlapping two blocks at their low ends is easy. The result is large enough to hold either block and is aligned for both. The constraint specifies that both blocks have the same base address.

$$\begin{aligned}
base(overlap_low(x,y)) &= base(x)\\
size(overlap_low(x,y)) &= max(size(x), size(y))\\
align(overlap_low(x,y)) &= lcm(align(x), align(y))\\
constr(overlap_low(x,y)) &= constr(x) \cup constr(y) \cup \{base(x) = base(y)\}
\end{aligned}$$

Surprisingly, overlapping blocks at their high ends is possible only if the difference between the blocks' sizes is a multiple of the smaller block's alignment. Otherwise, the smaller block cannot be placed at the high end of the larger block while simultaneously maintaining proper alignment of both base addresses. Padding cannot help: Padding the smaller block at the top would move it away from the high end of the larger block, and padding the larger block at its base would destroy its base alignment. The *overlap_high* operator is therefore defined only when its precondition is satisfied.

$$\begin{aligned}
overlap_high(x,y) &= \bot \quad \text{if } (size(y) - size(x)) \bmod align(x) \neq 0\\
base(overlap_high(x,y)) &= base(y) \quad \text{if } size(x) \leq size(y)\\
base(overlap_high(x,y)) &= base(x) \quad \text{if } size(x) > size(y)\\
size(overlap_high(x,y)) &= max(size(x), size(y))\\
align(overlap_high(x,y)) &= lcm(align(x), align(y))\\
constr(overlap_high(x,y)) &= constr(x) \cup constr(y) \cup\\
&\quad \{base(x) + size(x) = base(y) + size(y)\}
\end{aligned}$$

⟨*Lua compiler configuration*⟩≡

```
function Alpha.layout["C"](dummy, proc)
    local blocks = Stack.blocks(proc)
    blocks.in    = Block.overlap_low(64, blocks.youngblocks)
    blocks.out   = Block.overlap_low(64, blocks.oldblocks)
    local layout =               -- <-- high memory
        { blocks.in              -- incoming arguments
        , blocks.vfp             -- virtual frame pointer
        , blocks.spills          -- temporaries
        , blocks.conts           -- continuations (for exceptions)
        , blocks.data            -- user-allocated data
        , blocks.sp              -- stack pointer
        , blocks.out             -- argument build
        }                        -- <-- low memory
    local block = Block.cat(64, layout)
    block = Block.adjust(block)  -- round up size to multiple of 16
    Stack.freeze(proc, block)    -- resolve addresses
    return 1
end
```

Fig. 5. Lua procedure, from Quick C--, that defines the frame layout for the Tru64 Unix platform. Compare with Fig. 1.

The precondition on *overlap_high* depends on both blocks, and the only easy way to ensure that *overlap_high* is possible is to pick a basic alignment (such as word size) for all blocks and to make sure every block's size is a multiple of that alignment. This restriction may explain why *overlap_high* is rarely used in conventional frame layouts.

4 Specification of Frame Layout

The block-composition operators described above suffice to create a specification of frame layout that not only can be executed in the compiler but also can be compared with a diagram found in an architecture-specific ABI manual. In the Quick C-- compiler, this specification takes the form of a function written in the embedded scripting language Lua (Ierusalimschy et al., 1996; Ramsey, 2004), which does for Quick C-- what ELisp does for Emacs. As an example, Fig. 5 shows Lua procedure `Alpha.layout`, which composes the frame for the C calling convention on the Alpha Tru64 Unix platform. This procedure is executed each time the compiler lays out a stack frame. It lays out the frame using the table `blocks`, which contains all the blocks for the procedure `proc` under translation. The procedure begins composing the frame by overlapping all the blocks that can share space. For example, list `blocks.youngblocks` holds a block for each call site; each block reserves space for outgoing arguments passed on the stack. `Alpha.layout` overlaps them at their low end to form the argument-build area `blocks.out`. Similarly, `blocks.oldblocks` holds blocks for incoming arguments and any results that are returned on the stack; these blocks are overlapped to form `blocks.in`. The procedure then forms the list `layout`, which contains the blocks that represent the regions of the stack frame, in the order in which they should be laid out. Regions `blocks.vfp` and `blocks.sp` are empty blocks whose base addresses correspond to the virtual frame pointer and the stack pointer, respectively. The concatenation is done by `Block.cat`, which repeatedly applies concatenation operator $\oplus$, associating to the left.

Comparing Fig. 5 with the official specification in Fig. 1 reveals that the correspondence between our implementation and the manual is close but not exact. Our `blocks.in` and `blocks.out` correspond exactly to the incoming arguments and argument-build area of Fig. 1. Similarly, our `blocks.vfp` and `blocks.fp` correspond exactly to the pointers in Fig. 1. But our treatment of locals, temporaries, and saved registers is different. Because our compiler works with a nonstandard run-time system, we do not need to distinguish saved registers from spilled temporaries. And because of the way our compiler does allocation, we find it convenient to distinguish two kinds of locals: `blocks.data` and `blocks.conts`. These kinds of detailed differences would be very difficult to extract from a conventional implementation of frame layout. It is the ability to compare the implementation directly with a picture that is missing in other compilers. As a bonus, blocks make it trivial to change frame layout.

5 Implementation

An implementation can use blocks in many ways: one block per region, one block per slot, or even one block per slot in some regions and one block per region in other regions. Which arrangement is best depends on whether it is necessary to control the order of slots in a region; for example, it might be desirable to control the order in which callee-saves registers are stored. Different arrangements require slight variations, but for concreteness we assume one block per region, which is the arrangement used in our implementation.

Given one block per region, our compiler follows these steps:

1. For each region of the stack frame, a unique symbolic address is created.
2. Slots are allocated from various regions, in any order. Each region is given a mutable data structure that remembers alignment and uses an internal cursor to allocate from that region only. A slot's address is a function of the region's cursor and symbolic address. The address is then used to generate intermediate code.
3. When allocation is complete, each region (or each slot) is identified with a fresh, unconstrained block. The block is given the symbolic address of its region. The block's size and alignment come from the region's internal cursor and remembered alignment, and the block has no constraints.
4. If a stack pointer or frame pointer points to a particular location in the frame, an empty block is created to represent that location.
5. Using concatenation and overlapping operators, blocks are composed to form a single block, which represents the frame. This step is shown in Fig. 5.

6. The frame's constraints are solved, and the solution provides the value of each symbolic address. If any block composition was illegal, the problem shows up here as unsolvable constraints.
7. The value of each symbolic address is substituted in the intermediate code. This step and the previous step are performed in Fig. 5 by `Stack.freeze`.

Most of the implementation details are straightforward; for example, the implementation of blocks follows directly from the equations in Sect. 3, and it takes less than 100 lines of code. But a few details are worth discussing here.

Our compiler bases stack addresses on a virtual frame pointer, written `vfp`, so when we create a region, we know its address has the form $\texttt{vfp} + k$, but we don't know k. We therefore give a new block the symbolic address $\texttt{vfp} + v$, where v is a fresh *late compile-time constant.* A late compile-time constant is an integer whose value is not known until late in compilation. When created, it is represented by a string, and when the frame layout is known, that string is replaced by the integer for which it stands.

There is one exception to the rule that each block has a symbolic address of the form $\texttt{vfp} + v$. The empty block that represents the location of the virtual frame pointer has the symbolic address `vfp`. This block appears in Fig. 5 as `blocks.vfp`.

When n blocks are composed to form the stack frame, it requires $n-1$ block-composition operations. As shown in Sect. 3, each operation adds one equation to the set of constraints on a block. And because one block has the symbolic address `vfp` and each of the other $n-1$ blocks has a symbolic address of the form $\texttt{vfp}+v$, the `vfp`s cancel and we wind up with a system of $n-1$ equations in $n-1$ unknowns. These equations are solved using an iterative algorithm similar to those presented by Knuth (1986, §585) and Ramsey (1996). Experiments show that this algorithm is efficient in practice.

We have considered two extensions to the simple picture presented above. In the first extension, a stack pointer is used as well as a frame pointer, and some blocks addresses have the form $\texttt{sp}+k$ while others have the form $\texttt{fp}+k$. If the stack pointer moves, perhaps because the compiler supports `alloca()`, this arrangement amounts to solving two independent systems of equations. But if the stack pointer does not move, so the frame has a fixed size, this arrangement requires an extra equation $\texttt{fp} - \texttt{sp} = v_{frame}$, where v_{frame} is a late compile-time constant that stands for the frame size. When the equations are solved, the solution for v_{frame} gives the actual frame size.

The other extension we have considered is to minimize frame size by using register-allocation techniques to manage slots. In this extension, each slot gets its own block, and the compiler must compute an interference graph for these blocks. It can then color the graph and overlap blocks that share the same color.

6 When Things Go Wrong

Not every block composition yields a useful frame layout. Potential errors include

1. Concatenating a block with itself, directly or indirectly, which may produce an unsolvable constraint on the address of that block
2. Providing no block with a known address, so there are more unknowns than equations
3. Providing two blocks with known, inconsistent addresses, so there are more equations than unknowns
4. Using a single late compile-time constant in multiple blocks, so there are more equations than unknowns

Each of these errors manifests as failure in the equation solver, and it can be hard to track such an error to its source. Because solvability is a global property of the whole composition, it is hard to detect violations early. Our most important debugging aid is a routine that prints blocks and their equations. Such printouts have helped us debug frame layouts quickly. It also helps to choose good names for late compile-time constants.

Errors are detected at compile time, when the compiler actually lays out a frame for a procedure. Frame layout is therefore *not guaranteed* to be successful. But if the compiler lays out one frame successfully, it is *extremely likely* to lay out all frames successfully. The reason is that if a composition is unsolvable in general, it is unsolvable for almost all block sizes. (The “almost” is because certain incorrect compositions, such as concatenating a block with itself, may be solvable for one block size, such as zero.) To effectively eliminate the possibility of a compile-time error in frame layout, then, one could test layout with random block sizes. In practice, such testing appears to be unnecessary.

7 Related Work

Other compilers lay out stack frames using either a cursor or a mix of heterogenous abstractions. For example, the LCC C compiler uses a cursor (Fraser and Hanson, 1995). It allocates memory linearly, so the offset of each slot is known when the slot is allocated. When LCC leaves a local scope or a call site, it rolls back the cursor so the relevant slots can be reused. Therefore, when LCC compiles a function with multiple call sites, the slots used to pass arguments share stack space: The “argument-build area” is large enough to hold arguments passed at any call site. LCC tracks the size by using a “high-water mark” to hold the maximum value of the cursor. LCC’s correctness depends on executing allocations and rollbacks in exactly the right order. By contrast, blocks make correct layout independent of the order of allocation.

The GNU C compiler (GCC) defines virtual hardware registers that address arguments and private data in the frame; these registers are replaced late in compilation by a constant offset from either the frame pointer or the stack pointer, depending on the target (FSF, 2003). When a virtual register points to private data, that space is represented by a type `temp_slot` defined in `function.c`. Values of type `temp_slot` can be allocated and freed; adjacent freed slots are combined to avoid fragmentation. When a virtual register points to arguments, that space is allocated using a cursor implemented in `calls.c`. Module `calls.c` also maintains a bitmap for allocated memory and high/low marks to implement overlapping similar to LCC.

GCC's virtual hardware registers and their substitution achieve the same effect as late compile-time constants: They free the compiler from order constraints imposed by the frame layout. Joining adjacent `temp_slot` values resembles concatenation in our representation. But it is only defined for a part of the frame that holds temporaries; other parts are maintained by a different module with different mechanisms. By using a single, uniform representation (blocks), we make it possible to define and change frame layout in one central place (Fig. 5).

8 Conclusions

A compiler should not be required to allocate stack slots in the linear order demanded by a frame layout. For laying out an entire stack frame, therefore, using a cursor is inferior to using blocks, which enable a compiler to allocate stack slots in any order. But when order of slots within a region does not matter, it is very convenient to use a cursor in each region.

Defining the layout of a block by composing blocks with `cat` and `overlap` is declarative—it requires the user only to define the size and alignment of simple blocks. The layout of any intermediate block is automatically deduced. The block abstraction is conceptually close to layout specifications and diagrams as they used in architecture manuals (Compaq, 2000, Chapter 6; Intel, 2003, Chapter 6), and block layouts can be centralized *and* lifted from the compiler's innards to its surface, as shown in Fig. 5. Giving a user control over the layout of frames makes it easier to do experiments such as specializing calling conventions for optimized performance (Davidson and Whalley, 1991) or changing stack layout to combat buffer overruns that overwrite return addresses (Cowan et al., 1998).

Overlapping blocks at their high end is a dubious concept. It appears to be a natural dual to overlapping at the low end, but it is not possible in general. Small wonder that typical C calling conventions, such as IA-32, IA-64, MIPS, PowerPC, and Tru64 Unix, use overlap-low exclusively. Overlap-low is a natural fit for conventions in which the stack grows downward and later arguments are passed at higher addresses.

Blocks perform as well as other approaches. Stack layouts are, if not the same, equivalent in cost to layouts produced by other approaches. Compile-time costs are negligible. Constraint solving may sound scary, but the constraints are simply linear equations; indeed, a compiler that optimizes array access may already contain a suitable solver.

The only potential drawback of blocks is in debugging: It can be awkward to debug a bad system of equations instead of debugging frame layout directly. But the awkwardness fades after a little experience, and with the aid of the tools described in Sect. 6 we have had little difficulty debugging frame layouts. And we have no reason to believe that distributed allocation, with layout dependent on order, is any easier to debug.

In a retargetable compiler, which requires slight variations in layout for different platforms, blocks have made it easy for us to visualize, understand, and implement these variations. Because layout is centralized, it is easy to implement a new layout using the traditional method of copying and modifying an existing one. Overall, using blocks has made it significantly easier to deal with calling conventions.

Acknowledgements This work has been supported by NSF grants CCR-0096069 and ITR-0325460 and by a gift from Microsoft Research. Lex Stein, Stephan Neuhaus, Wolfram Amme, Andreas Rossberg, Andreas Zeller, and the 319 writing group provided helpful comments on drafts of the manuscript.

Bibliography

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

Compaq. *Tru64 Unix – Assembly Language Programmer's Guide*. Compaq Computer Corporation, 2000.

Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 63–78, Berkeley, January 1998. Usenix Association.

Jack W. Davidson and David B. Whalley. Methods for saving and restoring register values across function calls. *Software — Practice and Experience*, 21 (2):149–165, February 1991.

Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

FSF. *GCC Internals Manual*. Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA, 2003. URL `http://gcc.gnu.org/`. Corresponds to GCC 3.4.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software — Practice and Experience*, 26(6):635–652, 1996. URL `http://www.lua.org/`.

Intel. *IA-32 Intel Architecture Software Developers's Manual, Vol. 1*. Intel Corporation., P.O. Box 7641, Mt. Prospect, IL 60056, 2003.

Donald Ervin Knuth. *METAFONT: The program*, volume D of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System 3.06: Documentation and User's Manual*. INRIA, 2002. Available from `http://caml.inria.fr`.

Reuben Olinsky, Christian Lindig, and Norman Ramsey. Staged allocation: Engineering the specification and implementation of procedure calling conventions. Technical Report TR-02-04, Division of Engineering and Applied Sciences, Harvard University, 2004.

Norman Ramsey. A simple solver for linear equations containing nonlinear operators. *Software — Practice and Experience*, 26(4):467–487, April 1996.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 2004. To appear. Preliminary version appeared on pages 6–14 of *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003.

Norman Ramsey and Simon L. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):285–298, May 2000.