The Sea of Nodes

Russel Arbore

A Simple Graph-Based Intermediate Representation

Cliff Click
cliffc@hpl.hp.com

Michael Paleczny
mpal@cs.rice.edu

Abstract

We present a graph-based intermediate representation (IR) with simple semantics and a low-memory-cost C++ implementation. The IR uses a directed graph with labeled vertices and ordered inputs but unordered outputs. Vertices are labeled with opcodes, edges are unlabeled. We represent the CFG and basic blocks with the same vertex and edge structures. Each opcode is defined by a C++ class that encapsulates opcode-specific data and behavior. We use inheritance to abstract common opcode behavior, allowing new opcodes to be easily defined from old ones. The resulting IR is simple, fast and easy to use.

1. Introduction

Intermediate representations do not exist in a vac-

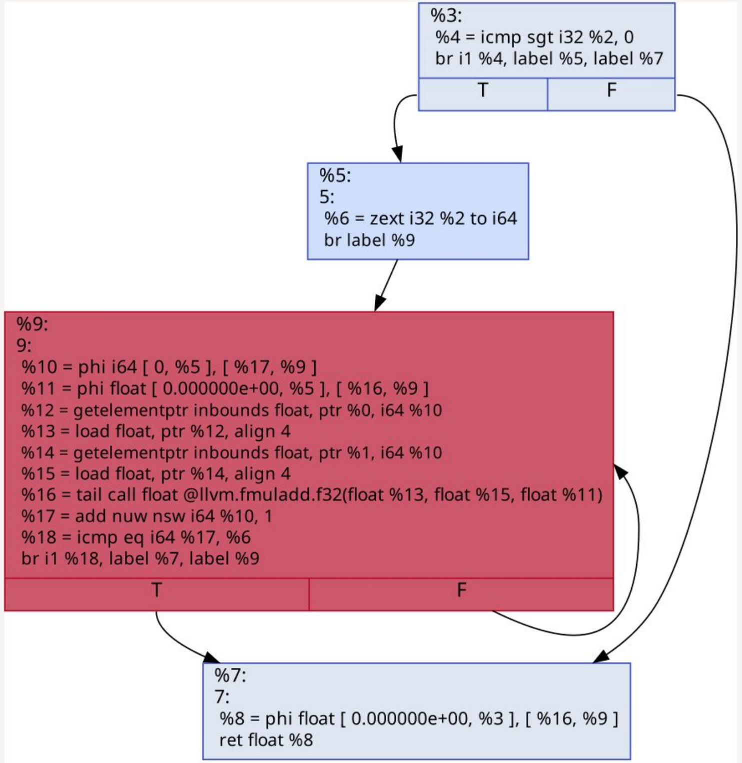
understand, and easy to extend. Our goal is a representation that is simple and light weight while allowing easy expression of fast optimizations.

This paper discusses the intermediate representation (IR) used in the research compiler implemented as part of the author's dissertation [8]. The parser that builds this IR performs significant parse-time optimizations, including building a form of *Static Single Assignment* (SSA) at parse-time. Classic optimizations such as *Conditional Constant Propagation* [23] and *Global Value Numbering* [20] as well as a novel global code motion algorithm [9] work well on the IR. These topics are beyond the scope of this paper but are covered in Click's thesis.

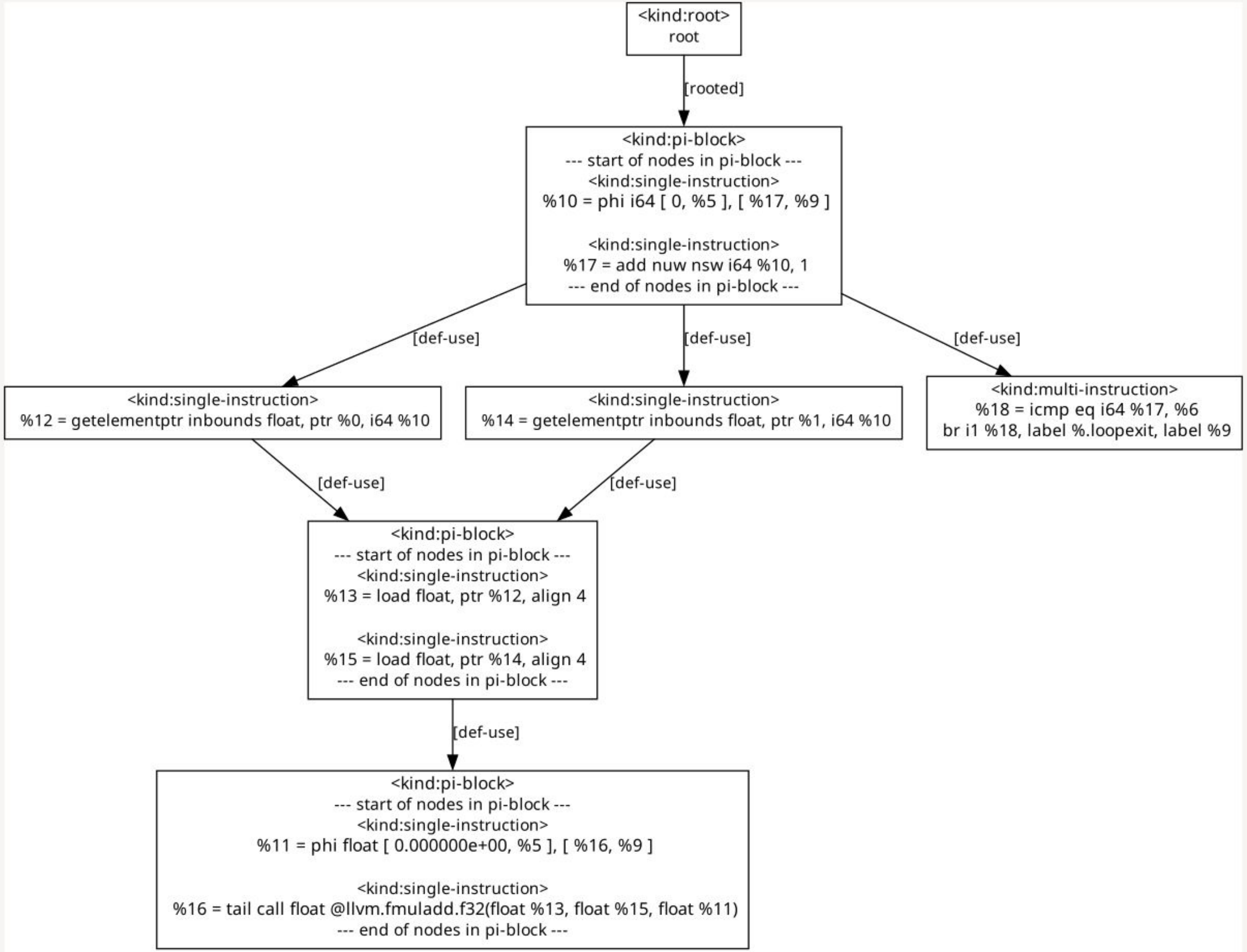
The intermediate representation is a graph-based,

In the beginning*, there was CFG + SSA

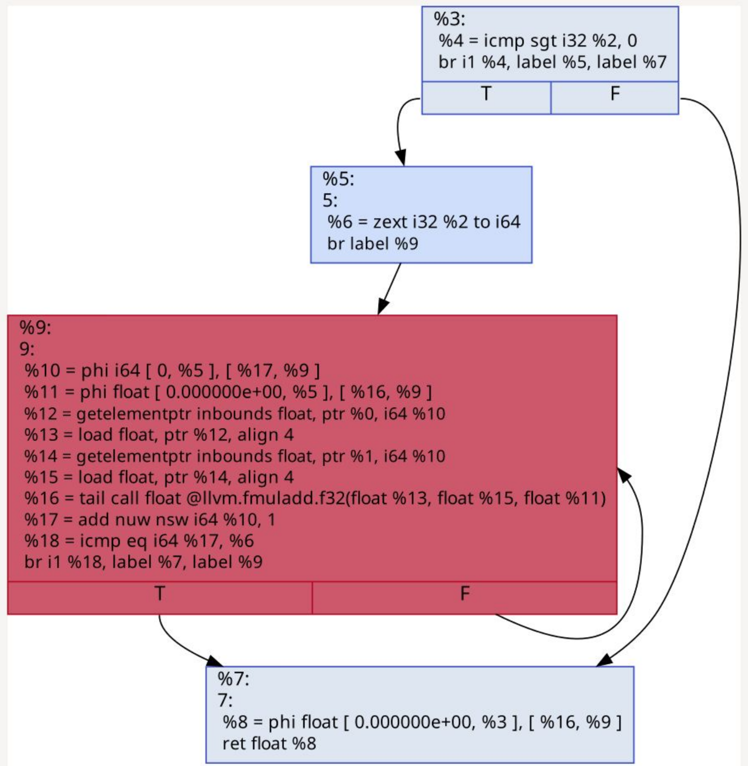
```
define dso_local float @dot(ptr %0, ptr %1, i32 %2) {
  %4 = icmp sgt i32 %2, 0
  br i1 %4, label %5, label %7

5:                                                ; preds = %3
  %6 = zext i32 %2 to i64
  br label %9

7:                                                ; preds = %9, %3
  %8 = phi float [ 0.000000e+00, %3 ], [ %16, %9 ]
  ret float %8

9:                                                ; preds = %5, %9
  %10 = phi i64 [ 0, %5 ], [ %17, %9 ]
  %11 = phi float [ 0.000000e+00, %5 ], [ %16, %9 ]
  %12 = getelementptr inbounds float, ptr %0, i64 %10
  %13 = load float, ptr %12, align 4
  %14 = getelementptr inbounds float, ptr %1, i64 %10
  %15 = load float, ptr %14, align 4
  %16 = tail call float @llvm.fmuladd.f32(float %13, float %15, float %11)
  %17 = add nuw nsw i64 %10, 1
  %18 = icmp eq i64 %17, %6
  br i1 %18, label %7, label %9
}

declare float @llvm.fmuladd.f32(float, float, float)
```



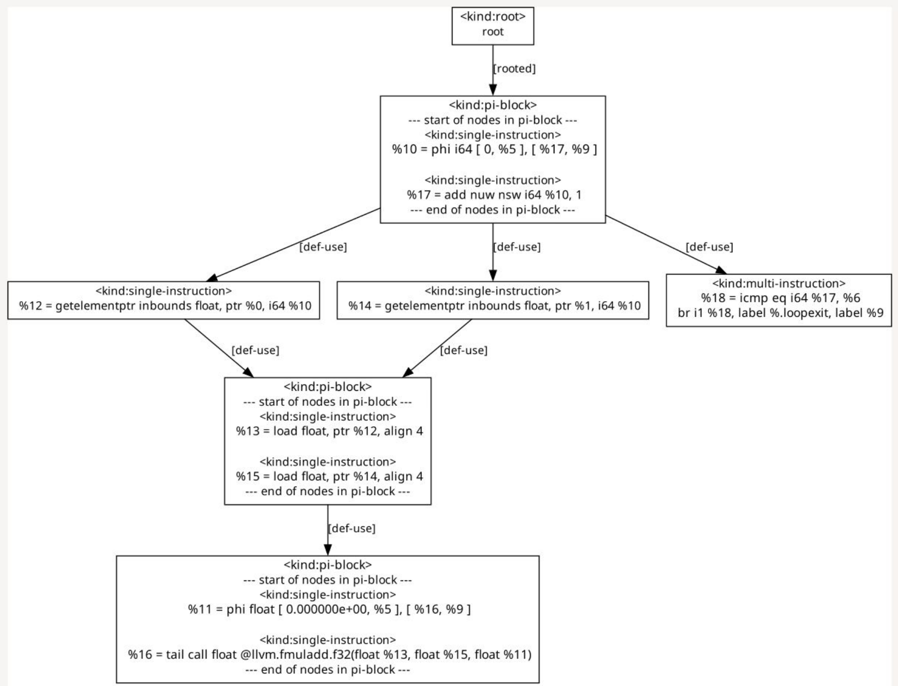
CFG for 'dot' function


<kind:root>
root
[rooted]
<kind:pi-block>
--- start of nodes in pi-block ---
<kind:single-instruction>
%10 = phi i64 [0, %5], [%17, %9]
<kind:single-instruction>
%17 = add nuw nsw i64 %10, 1
--- end of nodes in pi-block ---
[def-use]
[def-use]
[def-use]
<kind:single-instruction>
%12 = getelementptr inbounds float, ptr %0, i64 %10
<kind:single-instruction>
%14 = getelementptr inbounds float, ptr %1, i64 %10
<kind:multi-instruction>
%18 = icmp eq i64 %17, %6
br i1 %18, label %.loopexit, label %9
[def-use]
[def-use]
<kind:pi-block>
--- start of nodes in pi-block ---
<kind:single-instruction>
%13 = load float, ptr %12, align 4
<kind:single-instruction>
%15 = load float, ptr %14, align 4
--- end of nodes in pi-block ---
[def-use]
<kind:pi-block>
--- start of nodes in pi-block ---
<kind:single-instruction>
%11 = phi float [0.000000e+00, %5], [%16, %9]
<kind:single-instruction>
%16 = tail call float @llvm.fmuladd.f32(float %13, float %15, float %11)
--- end of nodes in pi-block ---

DDG for 'dot.'



CFG for 'dot' function



DDG for 'dot.'

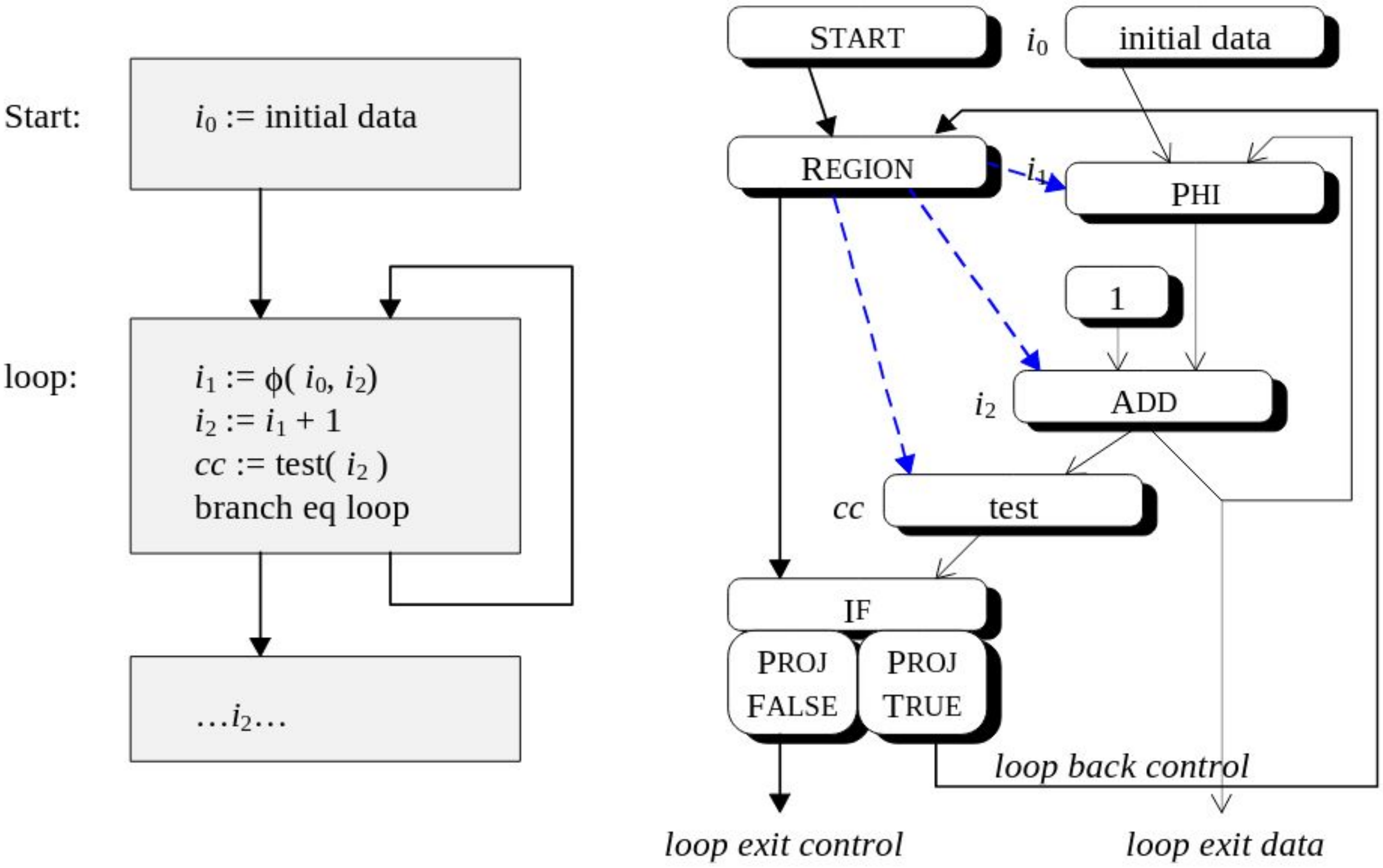


Figure 7 An example loop

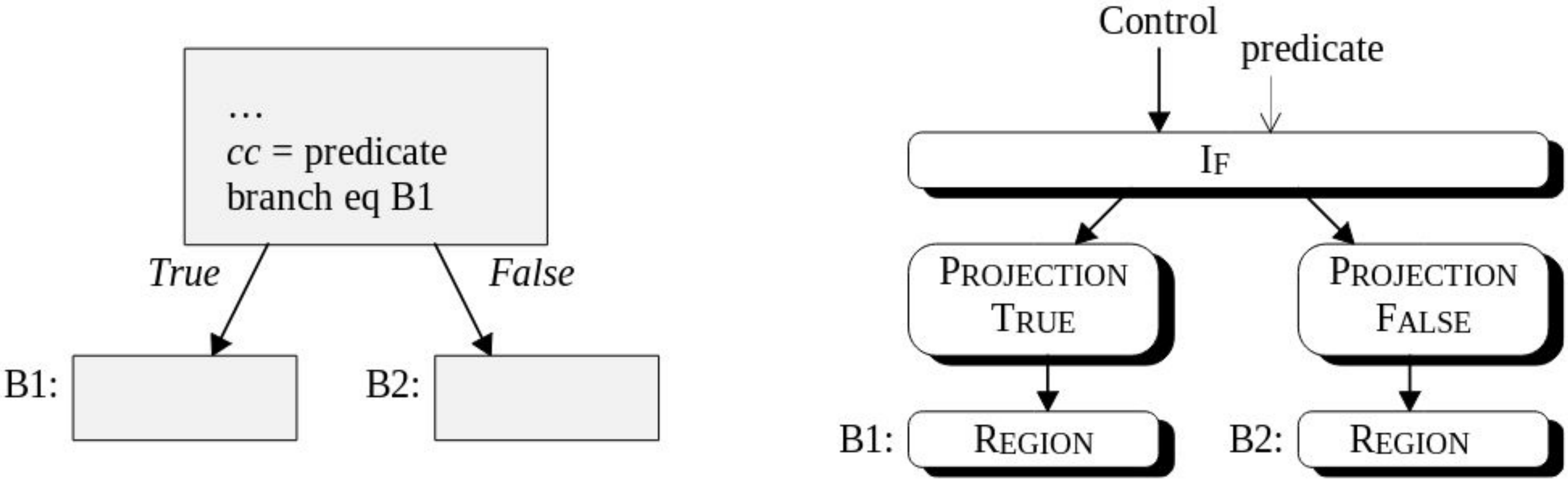
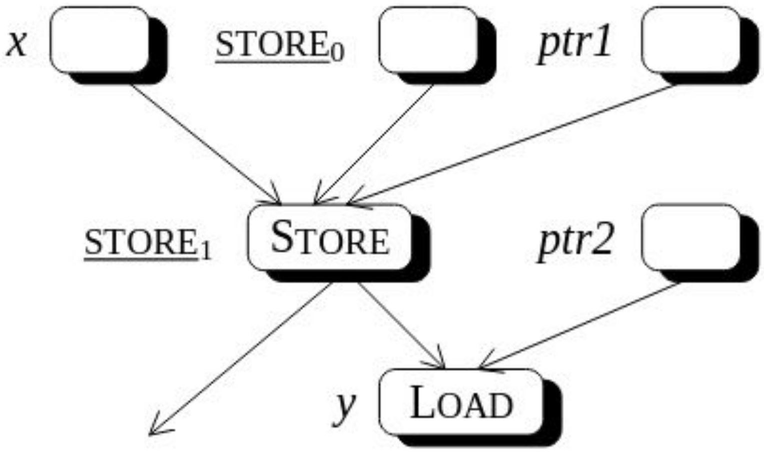


Figure 5 Projections following an If Node

2.5 Compound Values: Memory and I/O

We treat memory like any other value, and call it the STORE. The START node and a PROJECTION-STORE node produce the initial STORE. LOAD nodes take in a STORE and an address and produce a new value. STORE nodes take in a STORE, an address, and a value and produce a new STORE. PHI nodes merge the STORE like other values. Figure 6 shows a sample treatment of the STORE.




```
{
    x := …;            // Make some value
    *ptr1 := x;        // Store value to memory
    …
    y := *ptr2;        // Load some value
    …y…                // Use loaded value
    …
}
```

Figure 6 Treatment of memory (STORE)

The lack of anti-dependences[2] is a two-edged sword. Between STORE's we allow LOAD nodes to reorder. However, some valid schedules (serializations of the graph) might overlap two STOREs, requiring that all of memory be copied. Our serialization algorithm treats memory like a type of unique machine register with infinite spill cost. The algorithm schedules the code to avoid spills if possible, and for the STORE it always succeeds.

This design of the STORE is very coarse. A better design would break the global STORE into many smaller, unrelated STORE's. Every independent variable or array would get its own STORE. Operations on the separate STORE's could proceed independently from each other. We could also add some understanding of pointers [7].

Memory-mapped I/O (*e.g.*, **volatile** in C++) is treated like memory, except that both READ and WRITE nodes produce a new I/O state. The extra dependence (READs produce a new I/O state, while LOADs do not produce a new STORE) completely serializes I/O. At program exit, the I/O state is required, however, the STORE is not required. Non-memory-mapped I/O requires a subroutine call.

```
class Arena {                                // Arenas are linked lists of large chunks of heap
    enum { size = 10000 };                   // Chunk size in bytes
    Arena *next;                             // Next chunk
    char bin[size];                          // This chunk
    Arena( Arena *next ) : next(next) {}          // New Arena, plug in at head of linked list
    ~Arena() { if( next ) delete next; }     // Recursively delete all chunks
};
class Node {                                 // Base Node class
    static Arena *arena;                     // Arena to store nodes in
    static char *hwm, *max, *old;            // High water mark, limit in Arena
    static void grow();                      // Grow Arena size
    void *operator new( size_t x )           // Allocate a new Node of given size
    { if( hwm+x > max ) Node::grow(); old := hwm; hwm := hwm+x; return old; }
    void operator delete( void *ptr)         // Delete a Node
    { if( ptr = old ) hwm := old; }          // Check for deleting recently allocated space
};
Arena *Node::arena := NULL;                  // No initial Arena
char *Node::hwm := NULL;                     // First allocation attempt fails
char *Node::max := NULL;                     // … and makes initial Arena
void Node::grow()                            // Get more memory in the Arena
{
    arena := new Arena(arena);               // Grow the arena
    hwm := &arena→bin[0];                    // Update the high water mark
    max := &arena→bin[Arena::size];          // Cache the end of the chunk as well
}
```

Figure 14 Fast allocation with arenas