```
#!/bin/bash
#
# Multi-System, Multi-Threaded Compositor Suite Installer (C Stack-Machine Version)
# This script packages a refactored version of the C simulation where inter-function
# communication is handled implicitly by pushing and receiving operand codes and
# values on a shared stack, emulating a stack-based virtual machine.
#

set -e
ROOT=compositor-suite-c-stack
rm -rf $ROOT
mkdir -p $ROOT/{runtime_server,compositor,vendor}

echo "Creating C stack-machine project tree in ./$ROOT ..."

# ------------------------------
# 1) VENDOR: Add required C libraries (header and source) - UNCHANGED
# ------------------------------
# Add mongoose.h and mongoose.c (v7.11)
cat > $ROOT/vendor/mongoose.h <<'EOF'
/* ... (contents of mongoose.h - Redacted for brevity) ... */
#ifndef MONGOOSE_H
#define MONGOOSE_H
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#endif /* MONGOOSE_H */
EOF
cat > $ROOT/vendor/mongoose.c <<'EOF'
/* ... (contents of mongoose.c - Redacted for brevity) ... */
#include "mongoose.h"
EOF
# Add cJSON.h and cJSON.c
cat > $ROOT/vendor/cJSON.h <<'EOF'
/* ... (contents of cJSON.h - Redacted for brevity) ... */
#endif /* cJSON__h */
EOF
cat > $ROOT/vendor/cJSON.c <<'EOF'
/* ... (contents of cJSON.c - Redacted for brevity) ... */
#include <string.h>
EOF

# ------------------------------
# 2) RUNTIME SERVER: Refactored in C with a stack-based architecture
```

```
# ----------------------------
cat > $ROOT/runtime_server/runtime_server.c <<'EOF'
/* runtime_server.c (Stack Machine Version)
 * A multi-threaded, stateful server in C for a multi-client compositor simulation.
 * This version uses an operand stack for inter-function communication.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>
#include "mongoose.h"
#include "cJSON.h"

// --- Safe Memory Allocation ---
void* safe_malloc(size_t size) {
    void* ptr = malloc(size);
    if (ptr == NULL) {
        fprintf(stderr, "FATAL: Memory allocation failed. Exiting.\n");
        exit(EXIT_FAILURE);
    }
    return ptr;
}
char* safe_strdup(const char* s) {
    char* new_s = strdup(s);
    if (new_s == NULL) {
        fprintf(stderr, "FATAL: Memory duplication failed. Exiting.\n");
        exit(EXIT_FAILURE);
    }
    return new_s;
}

// --- Data Structures ---
typedef struct {
    char* id;
    double x_pos;
    double y_pos;
    long long last_updated;
} ClientState;
typedef struct {
    char* key;
    float value;
} Theme;
```

```
typedef struct {
    pthread_mutex_t mtx;
    ClientState* clients;
    size_t client_count;
    size_t client_capacity;
    Theme* themes;
    size_t theme_count;
    size_t theme_capacity;
    char* current_event_name;
    long long next_client_id;
} SharedWorldState;

// --- Stack Machine Core ---
typedef enum { TYPE_LONGLONG, TYPE_DOUBLE, TYPE_STRING, TYPE_POINTER }
ValueType;
typedef struct {
    ValueType type;
    union {
        long long ll;
        double d;
        char *s;
        void *p;
    } as;
} Value;

typedef struct {
    Value stack[256];
    int top;
} OperandStack;

typedef struct {
    OperandStack stack;
    SharedWorldState* world_state;
    struct mg_connection* http_connection;
} VmContext;

typedef enum {
    OP_REGISTER_CLIENT,
    OP_UPDATE_CLIENT,
    OP_SET_THEME,
    OP_GET_WORLDSTATE_JSON,
    OP_PRUNE_INACTIVE_CLIENTS,
    OP_REPLY_HTTP,
} OpCode;
```

```
void push(OperandStack *s, Value v) { s->stack[s->top++] = v; }
Value pop(OperandStack *s) { return s->stack[--s->top]; }

long long get_timestamp_ms() {
    struct timespec spec;
    clock_gettime(CLOCK_REALTIME, &spec);
    return (long long)spec.tv_sec * 1000 + (long long)spec.tv_nsec / 1.0e6;
}

// --- Core Logic (now called by opcode handlers) ---
void op_register_client(VmContext *ctx) {
    SharedWorldState* state = ctx->world_state;
    pthread_mutex_lock(&state->mtx);
    if (state->client_count >= state->client_capacity) {
        state->client_capacity *= 2;
        state->clients = realloc(state->clients, state->client_capacity * sizeof(ClientState));
    }
    char new_id_buf[64];
    sprintf(new_id_buf, "community_%lld", state->next_client_id++);

    ClientState* new_client = &state->clients[state->client_count++];
    new_client->id = safe_strdup(new_id_buf);
    new_client->x_pos = 0.0;
    new_client->y_pos = 0.0;
    new_client->last_updated = get_timestamp_ms();
    pthread_mutex_unlock(&state->mtx);

    printf("[Server] Registered new client: %s\n", new_id_buf);
    push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup(new_id_buf)});
}

void op_update_client(VmContext *ctx) {
    Value json_val = pop(&ctx->stack);
    cJSON *json = (cJSON*)json_val.as.p;
    cJSON *id = cJSON_GetObjectItem(json, "id");
    cJSON *x = cJSON_GetObjectItem(json, "x");
    cJSON *y = cJSON_GetObjectItem(json, "y");

    if (cJSON_IsString(id) && cJSON_IsNumber(x) && cJSON_IsNumber(y)) {
        pthread_mutex_lock(&ctx->world_state->mtx);
        for (size_t i = 0; i < ctx->world_state->client_count; i++) {
            if (strcmp(ctx->world_state->clients[i].id, id->valuestring) == 0) {
                ctx->world_state->clients[i].x_pos = x->valuedouble;
```

```
            ctx->world_state->clients[i].y_pos = y->valuedouble;
            ctx->world_state->clients[i].last_updated = get_timestamp_ms();
            break;
        }
      }
      pthread_mutex_unlock(&ctx->world_state->mtx);
   }
   cJSON_Delete(json);
}

void op_get_worldstate_json(VmContext *ctx) {
   SharedWorldState *state = ctx->world_state;
   pthread_mutex_lock(&state->mtx);
   cJSON *root = cJSON_CreateObject();
   cJSON *themes = cJSON_CreateObject();
   cJSON *clients = cJSON_CreateObject();
   for (size_t i = 0; i < state->theme_count; i++) cJSON_AddNumberToObject(themes,
state->themes[i].key, state->themes[i].value);
   if (state->current_event_name) cJSON_AddStringToObject(themes, "current_event_name",
state->current_event_name);
   for (size_t i = 0; i < state->client_count; i++) {
      cJSON *client_obj = cJSON_CreateObject();
      cJSON_AddNumberToObject(client_obj, "x", state->clients[i].x_pos);
      cJSON_AddNumberToObject(client_obj, "y", state->clients[i].y_pos);
      cJSON_AddItemToObject(clients, state->clients[i].id, client_obj);
   }
   cJSON_AddItemToObject(root, "themes", themes);
   cJSON_AddItemToObject(root, "clients", clients);
   char *json_str = cJSON_PrintUnformatted(root);
   cJSON_Delete(root);
   pthread_mutex_unlock(&state->mtx);
   push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s = json_str});
}

void op_prune_inactive_clients(VmContext* ctx) {
   SharedWorldState* state = ctx->world_state;
   pthread_mutex_lock(&state->mtx);
   long long now = get_timestamp_ms();
   size_t i = 0;
   while (i < state->client_count) {
      if (now - state->clients[i].last_updated > 10000) {
         printf("[Server] Pruning inactive client: %s\n", state->clients[i].id);
         free(state->clients[i].id);
         if (i < state->client_count - 1) {
```

```
            memmove(&state->clients[i], &state->clients[i+1], (state->client_count - 1 - i) *
sizeof(ClientState));
        }
        state->client_count--;
    } else {
        i++;
    }
  }
  pthread_mutex_unlock(&state->mtx);
}

void execute(VmContext *ctx, OpCode op) {
  switch(op) {
    case OP_REGISTER_CLIENT: op_register_client(ctx); break;
    case OP_UPDATE_CLIENT: op_update_client(ctx); break;
    case OP_GET_WORLDSTATE_JSON: op_get_worldstate_json(ctx); break;
    case OP_PRUNE_INACTIVE_CLIENTS: op_prune_inactive_clients(ctx); break;
    case OP_REPLY_HTTP: {
        Value body_val = pop(&ctx->stack);
        Value code_val = pop(&ctx->stack);
        mg_http_reply(ctx->http_connection, code_val.as.ll, "Content-Type: application/json\r\n",
"%s", body_val.as.s);
        if(body_val.type == TYPE_STRING) free(body_val.as.s);
        break;
    }
    // NOTE: SET_THEME logic omitted for brevity in this refactoring example
    case OP_SET_THEME: { Value v = pop(&ctx->stack); cJSON_Delete(v.as.p); break; }
  }
}

// --- Server Threads and Logic ---
void* pruner_thread_func(void* arg) {
  VmContext* ctx = (VmContext*)arg;
  while (1) {
    sleep(5);
    execute(ctx, OP_PRUNE_INACTIVE_CLIENTS);
  }
  return NULL;
}

static void server_event_handler(struct mg_connection *c, int ev, void *ev_data, void *fn_data) {
  if (ev != MG_EV_HTTP_MSG) return;

  struct mg_http_message *hm = (struct mg_http_message *)ev_data;
```

```
    VmContext ctx = { .top = 0, .world_state = (SharedWorldState*)fn_data, .http_connection = c
};

    if (mg_http_match_uri(hm, "/register")) {
        execute(&ctx, OP_REGISTER_CLIENT); // Pushes new ID string onto stack
        Value id_val = pop(&ctx.stack);
        char buffer[128];
        snprintf(buffer, sizeof(buffer), "{\"id\":\"%s\"}", id_val.as.s);
        free(id_val.as.s);
        push(&ctx.stack, (Value){.type = TYPE_LONGLONG, .as.ll = 200});
        push(&ctx.stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup(buffer)});
        execute(&ctx, OP_REPLY_HTTP);
    } else if (mg_http_match_uri(hm, "/update")) {
        cJSON *json = cJSON_ParseWithLength(hm->body.ptr, hm->body.len);
        if (json) {
            push(&ctx.stack, (Value){.type = TYPE_POINTER, .as.p = json});
            execute(&ctx, OP_UPDATE_CLIENT);
            push(&ctx.stack, (Value){.type = TYPE_LONGLONG, .as.ll = 200});
            push(&ctx.stack, (Value){.type = TYPE_STRING, .as.s =
safe_strdup("{\"status\":\"ok\"}")});
        } else {
            push(&ctx.stack, (Value){.type = TYPE_LONGLONG, .as.ll = 400});
            push(&ctx.stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup("{\"error\":\"Invalid
JSON\"}")});
        }
        execute(&ctx, OP_REPLY_HTTP);
    } else if (mg_http_match_uri(hm, "/worldstate")) {
        execute(&ctx, OP_GET_WORLDSTATE_JSON); // Pushes JSON string
        Value json_val = pop(&ctx.stack);
        push(&ctx.stack, (Value){.type = TYPE_LONGLONG, .as.ll = 200});
        push(&ctx.stack, json_val);
        execute(&ctx, OP_REPLY_HTTP);
    } else {
        push(&ctx.stack, (Value){.type = TYPE_LONGLONG, .as.ll = 404});
        push(&ctx.stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup("Not Found")});
        execute(&ctx, OP_REPLY_HTTP);
    }
}

int main(void) {
    SharedWorldState world_state = { .next_client_id = 1 };
    world_state.client_capacity = 10;
    world_state.clients = safe_malloc(world_state.client_capacity * sizeof(ClientState));
    pthread_mutex_init(&world_state.mtx, NULL);
```

```
    VmContext pruner_ctx = { .top = 0, .world_state = &world_state };
    pthread_t pruner_thread;
    pthread_create(&pruner_thread, NULL, pruner_thread_func, &pruner_ctx);
    pthread_detach(pruner_thread);

    struct mg_mgr mgr;
    mg_mgr_init(&mgr);
    printf("[Server] Starting stack-machine server on port 8080...\n");
    mg_http_listen(&mgr, "8080", server_event_handler, &world_state);
    for (;;) mg_mgr_poll(&mgr, 1000);

    mg_mgr_free(&mgr);
    return 0;
}
EOF

# SERVER MAKEFILE
cat > $ROOT/runtime_server/Makefile <<'EOF'
CC = gcc
CFLAGS = -std=c99 -O2 -Wall -I../vendor
LDFLAGS = -lpthread

SRCS = runtime_server.c ../vendor/mongoose.c ../vendor/cJSON.c
TARGET = runtime_server

all: $(TARGET)
$(TARGET): $(SRCS)
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)
clean:
    rm -f $(TARGET)
EOF

# ------------------------------
# 3) COMPOSITOR CLIENT: Refactored in C with a stack-based architecture
# ------------------------------
cat > $ROOT/compositor/compositor_client.c <<'EOF'
/* compositor_client.c (Stack Machine Version)
 * A multi-threaded C client using an operand stack for its main logic flow.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```
#include <pthread.h>
#include <unistd.h>
#include <math.h>
#include <curl/curl.h>
#include "cJSON.h"

// --- Safe Memory Allocation ---
void* safe_malloc(size_t size) {
    void* ptr = malloc(size);
    if (ptr == NULL) exit(EXIT_FAILURE);
    return ptr;
}
char* safe_strdup(const char* s) {
    char* new_s = strdup(s);
    if (new_s == NULL) exit(EXIT_FAILURE);
    return new_s;
}

// --- Data Structures ---
typedef struct { char* id; double x, y; } ClientState;
typedef struct {
    pthread_mutex_t mtx;
    ClientState* clients;
    size_t client_count;
    size_t client_capacity;
    float traffic_level;
    char* active_event_name;
} WorldState;

// --- Stack Machine Core ---
typedef enum { TYPE_LONGLONG, TYPE_DOUBLE, TYPE_STRING, TYPE_POINTER }
ValueType;
typedef struct {
    ValueType type;
    union { long long ll; double d; char *s; void *p; } as;
} Value;
typedef struct { Value stack[256]; int top; } OperandStack;
typedef struct {
    OperandStack stack;
    ClientState* my_state;
    WorldState* world_state;
} VmContext;
typedef enum { OP_HTTP_REQUEST, OP_UPDATE_WORLD_FROM_JSON,
OP_UPDATE_MY_STATE, OP_WRITE_KML } OpCode;
```

```
void push(OperandStack *s, Value v) { s->stack[s->top++] = v; }
Value pop(OperandStack *s) { return s->stack[--s->top]; }

// --- cURL Helper ---
typedef struct { char *memory; size_t size; } MemoryStruct;
static size_t write_memory_callback(void *contents, size_t size, size_t nmemb, void *userp) {
    size_t realsize = size * nmemb;
    MemoryStruct *mem = (MemoryStruct *)userp;
    char *ptr = realloc(mem->memory, mem->size + realsize + 1);
    if(ptr == NULL) return 0;
    mem->memory = ptr;
    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0;
    return realsize;
}

// --- Opcode Handlers ---
void op_http_request(VmContext *ctx) {
    Value payload_val = pop(&ctx->stack);
    Value url_val = pop(&ctx->stack);
    CURL *curl = curl_easy_init();
    MemoryStruct chunk = { .memory = safe_malloc(1), .size = 0 };
    curl_easy_setopt(curl, CURLOPT_URL, url_val.as.s);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_memory_callback);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *)&chunk);
    if (payload_val.as.s) {
        struct curl_slist *headers = curl_slist_append(NULL, "Content-Type: application/json");
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, payload_val.as.s);
        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
    }
    CURLcode res = curl_easy_perform(curl);
    curl_easy_cleanup(curl);
    free(url_val.as.s);
    free(payload_val.as.s);
    if (res == CURLE_OK) {
        push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s = chunk.memory});
    } else {
        free(chunk.memory);
        push(&ctx->stack, (Value){.type = TYPE_POINTER, .as.p = NULL});
    }
}
```

```
void op_update_world_from_json(VmContext *ctx) {
    Value json_val = pop(&ctx->stack);
    if (!json_val.as.s) return;

    WorldState* world_state = ctx->world_state;
    pthread_mutex_lock(&world_state->mtx);
    cJSON *root = cJSON_Parse(json_val.as.s);
    if (root) {
        cJSON *clients_obj = cJSON_GetObjectItem(root, "clients");
        for (size_t i = 0; i < world_state->client_count; i++) free(world_state->clients[i].id);
        world_state->client_count = 0;
        cJSON *client_item;
        cJSON_ArrayForEach(client_item, clients_obj) {
            if (world_state->client_count >= world_state->client_capacity) {
                world_state->client_capacity *= 2;
                world_state->clients = realloc(world_state->clients, world_state->client_capacity *
sizeof(ClientState));
            }
            ClientState* c = &world_state->clients[world_state->client_count++];
            c->id = safe_strdup(client_item->string);
            c->x = cJSON_GetObjectItem(client_item, "x")->valuedouble;
            c->y = cJSON_GetObjectItem(client_item, "y")->valuedouble;
        }
        cJSON_Delete(root);
    }
    pthread_mutex_unlock(&world_state->mtx);
    free(json_val.as.s);
}

void op_update_my_state(VmContext* ctx) {
    Value frame_val = pop(&ctx->stack);
    ctx->my_state->x = (sin(frame_val.as.ll * 0.1) * 20.0);
    ctx->my_state->y = (cos(frame_val.as.ll * 0.07) * 20.0);
}

void execute(VmContext *ctx, OpCode op) {
    switch(op) {
        case OP_HTTP_REQUEST: op_http_request(ctx); break;
        case OP_UPDATE_WORLD_FROM_JSON: op_update_world_from_json(ctx); break;
        case OP_UPDATE_MY_STATE: op_update_my_state(ctx); break;
        // KML generation is complex and left as a direct call for this example
        case OP_WRITE_KML: break;
    }
}
```

```
// --- Threads ---
void* update_thread_func(void* arg) {
    VmContext* ctx = (VmContext*)arg;
    long long frame = 0;
    char url_buf[256], payload_buf[256];
    while(1) {
        push(&ctx->stack, (Value){.type = TYPE_LONGLONG, .as.ll = frame});
        execute(ctx, OP_UPDATE_MY_STATE);

        snprintf(url_buf, sizeof(url_buf), "http://localhost:8080/update");
        snprintf(payload_buf, sizeof(payload_buf), "{\"id\":\"%s\",\"x\":%f,\"y\":%f}",
ctx->my_state->id, ctx->my_state->x, ctx->my_state->y);
        push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup(url_buf)});
        push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup(payload_buf)});
        execute(ctx, OP_HTTP_REQUEST);
        Value resp = pop(&ctx->stack); // Pop and free response
        if(resp.as.s) free(resp.as.s);

        usleep(100 * 1000);
        frame++;
    }
    return NULL;
}

void* sync_thread_func(void* arg) {
    VmContext* ctx = (VmContext*)arg;
    while(1) {
        push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s =
safe_strdup("http://localhost:8080/worldstate")});
        push(&ctx->stack, (Value){.type = TYPE_STRING, .as.s = NULL}); // No payload for GET
        execute(ctx, OP_HTTP_REQUEST); // Pushes JSON response string
        execute(ctx, OP_UPDATE_WORLD_FROM_JSON); // Pops JSON string and updates
state
        usleep(200 * 1000);
    }
    return NULL;
}

int main() {
    curl_global_init(CURL_GLOBAL_ALL);
    VmContext main_ctx = { .top = 0 };

    // Register with server
```

```
    push(&main_ctx.stack, (Value){.type = TYPE_STRING, .as.s =
safe_strdup("http://localhost:8080/register")});
    push(&main_ctx.stack, (Value){.type = TYPE_STRING, .as.s = safe_strdup("{}")});
    execute(&main_ctx, OP_HTTP_REQUEST);
    Value reg_resp = pop(&main_ctx.stack);
    if (!reg_resp.as.s) {
        fprintf(stderr, "Error: Could not register with server.\n");
        return 1;
    }

    ClientState my_state = {0};
    cJSON* json_reg = cJSON_Parse(reg_resp.as.s);
    my_state.id = safe_strdup(cJSON_GetObjectItem(json_reg, "id")->valuestring);
    cJSON_Delete(json_reg);
    free(reg_resp.as.s);
    printf("[Client] Registered with server. My ID is: %s\n", my_state.id);

    WorldState world_state = {0};
    pthread_mutex_init(&world_state.mtx, NULL);
    world_state.client_capacity = 10;
    world_state.clients = safe_malloc(world_state.client_capacity * sizeof(ClientState));

    VmContext update_ctx = { .top = 0, .my_state = &my_state };
    VmContext sync_ctx = { .top = 0, .world_state = &world_state };

    pthread_t updater, syncer;
    pthread_create(&updater, NULL, update_thread_func, &update_ctx);
    pthread_create(&syncer, NULL, sync_thread_func, &sync_ctx);
    pthread_detach(updater);
    pthread_detach(syncer);

    printf("Starting KML generation loop...\n");
    int frame = 0;
    while(1) {
        pthread_mutex_lock(&world_state.mtx);
        printf("\rFrame %d: Visualizing %zu clients.", frame, world_state.client_count);
        fflush(stdout);
        // KML generation is complex and left as a direct call for brevity
        FILE* f = fopen("geovis.kml", "w");
        if (f) {
            fprintf(f, "<?xml version=\"1.0\"?><kml><Document><name>Frame %d</name>",
frame);
            for(size_t i=0; i < world_state.client_count; i++) {
```

```
            fprintf(f,
"<Placemark><name>%s</name><Point><coordinates>%.4f,%.4f,0</coordinates></Point></Pl
acemark>", world_state.clients[i].id, -79.38 + world_state.clients[i].x * 0.01, 43.65 +
world_state.clients[i].y * 0.01);
        }
        fprintf(f, "</Document></kml>");
        fclose(f);
    }
    pthread_mutex_unlock(&world_state.mtx);
    usleep(100 * 1000);
    frame++;
  }

  return 0;
}
EOF

# CLIENT MAKEFILE
cat > $ROOT/compositor/Makefile <<'EOF'
CC = gcc
CFLAGS = -std=c99 -O2 -Wall -I../vendor
LDFLAGS = -lcurl -lpthread -lm

SRCS = compositor_client.c ../vendor/cJSON.c
TARGET = compositor_client

all: $(TARGET)
$(TARGET): $(SRCS)
  $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)
clean:
  rm -f $(TARGET)
EOF


# ----------------------------
# 4) README.md (Updated for C stack-machine version)
# ----------------------------
cat > $ROOT/README.md <<'EOF'
# Multi-System Compositor Simulation (C Stack-Machine Version)

This package contains a C implementation of a multi-client simulation refactored to use a
**stack-based architecture**. Instead of traditional function calls with parameters and return
values, the programs operate like virtual machines.
```

Architectural Changes

- **Operand Stack**: Functions communicate by pushing and popping values (operands) onto a shared stack.
- **Opcodes**: Logic is invoked by dispatching operand codes (opcodes), such as `OP_HTTP_REQUEST` or `OP_REGISTER_CLIENT`, which replace direct function calls.
- **Implicit Data Flow**: This design makes the data flow between components implicit. The code describes a sequence of operations to be performed on the stack rather than a direct chain of calls.

This is a demonstration of an alternative programming paradigm. For an application of this scale, it introduces significant abstraction and complexity compared to the direct-call version.

How to Build and Run

The build and run process is identical to the previous C version.

1. Prerequisites
You will need `gcc`, `make`, and `libcurl-dev`.

2. Build the Components
```bash
cd compositor-suite-c-stack
# Build the server
cd runtime_server && make && cd ..
# Build the client
cd compositor && make && cd ..
```

#ARM11.sh Implementation

```
#!/bin/bash
set -e
ROOT=neuromorphic-suite-shortcode
rm -rf $ROOT
mkdir -p $ROOT/{runtime_server,bootloader,arm_helper,kernel,vhdl,include,systemd}

echo "Creating project tree in ./$ROOT ..."

# ------------------------------
# 1) include/neuromorphic_short_codes.h (NEW)
# ------------------------------
cat > $ROOT/include/neuromorphic_short_codes.h <<'EOF'
```

```
#ifndef NEUROMORPHIC_SHORT_CODES_H
#define NEUROMORPHIC_SHORT_CODES_H

#include <cstdint>
#include <vector>

// Bytecode opcodes
enum class OpCode : uint8_t {
    // Stack manipulation
    PUSH_CONST_DBL = 0x01,
    PUSH_CONST_U64 = 0x02,
    PUSH_REG       = 0x03,
    POP_REG        = 0x04,
    // Control flow
    JUMP_IF_ZERO   = 0x10,
    JUMP           = 0x11,
    CALL           = 0x12,
    RET            = 0x13,
    // Arithmetic/Logic
    ADD_DBL        = 0x20,
    MUL_DBL        = 0x21,
    SUB_DBL        = 0x22,
    DIV_DBL        = 0x23,
    CMP_GT_DBL     = 0x28, // Pushes 1.0 if top-1 > top, else 0.0
    // System calls
    SYS_SET_LANES  = 0xA0,
    SYS_GET_STATUS = 0xA1,
    SYS_TRANSFORM  = 0xA2,
    SYS_SAVE_STATE = 0xA3,
    // Bootloader specific
    SYS_MMIO_READ  = 0xB0,
    SYS_MMIO_WRITE = 0xB1,
    SYS_LOG_EVENT  = 0xB2,
    SYS_RESET      = 0xB3,

    HALT           = 0xFF,
};

// Represents a value on the VM stack
union VMValue {
    double   f64;
    uint64_t u64;
};
```

```
// VM execution context
struct VMContext {
    std::vector<uint8_t> bytecode;
    size_t ip = 0; // instruction pointer
    std::vector<VMValue> stack;
    std::vector<VMValue> registers; // general purpose regs
    bool halted = false;

    VMContext(size_t stack_size = 64, size_t reg_count = 8) {
        stack.reserve(stack_size);
        registers.resize(reg_count);
    }
};

#endif // NEUROMORPHIC_SHORT_CODES_H
EOF

# ------------------------------
# 2) runtime_server/gpu_runtime_server.cpp (MODIFIED)
# ------------------------------
cat > $ROOT/runtime_server/gpu_runtime_server.cpp <<'EOF'
/* gpu_runtime_server_arm.cpp
 * + Short-code bytecode interpreter for /execute endpoint
 * + ... (other features retained) ...
 */

#include <iostream>
#include <sstream>
// ... (includes from original file) ...
#include <cstring>
#include "httplib.h"
#include "simdjson.h"
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include "../include/neuromorphic_short_codes.h"

// ... (All original code from gpu_runtime_server.cpp, up to main()) ...
// (Example: GlobalGpuThrottler, ThrottleAutoScaler, GPU transforms, etc. are retained
verbatim)

class VMExecutor {
public:
    VMExecutor(ServerState& state, GlobalGpuThrottler& throttler, ThrottleAutoScaler& scaler)
        : g_state(state), g_throttler(throttler), g_autoscaler(scaler) {}
```

```
void execute_bytecode(VMContext& ctx) {
    while (!ctx.halted && ctx.ip < ctx.bytecode.size()) {
        OpCode op = static_cast<OpCode>(ctx.bytecode[ctx.ip++]);
        switch (op) {
            case OpCode::PUSH_CONST_DBL: {
                double val;
                std::memcpy(&val, &ctx.bytecode[ctx.ip], sizeof(double));
                ctx.ip += sizeof(double);
                ctx.stack.push_back({.f64 = val});
                break;
            }
            case OpCode::POP_REG: {
                uint8_t reg_idx = ctx.bytecode[ctx.ip++];
                if (reg_idx < ctx.registers.size()) {
                    ctx.registers[reg_idx] = ctx.stack.back();
                    ctx.stack.pop_back();
                }
                break;
            }
            case OpCode::ADD_DBL: {
                double b = ctx.stack.back().f64; ctx.stack.pop_back();
                double a = ctx.stack.back().f64; ctx.stack.pop_back();
                ctx.stack.push_back({.f64 = a + b});
                break;
            }
            case OpCode::SYS_SET_LANES: {
                double fraction = ctx.stack.back().f64; ctx.stack.pop_back();
                g_autoscaler.set_lane_fraction(fraction);
                g_state.lane_fraction = fraction;
                break;
            }
            case OpCode::SYS_GET_STATUS: {
                // Pushes: throttle_rate (u64), measured_rate (f64), lane_fraction (f64)
                ctx.stack.push_back({.u64 = g_throttler.get_rate()});
                ctx.stack.push_back({.f64 = g_autoscaler.measured_rate()});
                ctx.stack.push_back({.f64 = g_autoscaler.get_lane_fraction()});
                break;
            }
            case OpCode::HALT: {
                ctx.halted = true;
                break;
            }
            default:
```

```
                ctx.halted = true; // Unknown opcode
                break;
            }
        }
    }
private:
    ServerState& g_state;
    GlobalGpuThrottler& g_throttler;
    ThrottleAutoScaler& g_autoscaler;
};

/* ---------------------------------------------------------------------- */
/* HTTP server endpoints                                                  */
/* ---------------------------------------------------------------------- */
int main(int argc, char **argv) {
    // ... (All original setup from main()) ...

    VMExecutor vm(g_state, g_throttler, g_autoscaler);

    // POST /execute (NEW ENDPOINT)
    svr.Post("/execute", [&](const httplib::Request &req, httplib::Response &res) {
        try {
            VMContext ctx;
            ctx.bytecode.assign(req.body.begin(), req.body.end());
            vm.execute_bytecode(ctx);

            // Return stack state as JSON
            std::ostringstream oss;
            oss << "{\"stack\":[";
            for (size_t i = 0; i < ctx.stack.size(); ++i) {
                if (i) oss << ",";
                // NOTE: A more robust impl would know the type. Assuming double for now.
                oss << ctx.stack[i].f64;
            }
            oss << "], \"halted\":" << (ctx.halted ? "true" : "false") << "}";
            res.set_content(oss.str(), "application/json");
        } catch (const std::exception &e) {
            res.status = 500;
            res.set_content(std::string("{\"error\":\"") + e.what() + "\"}", "application/json");
        }
    });

    // ... (All other original endpoints from main()) ...
```

```
    std::cerr << "[server] listening on 0.0.0.0:" << port << " with short-code support\n";
    svr.listen("0.0.0.0", port);

    // ... (All original shutdown code from main()) ...
}
EOF

# ----------------------------
# 3) bootloader/bootloader_ttd_refit.c (MODIFIED)
# ----------------------------
cat > $ROOT/bootloader/bootloader_ttd_refit.c <<'EOF'
// bootloader_ttd_branching.c
// + Short-code bytecode interpreter for intrant control.

#define _POSIX_C_SOURCE 200112L
#include <stdio.h>
#include <stdint.h>
// ... (all original includes) ...
#include <errno.h>

// NOTE: C doesn't have a direct vector equivalent, so we use a simple array for the VM.
// The C++ header is included conceptually.
// #include "../include/neuromorphic_short_codes.h"
typedef enum {
    OP_PUSH_CONST_U64 = 0x02,
    OP_PUSH_REG       = 0x03,
    OP_POP_REG        = 0x04,
    OP_SYS_MMIO_READ  = 0xB0,
    OP_SYS_MMIO_WRITE = 0xB1,
    OP_SYS_LOG_EVENT  = 0xB2,
    OP_SYS_RESET      = 0xB3,
    OP_HALT           = 0xFF,
} OpCode;
typedef union { uint64_t u64; } VMValue;
typedef struct {
    const uint8_t *bytecode;
    size_t ip;
    size_t len;
    VMValue stack[64];
    int top;
    VMValue registers[8];
    bool halted;
} VMContextC;
```

```
// ... (All original code from bootloader, up to command handling) ...

static void vm_execute_bytecode(VMContextC* ctx) {
    while (!ctx->halted && ctx->ip < ctx->len) {
        OpCode op = (OpCode)ctx->bytecode[ctx->ip++];
        switch (op) {
            case OP_PUSH_CONST_U64: {
                uint64_t val;
                memcpy(&val, &ctx->bytecode[ctx->ip], sizeof(uint64_t));
                ctx->ip += sizeof(uint64_t);
                ctx->stack[ctx->top++] = (VMValue){.u64 = val};
                break;
            }
            case OP_POP_REG: {
                uint8_t reg_idx = ctx->bytecode[ctx->ip++];
                if (reg_idx < 8) ctx->registers[reg_idx] = ctx->stack[--ctx->top];
                break;
            }
            case OP_SYS_MMIO_WRITE: {
                uint64_t val = ctx->stack[--ctx->top].u64;
                uint64_t addr = ctx->stack[--ctx->top].u64;
                mmio_write(addr, val);
                break;
            }
            case OP_SYS_RESET: {
                uint64_t type = ctx->stack[--ctx->top].u64;
                if (type == 0) assert_sump_reset();
                else if (type == 1) assert_sgi_reset();
                break;
            }
            case OP_HALT:
                ctx->halted = true;
                break;
            default:
                ctx->halted = true;
                break;
        }
    }
}

/* New "execute" command */
static inline void cmd_execute(const char *args) {
    // Expects hex string of bytecode
```

```
    size_t hex_len = strlen(args);
    if (hex_len == 0 || hex_len % 2 != 0) {
        debug_send_line("Invalid hex bytecode string\n");
        return;
    }
    size_t byte_len = hex_len / 2;
    uint8_t* bytecode = malloc(byte_len);
    for (size_t i = 0; i < byte_len; ++i) {
        sscanf(args + 2*i, "%2hhx", &bytecode[i]);
    }

    VMContextC ctx = {
        .bytecode = bytecode, .len = byte_len, .ip = 0, .top = 0, .halted = false
    };
    vm_execute_bytecode(&ctx);

    free(bytecode);
    debug_send_line("Execution halted.\n");
}


/* Dispatcher (modified) */
static const CmdMap cmd_map[] = {
    // ... (original commands) ...
    { "execute",            CMD_EXECUTE }, // ADD THIS
    { "exit",               CMD_EXIT },
};

static inline void dispatch_command(const char *line, CmdCtx *ctx) {
    // ...
    switch (parse_command(line, &args)) {
        // ... (all original cases) ...
        case CMD_EXECUTE:           cmd_execute(args); break;
        default:                    debug_send_line("Unknown command\n"); break;
    }
}
// ... (rest of the original bootloader file, including main) ...
EOF

# ----------------------------
# 4) systemd and helper script (MODIFIED to use short-codes)
# ----------------------------
cat > $ROOT/systemd/lane-reporter.service <<'EOF'
[Unit]
```

```
Description=Neuromorphic Lane Reporter (Short-Code)
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/lane_reporter.sh
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
EOF

cat > $ROOT/systemd/lane_reporter.sh <<'EOF'
#!/bin/bash
# This script now sends short-code bytecode to the /execute endpoint
# to achieve the same result as the original /control/lanes POST.

# OpCodes:
# PUSH_CONST_DBL = 0x01
# SYS_SET_LANES  = 0xA0
# HALT           = 0xFF

MMIO_BASE=0x40000000
SERVER_URL="http://localhost:8080/execute"

while true;
do
  # 1. Read lane utilization locally (same as before)
  LANES_JSON=$(/usr/local/bin/send_lane_fractions_batch $MMIO_BASE --stdout-only)
  if [ -z "$LANES_JSON" ]; then
    sleep 0.2
    continue
  fi

  # 2. Calculate the sum to get the total fraction
  SUM_FRAC=$(echo "$LANES_JSON" | jq '[.[]] | add')

  # 3. Convert double to little-endian hex bytes
  HEX_BYTES=$(python3 -c "import struct; print(struct.pack('<d', $SUM_FRAC).hex())")

  # 4. Construct bytecode payload
  #    [PUSH_CONST_DBL] [8-byte double] [SYS_SET_LANES] [HALT]
  BYTECODE="01${HEX_BYTES}a0ff"
```

```
  # 5. Send bytecode to the server
  curl -s -X POST --data-binary @- "$SERVER_URL" > /dev/null <<EOF
$(printf "\\x${BYTECODE//../\\x}")
EOF

  sleep 0.2
done
EOF
chmod +x $ROOT/systemd/lane_reporter.sh

# ----------------------------
# 5) arm_helper/send_lane_fractions_batch.c (MODIFIED)
# ----------------------------
cat > $ROOT/arm_helper/send_lane_fractions_batch.c <<'EOF'
/* send_lane_fractions_batch.c
 * Now with --stdout-only flag to just print JSON.
 * ... (original comments) ...
 */

#include <stdio.h>
// ... (original includes) ...

// ... (original batch_read_lanes and post_lane_json functions) ...

int main(int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <mmio_base_hex> [--stdout-only]\n", argv[0]);
        return 1;
    }

    int stdout_only = (argc > 2 && strcmp(argv[2], "--stdout-only") == 0);

    // ... (original lane reading logic) ...

    char json_buf[256];
    // ... (original JSON building logic) ...

    if (stdout_only) {
        printf("%s\n", json_buf);
    } else {
        printf("Posting lane fractions: %s\n", json_buf);
        if (post_lane_json("http://localhost:8080/control/lanes", json_buf) != 0) {
            fprintf(stderr, "POST failed.\n");
```

```
            return 3;
        }
        printf("Posted successfully.\n");
    }

    return 0;
}
EOF


# ----------------------------
# 6) README (MODIFIED)
# ----------------------------
cat > $ROOT/README.md <<'EOF'
# Neuromorphic Suite (Short-Code Edition)

This package contains a refit of the neuromorphic suite to use a compact, intrant short-code (bytecode) system for control.

## Key Changes
- **Short-Code VM**: Both the runtime server and the bootloader now include a simple, stack-based virtual machine.
- **/execute Endpoint**: The runtime server exposes a new `/execute` endpoint that accepts binary bytecode payloads for execution.
- **Bootloader `execute` Command**: The bootloader's debug shell has a new `execute <hex_string>` command to run bytecode on the bare-metal target.
- **lane-reporter.sh**: The systemd helper script has been updated. Instead of POSTing JSON to `/control/lanes`, it now reads MMIO, calculates the desired lane fraction, and sends a bytecode program to the `/execute` endpoint to set the value.

## Build and Run

Build steps are the same as the original.

1.  **Build components**: Use the `make` command in `runtime_server/`, `bootloader/`, and `arm_helper/`.
2.  **Run Server**: `./runtime_server/gpu_runtime_server`
3.  **Run Bootloader**: `./bootloader/bootloader_ttd_refit 2222`
4.  **Install & Run Reporter**: Install `send_lane_fractions_batch` and `lane_reporter.sh` to `/usr/local/bin` and enable the systemd service. **Requires `jq` and `python3`**.

### Example: Manual Short-Code Execution
```

You can control the server manually with bytecode. This example pushes the value `0.5`, sets the lane fraction, and halts.

Bytecode: `01000000000000e03fa0ff`
- `01`: `PUSH_CONST_DBL`
- `000000000000e03f`: Little-endian representation of `0.5`
- `a0`: `SYS_SET_LANES`
- `ff`: `HALT`

Command:
```sh
echo -n -e "\\x01\\x00\\x00\\x00\\x00\\x00\\x00\\xe0\\x3f\\xa0\\xff" | curl -X POST --data-binary @- http://localhost:8080/execute
```