//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

extern "C" PVOID \_ReturnAddress(VOID);

#pragma intrinsic(\_ReturnAddress)

class BailOutRecord;

extern "C" void \_\_cdecl \_alloca\_probe\_16();

namespace Js

{

class EHBailoutData;

enum InterpreterStackFrameFlags : UINT16

{

InterpreterStackFrameFlags\_None = 0,

InterpreterStackFrameFlags\_WithinTryBlock = 1,

InterpreterStackFrameFlags\_WithinCatchBlock = 2,

InterpreterStackFrameFlags\_WithinFinallyBlock = 4,

InterpreterStackFrameFlags\_FromBailOut = 8,

InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode = 0x10,

InterpreterStackFrameFlags\_All = 0xFFFF,

};

struct InterpreterStackFrame /\* Stack allocated, no virtuals \*/

{

PREVENT\_COPY(InterpreterStackFrame)

friend class BailOutRecord;

friend class JavascriptGeneratorFunction;

friend class JavascriptGenerator;

class Setup

{

public:

Setup(ScriptFunction \* function, Arguments& args, bool inlinee = false);

Setup(ScriptFunction \* function, Var \* inParams, int inSlotsCount, bool inlinee = false);

size\_t GetAllocationVarCount() const { return varAllocCount; }

InterpreterStackFrame \* AllocateAndInitialize(bool doProfile, bool \* releaseAlloc);

#if DBG

InterpreterStackFrame \* InitializeAllocation(\_\_in\_ecount(varAllocCount) Var \* allocation, bool initParams, bool profileParams, Var loopHeaderArray, DWORD\_PTR stackAddr, Var invalidStackVar);

#else

InterpreterStackFrame \* InitializeAllocation(\_\_in\_ecount(varAllocCount) Var \* allocation, bool initParams, bool profileParams, Var loopHeaderArray, DWORD\_PTR stackAddr);

#endif

uint GetLocalCount() const { return localCount; }

private:

template <class Fn>

void InitializeParams(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest);

template <class Fn>

void InitializeParamsAndUndef(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest);

void InitializeRestParam(InterpreterStackFrame \* newInstance, Var \*dest);

void SetupInternal();

Var \* inParams;

ScriptFunction \* const function;

FunctionBody \* const executeFunction;

void\*\* inlineCaches;

int inSlotsCount;

uint localCount;

uint varAllocCount;

uint inlineCacheCount;

Js::CallFlags callFlags;

bool bailedOutOfInlinee;

};

private:

ByteCodeReader m\_reader; // Reader for current function

int m\_inSlotsCount; // Count of actual incoming parameters to this function

Js::CallFlags m\_callFlags; // CallFlags passed to the current function

Var\* m\_inParams; // Range of 'in' parameters

Var\* m\_outParams; // Range of 'out' parameters (offset in m\_localSlots)

Var\* m\_outSp; // Stack pointer for next outparam

Var m\_arguments; // Dedicated location for this frame's arguments object

StackScriptFunction \* stackNestedFunctions;

FrameDisplay \* localFrameDisplay;

Var localClosure;

Var \*innerScopeArray;

ScriptContext\* scriptContext;

ScriptFunction \* function;

FunctionBody \* m\_functionBody;

void\*\* inlineCaches;

void \* returnAddress;

void \* addressOfReturnAddress; // Tag this frame with stack position, used by (remote) stack walker to test partially initialized interpreter stack frame.

InterpreterStackFrame \*previousInterpreterFrame;

Var loopHeaderArray; // Keeps alive any JITted loop bodies while the function is being interpreted

// 'stack address' of the frame, used for recursion detection during stepping.

// For frames created via interpreter path, we use 'this', for frames created by bailout we use stack addr of actual jitted frame

// the interpreter frame is created for.

DWORD\_PTR m\_stackAddress;

#if ENABLE\_PROFILE\_INFO

ImplicitCallFlags \* savedLoopImplicitCallFlags;

#endif

uint inlineCacheCount;

uint currentLoopNum;

uint currentLoopCounter; // This keeps tracks of how many times the current loop is executed. It's hit only in cases where jitloopbodies are not hit

// such as loops inside try\catch.

UINT16 m\_flags; // based on InterpreterStackFrameFlags

bool closureInitDone : 1;

#if ENABLE\_PROFILE\_INFO

bool switchProfileMode : 1;

bool isAutoProfiling : 1;

uint32 switchProfileModeOnLoopEndNumber;

#endif

int16 nestedTryDepth;

int16 nestedCatchDepth;

uint retOffset;

int16 nestedFinallyDepth;

void (InterpreterStackFrame::\*opLoopBodyStart)(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

#if ENABLE\_PROFILE\_INFO

void (InterpreterStackFrame::\*opProfiledLoopBodyStart)(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

#endif

#if DBG || DBG\_DUMP

void \* DEBUG\_currentByteOffset;

#endif

// Asm.js stack pointer

int\* m\_localIntSlots;

double\* m\_localDoubleSlots;

float\* m\_localFloatSlots;

\_SIMDValue\* m\_localSimdSlots;

EHBailoutData \* ehBailoutData;

// 16-byte aligned

\_\_declspec(align(16)) Var m\_localSlots[0]; // Range of locals and temporaries

static const int LocalsThreshold = 32 \* 1024; // Number of locals vars we'll allocate on the frame.

// If there are more, we'll use an arena.

typedef void(InterpreterStackFrame::\*ArrFunc)(uint32, RegSlot);

static const ArrFunc StArrFunc[8];

static const ArrFunc LdArrFunc[8];

//This class must have an empty ctor (otherwise it will break the code in InterpreterStackFrame::InterpreterThunk

inline InterpreterStackFrame() { }

void ProcessTryFinally(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException = Js::Constants::NoRegister, Js::RegSlot regOffset = Js::Constants::NoRegister, bool hasYield = false);

public:

void OP\_SetOutAsmDb(RegSlot outRegisterID, double val);

void OP\_SetOutAsmInt(RegSlot outRegisterID, int val);

void OP\_I\_SetOutAsmInt(RegSlot outRegisterID, int val);

void OP\_I\_SetOutAsmDb(RegSlot outRegisterID, double val);

void OP\_I\_SetOutAsmFlt(RegSlot outRegisterID, float val);

void OP\_I\_SetOutAsmSimd(RegSlot outRegisterID, AsmJsSIMDValue val);

void SetOut(ArgSlot outRegisterID, Var bValue);

void SetOut(ArgSlot\_OneByte outRegisterID, Var bValue);

void PushOut(Var aValue);

void PopOut(ArgSlot argCount);

FrameDisplay \* GetLocalFrameDisplay() const;

FrameDisplay \* GetFrameDisplayForNestedFunc() const;

Var InnerScopeFromRegSlot(RegSlot reg) const;

void SetClosureInitDone(bool done) { closureInitDone = done; }

void ValidateRegValue(Var value, bool allowStackVar = false, bool allowStackVarOnDisabledStackNestedFunc = true) const;

void ValidateSetRegValue(Var value, bool allowStackVar = false, bool allowStackVarOnDisabledStackNestedFunc = true) const;

template <typename RegSlotType> Var GetReg(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetReg(RegSlotType localRegisterID, Var bValue);

template <typename RegSlotType> Var GetRegAllowStackVar(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetRegAllowStackVar(RegSlotType localRegisterID, Var bValue);

template <typename RegSlotType> int GetRegRawInt( RegSlotType localRegisterID ) const;

template <typename RegSlotType> void SetRegRawInt( RegSlotType localRegisterID, int bValue );

template <typename RegSlotType> double GetRegRawDouble(RegSlotType localRegisterID) const;

template <typename RegSlotType> float GetRegRawFloat(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetRegRawDouble(RegSlotType localRegisterID, double bValue);

template <typename RegSlotType> void SetRegRawFloat(RegSlotType localRegisterID, float bValue);

template <typename T> T GetRegRaw( RegSlot localRegisterID ) const;

template <typename T> void SetRegRaw( RegSlot localRegisterID, T bValue );

template <typename RegSlotType> AsmJsSIMDValue GetRegRawSimd(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetRegRawSimd(RegSlotType localRegisterID, AsmJsSIMDValue bValue);

static DWORD GetAsmSimdValOffSet(AsmJsCallStackLayout\* stack);

template <class T> void OP\_SimdLdArrGeneric(const unaligned T\* playout);

template <class T> void OP\_SimdLdArrConstIndex(const unaligned T\* playout);

template <class T> void OP\_SimdStArrGeneric(const unaligned T\* playout);

template <class T> void OP\_SimdStArrConstIndex(const unaligned T\* playout);

template <typename RegSlotType>

Var GetRegAllowStackVarEnableOnly(RegSlotType localRegisterID) const;

template <typename RegSlotType>

void SetRegAllowStackVarEnableOnly(RegSlotType localRegisterID, Var bValue);

Var GetNonVarReg(RegSlot localRegisterID) const;

void SetNonVarReg(RegSlot localRegisterID, void \* bValue);

ScriptContext\* GetScriptContext() const { return scriptContext; }

Var GetRootObject() const;

ScriptFunction\* GetJavascriptFunction() const { return function; }

FunctionBody \* GetFunctionBody() const { return m\_functionBody; }

ByteCodeReader\* GetReader() { return &m\_reader;}

uint GetCurrentLoopNum() const { return currentLoopNum; }

InterpreterStackFrame\* GetPreviousFrame() const {return previousInterpreterFrame;}

void SetPreviousFrame(InterpreterStackFrame \*interpreterFrame) {previousInterpreterFrame = interpreterFrame;}

Var GetArgumentsObject() const { return m\_arguments; }

void SetArgumentsObject(Var args) { m\_arguments = args; }

UINT16 GetFlags() const { return m\_flags; }

void OrFlags(UINT16 addTo) { m\_flags |= addTo; }

bool IsInCatchOrFinallyBlock();

static bool IsDelayDynamicInterpreterThunk(void\* entryPoint);

Var LdEnv() const;

void SetEnv(FrameDisplay \*frameDisplay);

Var \* NewScopeSlots(unsigned int size, ScriptContext \*scriptContext, Var scope);

Var \* NewScopeSlots();

Var NewScopeObject();

FrameDisplay \* NewFrameDisplay(void \*argHead, void \*argEnv);

Var CreateHeapArguments(ScriptContext\* scriptContext);

bool IsCurrentLoopNativeAddr(void \* codeAddr) const;

void \* GetReturnAddress() { return returnAddress; }

static uint32 GetOffsetOfLocals() { return offsetof(InterpreterStackFrame, m\_localSlots); }

static uint32 GetOffsetOfArguments() { return offsetof(InterpreterStackFrame, m\_arguments); }

static uint32 GetOffsetOfInParams() { return offsetof(InterpreterStackFrame, m\_inParams); }

static uint32 GetOffsetOfInSlotsCount() { return offsetof(InterpreterStackFrame, m\_inSlotsCount); }

void PrintStack(const int\* const intSrc, const float\* const fltSrc, const double\* const dblSrc, int intConstCount, int floatConstCount, int doubleConstCount, const wchar\_t\* state);

static uint32 GetStartLocationOffset() { return offsetof(InterpreterStackFrame, m\_reader) + ByteCodeReader::GetStartLocationOffset(); }

static uint32 GetCurrentLocationOffset() { return offsetof(InterpreterStackFrame, m\_reader) + ByteCodeReader::GetCurrentLocationOffset(); }

static bool IsBrLong(OpCode op, const byte \* ip)

{

#ifdef BYTECODE\_BRANCH\_ISLAND

return (op == OpCode::ExtendedOpcodePrefix) && ((OpCode)(ByteCodeReader::PeekByteOp(ip) + (OpCode::ExtendedOpcodePrefix << 8)) == OpCode::BrLong);

#else

return false;

#endif

}

DWORD\_PTR GetStackAddress() const;

void\* GetAddressOfReturnAddress() const;

#if \_M\_IX86

static int GetRetType(JavascriptFunction\* func);

static int GetAsmJsArgSize(AsmJsCallStackLayout \* stack);

static int GetDynamicRetType(AsmJsCallStackLayout \* stack);

static DWORD GetAsmIntDbValOffSet(AsmJsCallStackLayout \* stack);

\_\_declspec(noinline) static int AsmJsInterpreter(AsmJsCallStackLayout \* stack);

#elif \_M\_X64

template <typename T>

static T AsmJsInterpreter(AsmJsCallStackLayout\* layout);

static void \* GetAsmJsInterpreterEntryPoint(AsmJsCallStackLayout\* stack);

template <typename T>

static T GetAsmJsRetVal(InterpreterStackFrame\* instance);

static Var AsmJsDelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...);

static \_\_m128 AsmJsInterpreterSimdJs(AsmJsCallStackLayout\* func);

#endif

#ifdef ASMJS\_PLAT

static void InterpreterAsmThunk(AsmJsCallStackLayout\* layout);

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

static Var DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...);

\_\_declspec(noinline) static Var InterpreterThunk(JavascriptCallStackLayout\* layout);

#else

\_\_declspec(noinline) static Var InterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...);

#endif

static Var InterpreterHelper(ScriptFunction\* function, ArgumentReader args, void\* returnAddress, void\* addressOfReturnAddress, const bool isAsmJs = false);

private:

#if DYNAMIC\_INTERPRETER\_THUNK

static JavascriptMethod EnsureDynamicInterpreterThunk(Js::ScriptFunction \* function);

#endif

template<typename T>

T ReadByteOp( const byte \*& ip

#if DBG\_DUMP

, bool isExtended = false

#endif

);

void\* \_\_cdecl operator new(size\_t byteSize, void\* previousAllocation) throw();

void \_\_cdecl operator delete(void\* allocationToFree, void\* previousAllocation) throw();

\_\_declspec(noinline) Var ProcessThunk(void\* returnAddress, void\* addressOfReturnAddress);

\_\_declspec(noinline) Var DebugProcessThunk(void\* returnAddress, void\* addressOfReturnAddress);

void AlignMemoryForAsmJs();

Var Process();

Var ProcessAsmJsModule();

Var ProcessLinkFailedAsmJsModule();

Var ProcessAsmJs();

Var ProcessProfiled();

Var ProcessUnprofiled();

Var ProcessWithDebugging();

Var DebugProcess();

// This will be called for reseting outs when resume from break on error happened

void ResetOut();

Var OP\_ArgIn0();

template <class T> void OP\_ArgOut\_Env(const unaligned T\* playout);

template <class T> void OP\_ArgOut\_A(const unaligned T\* playout);

template <class T> void OP\_ProfiledArgOut\_A(const unaligned T \* playout);

#if DBG

template <class T> void OP\_ArgOut\_ANonVar(const unaligned T\* playout);

#endif

FrameDisplay \* GetEnvForEvalCode();

BOOL OP\_BrFalse\_A(Var aValue, ScriptContext\* scriptContext);

BOOL OP\_BrTrue\_A(Var aValue, ScriptContext\* scriptContext);

BOOL OP\_BrNotNull\_A(Var aValue);

BOOL OP\_BrUndecl\_A(Var aValue);

BOOL OP\_BrNotUndecl\_A(Var aValue);

BOOL OP\_BrOnHasProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext);

BOOL OP\_BrOnNoProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext);

BOOL OP\_BrOnNoEnvProperty(Var envInstance, int32 slotIndex, uint propertyIdIndex, ScriptContext\* scriptContext);

BOOL OP\_BrOnClassConstructor(Var aValue);

RecyclableObject \* OP\_CallGetFunc(Var target);

template <class T> const byte \* OP\_Br(const unaligned T \* playout);

void OP\_AsmStartCall(const unaligned OpLayoutStartCall \* playout);

void OP\_StartCall( const unaligned OpLayoutStartCall \* playout );

void OP\_StartCall(uint outParamCount);

template <class T> void OP\_CallCommon(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

void OP\_CallAsmInternal( RecyclableObject \* function);

template <class T> void OP\_I\_AsmCall(const unaligned T\* playout) { OP\_CallAsmInternal((ScriptFunction\*)OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))); }

template <class T> void OP\_CallCommonI(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags);

template <class T> void OP\_ProfileCallCommon(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags, ProfileId profileId, InlineCacheIndex inlineCacheIndex = Js::Constants::NoInlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T> void OP\_ProfileReturnTypeCallCommon(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags, ProfileId profileId, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T> void OP\_CallPutCommon(const unaligned T \*playout, RecyclableObject \* aFunc);

template <class T> void OP\_CallPutCommonI(const unaligned T \*playout, RecyclableObject \* aFunc);

template <class T> void OP\_AsmCall(const unaligned T\* playout);

template <class T> void OP\_CallI(const unaligned T\* playout, unsigned flags) { OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags); }

template <class T> void OP\_CallIExtended(const unaligned T\* playout, unsigned flags) { OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_CallIExtendedFlags(const unaligned T\* playout, unsigned flags) { OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_CallIFlags(const unaligned T\* playout, unsigned flags) { playout->callFlags == Js::CallFlags::CallFlags\_NewTarget ? OP\_CallPutCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))) : OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags); }

template <class T> void OP\_ProfiledCallI(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId); }

template <class T> void OP\_ProfiledCallIExtended(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, Js::Constants::NoInlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIExtendedFlags(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId, Js::Constants::NoInlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, playout->inlineCacheIndex); }

template <class T> void OP\_ProfiledCallIExtendedWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, playout->inlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIExtendedFlagsWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId, playout->inlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIFlags(const unaligned T\* playout, unsigned flags) { playout->callFlags == Js::CallFlags::CallFlags\_NewTarget ? OP\_CallPutCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))) : OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId); }

template <class T> void OP\_ProfiledReturnTypeCallI(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId); }

template <class T> void OP\_ProfiledReturnTypeCallIExtended(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledReturnTypeCallIExtendedFlags(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledReturnTypeCallIFlags(const unaligned T\* playout, unsigned flags) { playout->callFlags == Js::CallFlags::CallFlags\_NewTarget ? OP\_CallPutCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))) : OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId); }

// Patching Fastpath Operations

template <class T> void OP\_GetRootProperty(unaligned T\* playout);

template <class T> void OP\_GetRootPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_GetRootProperty\_NoFastPath(unaligned T\* playout);

template <class T, bool Root, bool Method, bool CallApplyTarget> void ProfiledGetProperty(unaligned T\* playout, const Var instance);

template <class T> void OP\_ProfiledGetRootProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetRootPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_GetProperty(Var instance, unaligned T\* playout);

template <class T> void OP\_GetProperty(unaligned T\* playout);

template <class T> void OP\_GetLocalProperty(unaligned T\* playout);

template <class T> void OP\_GetSuperProperty(unaligned T\* playout);

template <class T> void OP\_GetPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_GetProperty\_NoFastPath(Var instance, unaligned T\* playout);

template <class T> void OP\_ProfiledGetProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetLocalProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetSuperProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_ProfiledGetPropertyCallApplyTarget(unaligned T\* playout);

template <class T> void OP\_GetRootMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetRootMethodProperty\_NoFastPath(unaligned T\* playout);

template <class T> void OP\_ProfiledGetRootMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetLocalMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetMethodProperty(Var varInstance, unaligned T\* playout);

template <class T> void OP\_GetMethodProperty\_NoFastPath(Var varInstance, unaligned T\* playout);

template <class T> void OP\_ProfiledGetMethodProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetLocalMethodProperty(unaligned T\* playout);

template <typename T> void OP\_GetPropertyScoped(const unaligned OpLayoutT\_ElementP<T>\* playout);

template <typename T> void OP\_GetPropertyForTypeOfScoped(const unaligned OpLayoutT\_ElementP<T>\* playout);

template <typename T> void OP\_GetPropertyScoped\_NoFastPath(const unaligned OpLayoutT\_ElementP<T>\* playout);

template <class T> void OP\_GetMethodPropertyScoped(unaligned T\* playout);

template <class T> void OP\_GetMethodPropertyScoped\_NoFastPath(unaligned T\* playout);

#if ENABLE\_PROFILE\_INFO

template <class T> void UpdateFldInfoFlagsForGetSetInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj);

template <class T> void UpdateFldInfoFlagsForCallApplyInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj);

#endif

template <class T> void OP\_SetProperty(unaligned T\* playout);

template <class T> void OP\_SetLocalProperty(unaligned T\* playout);

template <class T> void OP\_SetSuperProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetLocalProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetSuperProperty(unaligned T\* playout);

template <class T> void OP\_SetRootProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetRootProperty(unaligned T\* playout);

template <class T> void OP\_SetPropertyStrict(unaligned T\* playout);

template <class T> void OP\_ProfiledSetPropertyStrict(unaligned T\* playout);

template <class T> void OP\_SetRootPropertyStrict(unaligned T\* playout);

template <class T> void OP\_ProfiledSetRootPropertyStrict(unaligned T\* playout);

template <class T> void OP\_SetPropertyScoped(unaligned T\* playout, PropertyOperationFlags flags = PropertyOperation\_None);

template <class T> void OP\_SetPropertyScoped\_NoFastPath(unaligned T\* playout, PropertyOperationFlags flags);

template <class T> void OP\_SetPropertyScopedStrict(unaligned T\* playout);

template <class T> void OP\_ConsoleSetPropertyScoped(unaligned T\* playout);

template <class T> void DoSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T> void DoSetSuperProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T> void DoSetProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T> void DoSetSuperProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T, bool Root> void ProfiledSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T, bool Root> void ProfiledSetSuperProperty(unaligned T\* playout, Var instance, Var thisInstance, PropertyOperationFlags flags);

template <class T> void OP\_InitProperty(unaligned T\* playout);

template <class T> void OP\_InitLocalProperty(unaligned T\* playout);

template <class T> void OP\_InitRootProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclLetProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclLocalLetProperty(unaligned T\* playout);

void OP\_InitUndeclRootLetProperty(uint propertyIdIndex);

template <class T> void OP\_InitUndeclConstProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclLocalConstProperty(unaligned T\* playout);

void OP\_InitUndeclRootConstProperty(uint propertyIdIndex);

template <class T> void OP\_InitUndeclConsoleLetProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclConsoleConstProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitLocalProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitRootProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitUndeclProperty(unaligned T\* playout);

template <class T> void DoInitProperty(unaligned T\* playout, Var instance);

template <class T> void DoInitProperty\_NoFastPath(unaligned T\* playout, Var instance);

template <class T> void ProfiledInitProperty(unaligned T\* playout, Var instance);

template <class T> bool TrySetPropertyLocalFastPath(unaligned T\* playout, PropertyId pid, Var instance, InlineCache\*& inlineCache, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool doProfile> Var ProfiledDivide(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId);

template <bool doProfile> Var ProfileModulus(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId);

template <bool doProfile> Var ProfiledSwitch(Var exp, ProfileId profileId);

// Non-patching Fastpath operations

template <typename T> void OP\_GetElementI(const unaligned T\* playout);

template <typename T> void OP\_ProfiledGetElementI(const unaligned OpLayoutDynamicProfile<T>\* playout);

template <typename T> void OP\_SetElementI(const unaligned T\* playout, PropertyOperationFlags flags = PropertyOperation\_None);

template <typename T> void OP\_ProfiledSetElementI(const unaligned OpLayoutDynamicProfile<T>\* playout, PropertyOperationFlags flags = PropertyOperation\_None);

template <typename T> void OP\_SetElementIStrict(const unaligned T\* playout);

template <typename T> void OP\_ProfiledSetElementIStrict(const unaligned OpLayoutDynamicProfile<T>\* playout);

template<class T> void OP\_LdLen(const unaligned T \*const playout);

template<class T> void OP\_ProfiledLdLen(const unaligned OpLayoutDynamicProfile<T> \*const playout);

Var OP\_ProfiledLdThis(Var thisVar, int moduleID, ScriptContext\* scriptContext);

Var OP\_ProfiledStrictLdThis(Var thisVar, ScriptContext\* scriptContext);

template <class T> void OP\_SetArrayItemI\_CI4(const unaligned T\* playout);

template <class T> void OP\_SetArrayItemC\_CI4(const unaligned T\* playout);

template <class T> void OP\_SetArraySegmentItem\_CI4(const unaligned T\* playout);

template <class T> void SetArrayLiteralItem(JavascriptArray \*arr, uint32 index, T value);

void OP\_SetArraySegmentVars(const unaligned OpLayoutAuxiliary \* playout);

template <class T> void OP\_NewScArray(const unaligned T \* playout);

template <bool Profiled, class T> void ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout);

template <class T> void OP\_ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout) { ProfiledNewScArray<true, T>(playout); }

template <class T> void OP\_ProfiledNewScArray\_NoProfile(const unaligned OpLayoutDynamicProfile<T> \* playout) { ProfiledNewScArray<false, T>(playout); }

void OP\_NewScIntArray(const unaligned OpLayoutAuxiliary \* playout);

void OP\_NewScFltArray(const unaligned OpLayoutAuxiliary \* playout);

void OP\_ProfiledNewScIntArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout);

void OP\_ProfiledNewScFltArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout);

template <class T> void OP\_LdArrayHeadSegment(const unaligned T\* playout);

inline Var GetFunctionExpression();

template <class T> inline void OP\_LdFunctionExpression(const unaligned T\* playout);

template <class T> inline void OP\_StFunctionExpression(const unaligned T\* playout);

template <class T> inline void OP\_StLocalFunctionExpression(const unaligned T\* playout);

void OP\_StFunctionExpression(Var instance, Var value, PropertyIdIndexType index);

template <class T> inline void OP\_LdNewTarget(const unaligned T\* playout);

inline Var OP\_Ld\_A(Var aValue);

inline Var OP\_LdLocalObj();

void OP\_ChkUndecl(Var aValue);

void OP\_ChkNewCallFlag();

void OP\_EnsureNoRootProperty(uint propertyIdIndex);

void OP\_EnsureNoRootRedeclProperty(uint propertyIdIndex);

void OP\_ScopedEnsureNoRedeclProperty(Var aValue, uint propertyIdIndex, Var aValue2);

Var OP\_InitUndecl();

void OP\_InitUndeclSlot(Var aValue, int32 slot);

template <class T> inline void OP\_InitInnerFld(const unaligned T \* playout);

template <class T> inline void OP\_InitLetFld(const unaligned T \* playout);

template <class T> inline void OP\_InitLocalLetFld(const unaligned T\* playout);

template <class T> inline void OP\_InitInnerLetFld(const unaligned T \* playout);

template <class T> inline void OP\_InitRootLetFld(const unaligned T \* playout);

template <class T> inline void OP\_InitConstFld(const unaligned T \* playout);

template <class T> inline void OP\_InitRootConstFld(const unaligned T \* playout);

template <class T> inline void DoInitLetFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags = PropertyOperation\_None);

template <class T> inline void DoInitConstFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags = PropertyOperation\_None);

template <class T> inline void OP\_InitClassMember(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberComputedName(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberGet(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberSet(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberGetComputedName(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberSetComputedName(const unaligned T \* playout);

template<typename T> uint32 LogSizeOf();

template <typename T2> inline void OP\_LdArr( uint32 index, RegSlot value );

template <class T> inline void OP\_LdArrFunc(const unaligned T\* playout);

template <class T> inline void OP\_ReturnDb(const unaligned T\* playout);

template<typename T> T GetArrayViewOverflowVal();

template <typename T2> inline void OP\_StArr( uint32 index, RegSlot value );

template <class T> inline Var OP\_LdAsmJsSlot(Var instance, const unaligned T\* playout );

template <class T, typename T2> inline void OP\_StSlotPrimitive(const unaligned T\* playout);

template <class T, typename T2> inline void OP\_LdSlotPrimitive( const unaligned T\* playout );

template <class T> inline void OP\_LdArrGeneric ( const unaligned T\* playout );

template <class T> inline void OP\_LdArrConstIndex( const unaligned T\* playout );

template <class T> inline void OP\_StArrGeneric ( const unaligned T\* playout );

template <class T> inline void OP\_StArrConstIndex( const unaligned T\* playout );

inline Var OP\_LdSlot(Var instance, int32 slotIndex);

inline Var OP\_LdObjSlot(Var instance, int32 slotIndex);

inline Var OP\_LdFrameDisplaySlot(Var instance, int32 slotIndex);

template <class T> inline Var OP\_LdSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdInnerSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdInnerSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdInnerObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdInnerObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdEnvSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdEnvSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdEnvObj(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdEnvObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdEnvObjSlot(Var instance, const unaligned T\* playout);

inline void OP\_StSlot(Var instance, int32 slotIndex, Var value);

inline void OP\_StSlotChkUndecl(Var instance, int32 slotIndex, Var value);

inline void OP\_StEnvSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void OP\_StEnvSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void OP\_StObjSlot(Var instance, int32 slotIndex, Var value);

inline void OP\_StObjSlotChkUndecl(Var instance, int32 slotIndex, Var value);

inline void OP\_StEnvObjSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void OP\_StEnvObjSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void\* OP\_LdArgCnt();

template <bool letArgs> Var LdHeapArgumentsImpl(Var argsArray, ScriptContext\* scriptContext);

inline Var OP\_LdHeapArguments(Var argsArray, ScriptContext\* scriptContext);

inline Var OP\_LdLetHeapArguments(Var argsArray, ScriptContext\* scriptContext);

inline Var OP\_LdHeapArgsCached(ScriptContext\* scriptContext);

inline Var OP\_LdLetHeapArgsCached(ScriptContext\* scriptContext);

inline Var OP\_LdStackArgPtr();

inline Var OP\_LdArgumentsFromFrame();

Var OP\_NewScObjectSimple();

void OP\_NewScObjectLiteral(const unaligned OpLayoutAuxiliary \* playout);

void OP\_NewScObjectLiteral\_LS(const unaligned OpLayoutAuxiliary \* playout, RegSlot& target);

void OP\_LdPropIds(const unaligned OpLayoutAuxiliary \* playout);

template <bool Profile, bool JITLoopBody> void LoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

LoopHeader const \* DoLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, const bool doProfileLoopCheck, bool isFirstIteration);

template <bool Profile, bool JITLoopBody> void ProfiledLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

void OP\_RecordImplicitCall(uint loopNumber);

template <class T, bool Profiled, bool ICIndex> void OP\_NewScObject\_Impl(const unaligned T\* playout, InlineCacheIndex inlineCacheIndex = Js::Constants::NoInlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T, bool Profiled> void OP\_NewScObjArray\_Impl(const unaligned T\* playout, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T> void OP\_NewScObject(const unaligned T\* playout) { OP\_NewScObject\_Impl<T, false, false>(playout); }

template <class T> void OP\_NewScObjectNoCtorFull(const unaligned T\* playout);

template <class T> void OP\_NewScObjectSpread(const unaligned T\* playout) { OP\_NewScObject\_Impl<T, false, false>(playout, Js::Constants::NoInlineCacheIndex, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_NewScObjArray(const unaligned T\* playout) { OP\_NewScObjArray\_Impl<T, false>(playout); }

template <class T> void OP\_NewScObjArraySpread(const unaligned T\* playout) { OP\_NewScObjArray\_Impl<T, false>(playout, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_ProfiledNewScObject(const unaligned OpLayoutDynamicProfile<T>\* playout) { OP\_NewScObject\_Impl<T, true, false>(playout); }

template <class T> void OP\_ProfiledNewScObjectSpread(const unaligned OpLayoutDynamicProfile<T>\* playout) { OP\_NewScObject\_Impl<T, true, false>(playout, Js::Constants::NoInlineCacheIndex, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_ProfiledNewScObjectWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout) { OP\_NewScObject\_Impl<T, true, true>(playout, playout->inlineCacheIndex); }

template <class T> void OP\_ProfiledNewScObjArray(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, true>(playout); }

template <class T> void OP\_ProfiledNewScObjArray\_NoProfile(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, false>(playout); }

template <class T> void OP\_ProfiledNewScObjArraySpread(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, true>(playout, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_ProfiledNewScObjArraySpread\_NoProfile(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, true>(playout, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

Var NewScObject\_Helper(Var target, ArgSlot ArgCount, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

Var ProfiledNewScObject\_Helper(Var target, ArgSlot ArgCount, ProfileId profileId, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T, bool Profiled, bool ICIndex> Var OP\_NewScObjectNoArg\_Impl(const unaligned T \*playout, InlineCacheIndex inlineCacheIndex = Js::Constants::NoInlineCacheIndex);

void OP\_NewScObject\_A\_Impl(const unaligned OpLayoutAuxiliary \* playout, RegSlot \*target = nullptr);

void OP\_NewScObject\_A(const unaligned OpLayoutAuxiliary \* playout) { return OP\_NewScObject\_A\_Impl(playout); }

void OP\_InitCachedFuncs(const unaligned OpLayoutAuxNoReg \* playout);

Var OP\_GetCachedFunc(Var instance, int32 index);

void OP\_CommitScope(const unaligned OpLayoutAuxNoReg \* playout);

void OP\_CommitScopeHelper(const unaligned OpLayoutAuxNoReg \*playout, const PropertyIdArray \*propIds);

void OP\_TryCatch(const unaligned OpLayoutBr\* playout);

void ProcessCatch();

int ProcessFinally();

void ProcessTryCatchBailout(EHBailoutData \* innermostEHBailoutData, uint32 tryNestingDepth);

void OP\_TryFinally(const unaligned OpLayoutBr\* playout);

void OP\_TryFinallyWithYield(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException, Js::RegSlot regOffset);

void OP\_ResumeCatch();

void OP\_ResumeFinally(const byte\* ip, Js::JumpOffset jumpOffset, RegSlot exceptionRegSlot, RegSlot offsetRegSlot);

inline Var OP\_ResumeYield(Var yieldDataVar, RegSlot yieldStarIterator = Js::Constants::NoRegister);

template <typename T> void OP\_IsInst(const unaligned T \* playout);

template <class T> void OP\_InitClass(const unaligned OpLayoutT\_Class<T> \* playout);

inline Var OP\_LdSuper(ScriptContext \* scriptContext);

inline Var OP\_LdSuperCtor(ScriptContext \* scriptContext);

inline Var OP\_ScopedLdSuper(ScriptContext \* scriptContext);

inline Var OP\_ScopedLdSuperCtor(ScriptContext \* scriptContext);

template <typename T> void OP\_LdElementUndefined(const unaligned OpLayoutT\_ElementU<T>\* playout);

template <typename T> void OP\_LdLocalElementUndefined(const unaligned OpLayoutT\_ElementRootU<T>\* playout);

template <typename T> void OP\_LdElementUndefinedScoped(const unaligned OpLayoutT\_ElementScopedU<T>\* playout);

void OP\_SpreadArrayLiteral(const unaligned OpLayoutReg2Aux \* playout);

template <LayoutSize layoutSize,bool profiled> const byte \* OP\_ProfiledLoopStart(const byte \*ip);

template <LayoutSize layoutSize,bool profiled> const byte \* OP\_ProfiledLoopEnd(const byte \*ip);

template <LayoutSize layoutSize,bool profiled> const byte \* OP\_ProfiledLoopBodyStart(const byte \*ip);

template <typename T> void OP\_ApplyArgs(const unaligned OpLayoutT\_Reg5<T> \* playout);

template <class T> void OP\_EmitTmpRegCount(const unaligned OpLayoutT\_Unsigned1<T> \* ip);

Var InnerScopeFromIndex(uint32 index) const;

void SetInnerScopeFromIndex(uint32 index, Var scope);

void OP\_NewInnerScopeSlots(uint index, uint count, int scopeIndex, ScriptContext \*scriptContext, FunctionBody \*functionBody);

template <typename T> void OP\_CloneInnerScopeSlots(const unaligned OpLayoutT\_Unsigned1<T> \*playout);

template <typename T> void OP\_CloneBlockScope(const unaligned OpLayoutT\_Unsigned1<T> \*playout);

FrameDisplay \* OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdFrameDisplaySetLocal(void \*argHead, void \*argEnv, ScriptContext \*scriptContext);

template <bool innerFD> FrameDisplay \* OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdFuncExprFrameDisplaySetLocal(void \*argHead1, void \*argHead2, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext);

template <class T> void OP\_NewStackScFunc(const unaligned T \* playout);

template <class T> void OP\_NewInnerStackScFunc(const unaligned T \* playout);

template <class T> void OP\_DeleteFld(const unaligned T \* playout);

template <class T> void OP\_DeleteLocalFld(const unaligned T \* playout);

template <class T> void OP\_DeleteRootFld(const unaligned T \* playout);

template <class T> void OP\_DeleteFldStrict(const unaligned T \* playout);

template <class T> void OP\_DeleteRootFldStrict(const unaligned T \* playout);

template <typename T> void OP\_ScopedDeleteFld(const unaligned OpLayoutT\_ElementScopedC<T> \* playout);

template <typename T> void OP\_ScopedDeleteFldStrict(const unaligned OpLayoutT\_ElementScopedC<T> \* playout);

template <class T> void OP\_ScopedLdInst(const unaligned T \* playout);

template <typename T> void OP\_ScopedInitFunc(const unaligned OpLayoutT\_ElementScopedC<T> \* playout);

template <class T> void OP\_ClearAttributes(const unaligned T \* playout);

template <class T> void OP\_InitGetFld(const unaligned T \* playout);

template <class T> void OP\_InitSetFld(const unaligned T \* playout);

template <class T> void OP\_InitSetElemI(const unaligned T \* playout);

template <class T> void OP\_InitGetElemI(const unaligned T \* playout);

template <class T> void OP\_InitComputedProperty(const unaligned T \* playout);

template <class T> void OP\_InitProto(const unaligned T \* playout);

uint CallLoopBody(JavascriptMethod address);

uint CallAsmJsLoopBody(JavascriptMethod address);

void DoInterruptProbe();

void CheckIfLoopIsHot(uint profiledLoopCounter);

bool CheckAndResetImplicitCall(DisableImplicitFlags prevDisableImplicitFlags,ImplicitCallFlags savedImplicitCallFlags);

class PushPopFrameHelper

{

public:

PushPopFrameHelper(InterpreterStackFrame \*interpreterFrame, void \*returnAddress, void \*addressOfReturnAddress)

: m\_threadContext(interpreterFrame->GetScriptContext()->GetThreadContext()), m\_interpreterFrame(interpreterFrame), m\_isHiddenFrame(false)

{

interpreterFrame->returnAddress = returnAddress; // Ensure these are set before pushing to interpreter frame list

interpreterFrame->addressOfReturnAddress = addressOfReturnAddress;

if (interpreterFrame->GetFunctionBody()->GetIsAsmJsFunction())

{

m\_isHiddenFrame = true;

}

else

{

m\_threadContext->PushInterpreterFrame(interpreterFrame);

}

}

~ PushPopFrameHelper()

{

if (!m\_isHiddenFrame)

{

Js::InterpreterStackFrame \*interpreterFrame = m\_threadContext->PopInterpreterFrame();

AssertMsg(interpreterFrame == m\_interpreterFrame,

"Interpreter frame chain corrupted?");

}

}

private:

ThreadContext \*m\_threadContext;

InterpreterStackFrame \*m\_interpreterFrame;

bool m\_isHiddenFrame;

};

inline InlineCache\* GetInlineCache(uint cacheIndex)

{

Assert(this->inlineCaches != nullptr);

Assert(cacheIndex < this->inlineCacheCount);

return reinterpret\_cast<InlineCache \*>(this->inlineCaches[cacheIndex]);

}

inline IsInstInlineCache\* GetIsInstInlineCache(uint cacheIndex)

{

return m\_functionBody->GetIsInstInlineCache(cacheIndex);

}

inline PropertyId GetPropertyIdFromCacheId(uint cacheIndex)

{

return m\_functionBody->GetPropertyIdFromCacheId(cacheIndex);

}

void InitializeStackFunctions(StackScriptFunction \* scriptFunctions);

StackScriptFunction \* GetStackNestedFunction(uint index);

void SetExecutingStackFunction(ScriptFunction \* scriptFunction);

friend class StackScriptFunction;

void InitializeClosures();

void SetLocalFrameDisplay(FrameDisplay \*frameDisplay);

Var GetLocalClosure() const;

void SetLocalClosure(Var closure);

void TrySetRetOffset();

};

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// Used to track how many interpreter stack frames we have on stack.

class InterpreterThunkStackCountTracker

{

public:

InterpreterThunkStackCountTracker() { ++s\_count; }

~InterpreterThunkStackCountTracker() { --s\_count; }

static int GetCount() { return s\_count; }

private:

\_\_declspec(thread) static int s\_count;

};

#endif

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "EHBailoutData.h"

#include "Library\JavascriptRegularExpression.h"

#if DBG\_DUMP

#include "ByteCode\OpCodeUtilAsmJs.h"

#endif

#include "Language\InterpreterStackFrame.h"

#include "Library\JavascriptGeneratorFunction.h"

#include "Bytecode\ByteCodeDumper.h"

///----------------------------------------------------------------------------

///

/// macros PROCESS\_INtoOUT

///

/// This set of macros defines standard patterns for processing OpCodes in

/// RcInterpreter::Run(). Each macro is named for "in" - "out":

/// - A: Var

/// - I: Integer

/// - R: Double

/// - X: Nothing

///

/// Examples:

/// - "A2toA1" reads two registers, each storing an Var, and writes a single

/// register with a new Var.

/// - "A1I1toA2" reads two registers, first an Var and second an Int32, then

/// writes two Var registers.

///

/// Although these could use lookup tables to standard OpLayout types, this

/// additional indirection would slow the main interpreter loop further by

/// preventing the main 'switch' statement from using the OpCode to become a

/// direct local-function jump.

///----------------------------------------------------------------------------

#define PROCESS\_FALLTHROUGH(name, func) \

case OpCode::name:

#define PROCESS\_FALLTHROUGH\_COMMON(name, func, suffix) \

case OpCode::name:

#define PROCESS\_READ\_LAYOUT(name, layout, suffix) \

CompileAssert(OpCodeInfo<OpCode::name>::Layout == OpLayoutType::layout); \

const unaligned OpLayout##layout##suffix \* playout = m\_reader.layout##suffix(ip); \

Assert((playout != nullptr) == (Js::OpLayoutType::##layout != Js::OpLayoutType::Empty)); // Make sure playout is used

#define PROCESS\_NOP\_COMMON(name, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

break; \

}

#define PROCESS\_NOP(name, layout) PROCESS\_NOP\_COMMON(name, layout,)

#define PROCESS\_CUSTOM\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CUSTOM(name, func, layout) PROCESS\_CUSTOM\_COMMON(name, func, layout,)

#define PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, regslot, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CUSTOM\_L(name, func, layout, regslot) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, regslot,)

#define PROCESS\_CUSTOM\_L\_Arg\_COMMON(name, func, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, Arg, Arg, suffix)

#define PROCESS\_CUSTOM\_L\_Arg2\_COMMON(name, func, layout, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, Arg, suffix)

#define PROCESS\_CUSTOM\_L\_Arg(name, func) PROCESS\_CUSTOM\_L\_COMMON(name, func, Arg, Arg,)

#define PROCESS\_CUSTOM\_ArgNoSrc\_COMMON(name, func, suffix) PROCESS\_CUSTOM\_COMMON(name, func, ArgNoSrc, suffix)

#define PROCESS\_CUSTOM\_ArgNoSrc(name, func) PROCESS\_CUSTOM\_COMMON(name, func, ArgNoSrc,)

#define PROCESS\_CUSTOM\_L\_R0\_COMMON(name, func, layout, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, R0, suffix)

#define PROCESS\_CUSTOM\_L\_R0(name, func, layout) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, R0,)

#define PROCESS\_CUSTOM\_L\_Value\_COMMON(name, func, layout, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, Value, suffix)

#define PROCESS\_CUSTOM\_L\_Value(name, func, layout) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, Value,)

#define PROCESS\_TRY(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Br,); \

func(playout); \

ip = m\_reader.GetIP(); \

break; \

}

#define PROCESS\_EMPTY(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Empty, ); \

func(); \

ip = m\_reader.GetIP(); \

break; \

}

#define PROCESS\_TRYBR2\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg2, suffix); \

func((const byte\*)(playout + 1), playout->RelativeJumpOffset, playout->R1, playout->R2); \

ip = m\_reader.GetIP(); \

break; \

}

#define PROCESS\_CALL\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CALL(name, func, layout) PROCESS\_CALL\_COMMON(name, func, layout,)

#define PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, flags, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout, flags); \

break; \

}

#define PROCESS\_CALL\_FLAGS(name, func, layout, regslot) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, regslot,)

#define PROCESS\_CALL\_FLAGS\_None\_COMMON(name, func, layout, suffix) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_None, suffix)

#define PROCESS\_CALL\_FLAGS\_None(name, func, layout) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_None,)

#define PROCESS\_CALL\_FLAGS\_Value\_COMMON(name, func, layout, suffix) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_Value, suffix)

#define PROCESS\_CALL\_FLAGS\_Value(name, func, layout) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_Value,)

#define PROCESS\_CALL\_FLAGS\_CallEval\_COMMON(name, func, layout, suffix) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_ExtraArg, suffix)

#define PROCESS\_CALL\_FLAGS\_CallEval(name, func, layout) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_ExtraArg,)

#define PROCESS\_A1toXX\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func(GetRegAllowStackVar(playout->R0)); \

break; \

}

#define PROCESS\_A1toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func(GetReg(playout->R0)); \

break; \

}

#define PROCESS\_A1toXX(name, func) PROCESS\_A1toXX\_COMMON(name, func,)

#define PROCESS\_A1toXXMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func(GetReg(playout->R0), GetScriptContext()); \

break; \

}

#define PROCESS\_A1toXXMem(name, func) PROCESS\_A1toXXMem\_COMMON(name, func,)

#define PROCESS\_A1toXXMemNonVar\_COMMON(name, func, type, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func((type)GetNonVarReg(playout->R0), GetScriptContext()); \

break; \

}

#define PROCESS\_A1toXXMemNonVar(name, func, type) PROCESS\_A1toXXMemNonVar\_COMMON(name, func, type,)

#define PROCESS\_XXtoA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

SetReg(playout->R0, \

func()); \

break; \

}

#define PROCESS\_XXtoA1(name, func) PROCESS\_XXtoA1\_COMMON(name, func,)

#define PROCESS\_XXtoA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

SetNonVarReg(playout->R0, \

func()); \

break; \

}

#define PROCESS\_XXtoA1NonVar(name, func) PROCESS\_XXtoA1NonVar\_COMMON(name, func,)

#define PROCESS\_XXtoA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

SetReg(playout->R0, \

func(GetScriptContext())); \

break; \

}

#define PROCESS\_XXtoA1Mem(name, func) PROCESS\_XXtoA1Mem\_COMMON(name, func,)

#define PROCESS\_A1toA1\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetRegAllowStackVar(playout->R0, \

func(GetRegAllowStackVar(playout->R1))); \

break; \

}

#define PROCESS\_A1toA1\_ALLOW\_STACK(name, func) PROCESS\_A1toA1\_ALLOW\_STACK\_COMMON(name, func,)

#define PROCESS\_A1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1))); \

break; \

}

#define PROCESS\_A1toA1(name, func) PROCESS\_A1toA1\_COMMON(name, func,)

#define PROCESS\_A1toA1Profiled\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ProfiledReg2, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), playout->profileId)); \

break; \

}

#define PROCESS\_A1toA1Profiled(name, func) PROCESS\_A1toA1Profiled\_COMMON(name, func,)

#define PROCESS\_A1toA1CallNoArg\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetReg(playout->R0, \

func(playout)); \

break; \

}

#define PROCESS\_A1toA1CallNoArg(name, func, layout) PROCESS\_A1toA1CallNoArg\_COMMON(name, func, layout,)

#define PROCESS\_A1toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1),GetScriptContext())); \

break; \

}

#define PROCESS\_A1toA1Mem(name, func) PROCESS\_A1toA1Mem\_COMMON(name, func,)

#define PROCESS\_A1toA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1))); \

break; \

}

#define PROCESS\_A1toA1NonVar(name, func) PROCESS\_A1toA1NonVar\_COMMON(name, func,)

#define PROCESS\_A1toA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1),GetScriptContext())); \

break; \

}

#define PROCESS\_A1toA1MemNonVar(name, func) PROCESS\_A1toA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_INNERtoA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, InnerScopeFromIndex(playout->C1)); \

break; \

}

#define PROCESS\_INNERtoA1(name, fun) PROCESS\_INNERtoA1\_COMMON(name, func,)

#define PROCESS\_U1toINNERMemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Unsigned1, suffix); \

SetInnerScopeFromIndex(playout->C1, func(GetScriptContext())); \

break; \

}

#define PROCESS\_U1toINNERMemNonVar(name, func) PROCESS\_U1toINNERMemNonVar\_COMMON(name, func,)

#define PROCESS\_XXINNERtoA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetNonVarReg(playout->R0, \

func(InnerScopeFromIndex(playout->C1), GetScriptContext())); \

break; \

}

#define PROCESS\_XXINNERtoA1MemNonVar(name, func) PROCESS\_XXINNERtoA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_A1INNERtoA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2Int1, suffix); \

SetNonVarReg(playout->R0, \

func(InnerScopeFromIndex(playout->C1), GetNonVarReg(playout->R1), GetScriptContext())); \

break; \

}

#define PROCESS\_A1LOCALtoA1MemNonVar(name, func) PROCESS\_A1LOCALtoA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_LOCALI1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, \

func(this->localClosure, playout->C1)); \

break; \

}

#define PROCESS\_LOCALI1toA1(name, func) PROCESS\_LOCALI1toA1\_COMMON(name, func,)

#define PROCESS\_A1I1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2Int1, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), playout->C1)); \

break; \

}

#define PROCESS\_A1I1toA1(name, func) PROCESS\_A1I1toA1\_COMMON(name, func,)

#define PROCESS\_A1I1toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2Int1, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), playout->C1, GetScriptContext())); \

break; \

}

#define PROCESS\_A1I1toA1Mem(name, func) PROCESS\_A1I1toA1Mem\_COMMON(name, func,)

#define PROCESS\_RegextoA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, \

func(this->m\_functionBody->GetLiteralRegex(playout->C1), GetScriptContext())); \

break; \

}

#define PROCESS\_RegextoA1(name, func) PROCESS\_RegextoA1\_COMMON(name, func,)

#define PROCESS\_A2toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

func(GetReg(playout->R0), GetReg(playout->R1)); \

break; \

}

#define PROCESS\_A2toXX(name, func) PROCESS\_A2toXX\_COMMON(name, func,)

#define PROCESS\_A2toXXMemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

func(GetNonVarReg(playout->R0), GetNonVarReg(playout->R1), GetScriptContext()); \

break; \

}

#define PROCESS\_A2toXXMemNonVar(name, func) PROCESS\_A2toXXMemNonVar\_COMMON(name, func,)

#define PROCESS\_A1NonVarToA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetReg(playout->R0, \

func(GetNonVarReg(playout->R1))); \

break; \

}

#define PROCESS\_A2NonVarToA1Reg\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetReg(playout->R0, \

func(GetNonVarReg(playout->R1), playout->R2)); \

break; \

}

#define PROCESS\_A2toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2),GetScriptContext())); \

break; \

}

#define PROCESS\_A2toA1Mem(name, func) PROCESS\_A2toA1Mem\_COMMON(name, func,)

#define PROCESS\_A2toA1MemProfiled\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ProfiledReg3, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2),GetScriptContext(), playout->profileId)); \

break; \

}

#define PROCESS\_A2toA1MemProfiled(name, func) PROCESS\_A2toA1MemProfiled\_COMMON(name, func,)

#define PROCESS\_A2toA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1), GetNonVarReg(playout->R2))); \

break; \

}

#define PROCESS\_A2toA1NonVar(name, func) PROCESS\_A2toA1NonVar\_COMMON(name, func,)

#define PROCESS\_A2toA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1), GetNonVarReg(playout->R2),GetScriptContext())); \

break; \

}

#define PROCESS\_A2toA1MemNonVar(name, func) PROCESS\_A2toA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_CMMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2), GetScriptContext()) ? JavascriptBoolean::OP\_LdTrue(GetScriptContext()) : \

JavascriptBoolean::OP\_LdFalse(GetScriptContext())); \

break; \

}

#define PROCESS\_CMMem(name, func) PROCESS\_CMMem\_COMMON(name, func,)

#define PROCESS\_ELEM\_RtU\_to\_XX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementRootU, suffix); \

func(playout->PropertyIdIndex); \

break; \

}

#define PROCESS\_ELEM\_RtU\_to\_XX(name, func) PROCESS\_ELEM\_RtU\_to\_XX\_COMMON(name, func,)

#define PROCESS\_ELEM\_C2\_to\_XX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementScopedC, suffix); \

func(GetEnvForEvalCode(), playout->PropertyIdIndex, GetReg(playout->Value)); \

break; \

}

#define PROCESS\_ELEM\_C2\_to\_XX(name, func) PROCESS\_ELEM\_C2\_to\_XX\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_SLOT\_FB\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlot, suffix); \

SetReg(playout->Value, \

func((FrameDisplay\*)GetNonVarReg(playout->Instance), reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(playout->SlotIndex)))); \

break; \

}

#define PROCESS\_GET\_ELEM\_SLOT\_FB(name, func) PROCESS\_GET\_ELEM\_SLOT\_FB\_COMMON(name, func,)

#define PROCESS\_GET\_SLOT\_FB\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI1, suffix); \

SetReg(playout->Value, \

func(this->GetFrameDisplayForNestedFunc(), reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(playout->SlotIndex)))); \

break; \

}

#define PROCESS\_GET\_SLOT\_FB(name, func) PROCESS\_GET\_SLOT\_FB\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_IMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementI, suffix); \

SetReg(playout->Value, \

func(GetReg(playout->Instance), GetReg(playout->Element), GetScriptContext())); \

break; \

}

#define PROCESS\_GET\_ELEM\_IMem(name, func) PROCESS\_GET\_ELEM\_IMem\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_IMem\_Strict\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementI, suffix); \

SetReg(playout->Value, \

func(GetReg(playout->Instance), GetReg(playout->Element), GetScriptContext(), PropertyOperation\_StrictMode)); \

break; \

}

#define PROCESS\_GET\_ELEM\_IMem\_Strict(name, func) PROCESS\_GET\_ELEM\_IMem\_Strict\_COMMON(name, func,)

#define PROCESS\_BR(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Br,); \

ip = func(playout); \

break; \

}

#ifdef BYTECODE\_BRANCH\_ISLAND

#define PROCESS\_BRLONG(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrLong,); \

ip = func(playout); \

break; \

}

#endif

#define PROCESS\_BRS(name,func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrS,); \

if (func(playout->val,GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRB\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetReg(playout->R1))) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRB(name, func) PROCESS\_BRB\_COMMON(name, func,)

#define PROCESS\_BRB\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetRegAllowStackVar(playout->R1))) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRB\_ALLOW\_STACK(name, func) PROCESS\_BRB\_ALLOW\_STACK\_COMMON(name, func,)

#define PROCESS\_BRBS\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetReg(playout->R1), GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRBS(name, func) PROCESS\_BRBS\_COMMON(name, func,)

#define PROCESS\_BRBReturnP1toA1\_COMMON(name, func, type, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg2, suffix); \

SetReg(playout->R1, func((type)GetNonVarReg(playout->R2))); \

if (!GetReg(playout->R1)) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRBReturnP1toA1(name, func, type) PROCESS\_BRBReturnP1toA1\_COMMON(name, func, type,)

#define PROCESS\_BRBMem\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetRegAllowStackVar(playout->R1),GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRBMem\_ALLOW\_STACK(name, func) PROCESS\_BRBMem\_ALLOW\_STACK\_COMMON(name, func,)

#define PROCESS\_BRCMem\_COMMON(name, func,suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg2, suffix); \

if (func(GetReg(playout->R1), GetReg(playout->R2),GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRCMem(name, func) PROCESS\_BRCMem\_COMMON(name, func,)

#define PROCESS\_BRPROP(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrProperty,); \

if (func(GetReg(playout->Instance), playout->PropertyIdIndex, GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRLOCALPROP(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrLocalProperty,); \

if (func(this->localClosure, playout->PropertyIdIndex, GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRENVPROP(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrEnvProperty,); \

if (func(LdEnv(), playout->SlotIndex, playout->PropertyIdIndex, GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_W1(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, W1,); \

func(playout->C1, GetScriptContext()); \

break; \

}

#define PROCESS\_U1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, \

func(playout->C1,GetScriptContext())); \

break; \

}

#define PROCESS\_U1toA1(name, func) PROCESS\_U1toA1\_COMMON(name, func,)

#define PROCESS\_U1toA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetNonVarReg(playout->R0, \

func(playout->C1)); \

break; \

}

#define PROCESS\_U1toA1NonVar(name, func) PROCESS\_U1toA1NonVar\_COMMON(name, func,)

#define PROCESS\_U1toA1NonVar\_FuncBody\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetNonVarReg(playout->R0, \

func(playout->C1,GetScriptContext(), this->m\_functionBody)); \

break; \

}

#define PROCESS\_U1toA1NonVar\_FuncBody(name, func) PROCESS\_U1toA1NonVar\_FuncBody\_COMMON(name, func,)

#define PROCESS\_A1I2toXXNonVar\_FuncBody(name, func) PROCESS\_A1I2toXXNonVar\_FuncBody\_COMMON(name, func,)

#define PROCESS\_A1I2toXXNonVar\_FuncBody\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

func(playout->R0, playout->R1, playout->R2, GetScriptContext(), this->m\_functionBody); \

break; \

}

#define PROCESS\_A1U1toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

func(GetNonVarReg(playout->R0), playout->C1); \

break; \

}

#define PROCESS\_A1U1toXX(name, func) PROCESS\_A1U1toXX\_COMMON(name, func,)

#define PROCESS\_EnvU1toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Unsigned1, suffix); \

func(LdEnv(), playout->C1); \

break; \

}

#define PROCESS\_EnvU1toXX(name, func) PROCESS\_EnvU1toXX\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_SLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func(GetNonVarReg(playout->Instance), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_SLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_SLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_GET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func((Var\*)GetLocalClosure(), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_LOCALSLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_GET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func(InnerScopeFromIndex(playout->SlotIndex1), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_INNERSLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_GET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func(LdEnv(), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_ENVSLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_SET\_ELEM\_SLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlot, suffix); \

func(GetNonVarReg(playout->Instance), playout->SlotIndex, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_SLOTNonVar(name, func) PROCESS\_SET\_ELEM\_SLOTNonVar\_COMMON(name, func,)

#define PROCESS\_SET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI1, suffix); \

func((Var\*)GetLocalClosure(), playout->SlotIndex, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_LOCALSLOTNonVar(name, func) PROCESS\_SET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func,)

#define PROCESS\_SET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI2, suffix); \

func(InnerScopeFromIndex(playout->SlotIndex1), playout->SlotIndex2, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_INNERSLOTNonVar(name, func) PROCESS\_SET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func,)

#define PROCESS\_SET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI2, suffix); \

func(LdEnv(), playout->SlotIndex1, playout->SlotIndex2, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_ENVSLOTNonVar(name, func) PROCESS\_SET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A3toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg4, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2), GetReg(playout->R3), GetScriptContext())); \

break; \

}

#define PROCESS\_A3toA1Mem(name, func) PROCESS\_A3toA1Mem\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A2I1toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3B1, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2), playout->B3, GetScriptContext())); \

break; \

}

#define PROCESS\_A2I1toA1Mem(name, func) PROCESS\_A2I1toA1Mem\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A2I1toXXMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2B1, suffix); \

func(GetReg(playout->R0), GetReg(playout->R1), playout->B2, scriptContext); \

break; \

}

#define PROCESS\_A2I1toXXMem(name, func) PROCESS\_A2I1toXXMem\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A3I1toXXMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3B1, suffix); \

func(GetReg(playout->R0), GetReg(playout->R1), GetReg(playout->R2), playout->B3, scriptContext); \

break; \

}

#define PROCESS\_A3I1toXXMem(name, func) PROCESS\_A3I1toXXMem\_COMMON(name, func,)

#if ENABLE\_PROFILE\_INFO

#define PROCESS\_IP\_TARG\_IMPL(name, func, layoutSize) \

case OpCode::name: \

{ \

Assert(!switchProfileMode); \

ip = func<layoutSize, INTERPRETERPROFILE>(ip); \

if(switchProfileMode) \

{ \

m\_reader.SetIP(ip); \

return nullptr; \

} \

break; \

}

#else

#define PROCESS\_IP\_TARG\_IMPL(name, func, layoutSize) \

case OpCode::name: \

{ \

ip = func<layoutSize, INTERPRETERPROFILE>(ip); \

break; \

}

#endif

#define PROCESS\_IP\_TARG\_COMMON(name, func, suffix) PROCESS\_IP\_TARG##suffix(name, func)

#define PROCESS\_IP\_TARG\_Large(name, func) PROCESS\_IP\_TARG\_IMPL(name, func, Js::LargeLayout)

#define PROCESS\_IP\_TARG\_Medium(name, func) PROCESS\_IP\_TARG\_IMPL(name, func, Js::MediumLayout)

#define PROCESS\_IP\_TARG\_Small(name, func) PROCESS\_IP\_TARG\_IMPL(name, func, Js::SmallLayout)

namespace Js

{

extern const \_\_declspec(selectany) uint32 TypedArrayViewMask[] =

{

(uint32)~0 //TYPE\_INT8

, (uint32)~0 //TYPE\_UINT8

, (uint32)~1 //TYPE\_INT16

, (uint32)~1 //TYPE\_UINT16

, (uint32)~3 //TYPE\_INT32

, (uint32)~3 //TYPE\_UINT32

, (uint32)~3 //TYPE\_FLOAT32

, (uint32)~7 //TYPE\_FLOAT64

};

#ifndef TEMP\_DISABLE\_ASMJS

typedef void(InterpreterStackFrame::\*ArrFunc)(uint32, RegSlot);

const ArrFunc InterpreterStackFrame::StArrFunc[8] =

{

&InterpreterStackFrame::OP\_StArr<int8>,

&InterpreterStackFrame::OP\_StArr<uint8>,

&InterpreterStackFrame::OP\_StArr<int16>,

&InterpreterStackFrame::OP\_StArr<uint16>,

&InterpreterStackFrame::OP\_StArr<int32>,

&InterpreterStackFrame::OP\_StArr<uint32>,

&InterpreterStackFrame::OP\_StArr<float>,

&InterpreterStackFrame::OP\_StArr<double>,

};

const ArrFunc InterpreterStackFrame::LdArrFunc[8] =

{

&InterpreterStackFrame::OP\_LdArr<int8>,

&InterpreterStackFrame::OP\_LdArr<uint8>,

&InterpreterStackFrame::OP\_LdArr<int16>,

&InterpreterStackFrame::OP\_LdArr<uint16>,

&InterpreterStackFrame::OP\_LdArr<int32>,

&InterpreterStackFrame::OP\_LdArr<uint32>,

&InterpreterStackFrame::OP\_LdArr<float>,

&InterpreterStackFrame::OP\_LdArr<double>,

};

#endif

Var InterpreterStackFrame::InnerScopeFromRegSlot(RegSlot reg) const

{

return InnerScopeFromIndex(reg - m\_functionBody->FirstInnerScopeReg());

}

Var InterpreterStackFrame::InnerScopeFromIndex(uint32 index) const

{

if (index >= m\_functionBody->GetInnerScopeCount())

{

AssertMsg(false, "Illegal byte code: bad inner scope index");

Js::Throw::FatalInternalError();

}

Assert(this->innerScopeArray != nullptr);

return this->innerScopeArray[index];

}

void InterpreterStackFrame::SetInnerScopeFromIndex(uint32 index, Var scope)

{

if (index >= m\_functionBody->GetInnerScopeCount())

{

AssertMsg(false, "Illegal byte code: bad inner scope index");

Js::Throw::FatalInternalError();

}

Assert(this->innerScopeArray != nullptr);

this->innerScopeArray[index] = scope;

}

const int k\_stackFrameVarCount = (sizeof(InterpreterStackFrame) + sizeof(Var) - 1) / sizeof(Var);

InterpreterStackFrame::Setup::Setup(Js::ScriptFunction \* function, Js::Arguments& args, bool inlinee)

: function(function), inParams(args.Values), inSlotsCount(args.Info.Count), executeFunction(function->GetFunctionBody()), callFlags(args.Info.Flags), bailedOutOfInlinee(inlinee)

{

SetupInternal();

}

InterpreterStackFrame::Setup::Setup(Js::ScriptFunction \* function, Var \* inParams, int inSlotsCount, bool inlinee)

: function(function), inParams(inParams), inSlotsCount(inSlotsCount), executeFunction(function->GetFunctionBody()), callFlags(CallFlags\_None), bailedOutOfInlinee(inlinee)

{

SetupInternal();

}

void InterpreterStackFrame::Setup::SetupInternal()

{

if (this->function->GetHasInlineCaches() && Js::ScriptFunctionWithInlineCache::Is(this->function))

{

this->inlineCaches = Js::ScriptFunctionWithInlineCache::FromVar(this->function)->GetInlineCaches();

}

else

{

this->inlineCaches = this->executeFunction->GetInlineCaches();

}

this->inlineCacheCount = this->executeFunction->GetInlineCacheCount();

//

// Compute the amount of memory needed on the stack:

// - We compute this in 'Atoms' instead of 'bytes' to keep everything natural word aligned.

//

this->localCount = this->executeFunction->GetLocalsCount();

uint extraVarCount = 0;

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(this->executeFunction))

{

extraVarCount += (sizeof(ImplicitCallFlags) \* this->executeFunction->GetLoopCount() + sizeof(Var) - 1) / sizeof(Var);

}

#endif

this->varAllocCount = k\_stackFrameVarCount + localCount + this->executeFunction->GetOutParamsDepth() + extraVarCount + this->executeFunction->GetInnerScopeCount();

if (this->executeFunction->DoStackNestedFunc() && this->executeFunction->GetNestedCount() != 0)

{

// Track stack funcs...

this->varAllocCount += (sizeof(StackScriptFunction) \* this->executeFunction->GetNestedCount()) / sizeof(Var);

if (!this->bailedOutOfInlinee)

{

// Frame display (if environment depth is statically known)...

if (this->executeFunction->DoStackFrameDisplay())

{

uint16 envDepth = this->executeFunction->GetEnvDepth();

Assert(envDepth != (uint16)-1);

this->varAllocCount += sizeof(FrameDisplay) / sizeof(Var) + (envDepth + 1);

}

// ...and scope slots (if any)

if (this->executeFunction->DoStackScopeSlots())

{

uint32 scopeSlots = this->executeFunction->scopeSlotArraySize;

Assert(scopeSlots != 0);

this->varAllocCount += scopeSlots + Js::ScopeSlots::FirstSlotIndex;

}

}

}

}

InterpreterStackFrame \*

InterpreterStackFrame::Setup::InitializeAllocation(\_\_in\_ecount(varAllocCount) Var \* allocation, bool initParams, bool profileParams, Var loopHeaderArray, DWORD\_PTR stackAddr

#if DBG

, Var invalidStackVar

#endif

)

{

//

// Initialize the new InterpreterStackFrame instance on the program stack.

//

//This will fail if InterpreterStackFrame ever gets a non-empty ctor (you'll need to use

//placement\_new(allocation, InterpreterStackFrame) instead, though that will cause problems

//if the placement\_new is surrounded by a try/finally since this would mix C++/SEH exception

//handling.

\_\_analysis\_assume(varAllocCount >= k\_stackFrameVarCount + localCount);

InterpreterStackFrame\* newInstance = (InterpreterStackFrame\*)allocation;

newInstance->scriptContext = this->executeFunction->GetScriptContext();

newInstance->m\_inSlotsCount = this->inSlotsCount;

newInstance->m\_inParams = this->inParams;

newInstance->m\_callFlags = this->callFlags;

newInstance->m\_outParams = newInstance->m\_localSlots + localCount;

newInstance->m\_outSp = newInstance->m\_outParams;

newInstance->m\_arguments = NULL;

newInstance->function = this->function;

newInstance->m\_functionBody = this->executeFunction;

newInstance->inlineCaches = this->inlineCaches;

newInstance->inlineCacheCount = this->inlineCacheCount;

newInstance->currentLoopNum = LoopHeader::NoLoop;

newInstance->currentLoopCounter = 0;

newInstance->m\_flags = InterpreterStackFrameFlags\_None;

newInstance->closureInitDone = false;

#if ENABLE\_PROFILE\_INFO

newInstance->switchProfileMode = false;

newInstance->isAutoProfiling = false;

newInstance->switchProfileModeOnLoopEndNumber = 0u - 1;

#endif

newInstance->ehBailoutData = nullptr;

newInstance->nestedTryDepth = -1;

newInstance->nestedCatchDepth = -1;

newInstance->nestedFinallyDepth = -1;

newInstance->retOffset = 0;

newInstance->localFrameDisplay = nullptr;

newInstance->localClosure = nullptr;

newInstance->innerScopeArray = nullptr;

bool doInterruptProbe = newInstance->scriptContext->GetThreadContext()->DoInterruptProbe(this->executeFunction);

#if ENABLE\_NATIVE\_CODEGEN

bool doJITLoopBody =

!this->executeFunction->GetScriptContext()->GetConfig()->IsNoNative() &&

!(this->executeFunction->GetHasTry() && (PHASE\_OFF((Js::JITLoopBodyInTryCatchPhase), this->executeFunction) || this->executeFunction->GetHasFinally())) &&

(this->executeFunction->ForceJITLoopBody() || this->executeFunction->IsJitLoopBodyPhaseEnabled()) &&

!this->executeFunction->GetScriptContext()->IsInDebugMode();

#else

const bool doJITLoopBody = false;

#endif

// Pick a version of the LoopBodyStart OpCode handlers that is hardcoded to do loop body JIT and

// interrupt probes as needed.

if (doInterruptProbe)

{

#if ENABLE\_NATIVE\_CODEGEN

if (doJITLoopBody)

{

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<true, true>;

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<true, true>;

}

else

#endif

{

#if ENABLE\_PROFILE\_INFO

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<true, false>;

#endif

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<true, false>;

}

}

else

{

#if ENABLE\_NATIVE\_CODEGEN

if (doJITLoopBody)

{

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<false, true>;

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<false, true>;

}

else

#endif

{

#if ENABLE\_PROFILE\_INFO

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<false, false>;

#endif

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<false, false>;

}

}

newInstance->loopHeaderArray = loopHeaderArray;

newInstance->m\_stackAddress = stackAddr;

#if ENABLE\_PROFILE\_INFO

// the savedLoopImplicitCallFlags is allocated at the end of the out param array

newInstance->savedLoopImplicitCallFlags = nullptr;

#endif

char \* nextAllocBytes = (char \*)(newInstance->m\_outParams + this->executeFunction->GetOutParamsDepth());

if (this->executeFunction->GetInnerScopeCount())

{

newInstance->innerScopeArray = (Var\*)nextAllocBytes;

nextAllocBytes += this->executeFunction->GetInnerScopeCount() \* sizeof(Var);

}

if (this->executeFunction->DoStackNestedFunc() && this->executeFunction->GetNestedCount() != 0)

{

newInstance->InitializeStackFunctions((StackScriptFunction \*)nextAllocBytes);

nextAllocBytes = nextAllocBytes + sizeof(StackScriptFunction) \* this->executeFunction->GetNestedCount();

if (!this->bailedOutOfInlinee)

{

if (this->executeFunction->DoStackFrameDisplay())

{

uint16 envDepth = this->executeFunction->GetEnvDepth();

Assert(envDepth != (uint16)-1);

newInstance->localFrameDisplay = (FrameDisplay\*)nextAllocBytes;

nextAllocBytes += sizeof(FrameDisplay) + (envDepth + 1) \* sizeof(Var);

}

if (this->executeFunction->DoStackScopeSlots())

{

uint32 scopeSlots = this->executeFunction->scopeSlotArraySize;

Assert(scopeSlots != 0);

newInstance->localClosure = nextAllocBytes;

nextAllocBytes += (scopeSlots + ScopeSlots::FirstSlotIndex) \* sizeof(Var);

}

}

}

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(this->executeFunction))

{

/\*

\_\_analysis\_assume(varAllocCount == (k\_stackFrameVarCount + localCount + executeFunction->GetOutParamsDepth()

+ ((sizeof(ImplicitCallFlags) \* executeFunction->GetLoopCount() + sizeof(Var) - 1) / sizeof(Var))));

\*/

newInstance->savedLoopImplicitCallFlags = (ImplicitCallFlags \*)nextAllocBytes;

for (uint i = 0; i < this->executeFunction->GetLoopCount(); i++)

{

#pragma prefast(suppress:26015, "Above analysis assume doesn't work")

newInstance->savedLoopImplicitCallFlags[i] = ImplicitCall\_None;

}

}

#endif

#if DBG

if (CONFIG\_ISENABLED(InitializeInterpreterSlotsWithInvalidStackVarFlag))

{

// Fill the local slots with the invalid stack var so that we will crash deterministically if something goes wrong

for (uint i = 0; i < localCount; ++i)

{

newInstance->m\_localSlots[i] = invalidStackVar;

}

}

else

{

memset(newInstance->m\_localSlots, 0, sizeof(Js::Var) \* localCount);

}

#else

if (newInstance->scriptContext->IsInDebugMode())

{

// In the debug mode zero out the local slot, so this could prevent locals being uninitialized in the case of setNextStatement.

memset(newInstance->m\_localSlots, 0, sizeof(Js::Var) \* localCount);

}

// Zero out only the return slot. This is not a user local, so the byte code will not initialize

// it to "undefined". And it's not an expression temp, so, for instance, a jitted loop body may expect

// it to be valid on entry to the loop, where "valid" means either a var or null.

newInstance->SetNonVarReg(0, NULL);

#endif

// Initialize the low end of the local slots from the constant table.

// Skip the slot for the return value register.

this->executeFunction->InitConstantSlots(&newInstance->m\_localSlots[FunctionBody::FirstRegSlot]);

// Set local FD/SS pointers to null until after we've successfully probed the stack in the process loop.

// That way we avoid trying to box these structures before they've been initialized in the byte code.

if (this->executeFunction->DoStackFrameDisplay())

{

newInstance->SetNonVarReg(executeFunction->GetLocalFrameDisplayReg(), nullptr);

}

if (this->executeFunction->DoStackScopeSlots())

{

Assert(!executeFunction->HasScopeObject());

newInstance->SetNonVarReg(executeFunction->GetLocalClosureReg(), nullptr);

}

Var \*prestDest = &newInstance->m\_localSlots[this->executeFunction->GetConstantCount()];

if (initParams)

{

#if ENABLE\_PROFILE\_INFO

Assert(!this->executeFunction->NeedEnsureDynamicProfileInfo());

#endif

if (profileParams)

{

#if ENABLE\_PROFILE\_INFO

Assert(this->executeFunction->HasExecutionDynamicProfileInfo());

#endif

FunctionBody\* functionBody = this->executeFunction;

InitializeParams(newInstance, [functionBody](Var param, ArgSlot index)

{

#if ENABLE\_PROFILE\_INFO

functionBody->GetDynamicProfileInfo()->RecordParameterInfo(functionBody, index - 1, param);

#endif

}, &prestDest);

}

else

{

InitializeParams(newInstance, [](Var param, ArgSlot index) {}, &prestDest);

}

}

if (this->executeFunction->GetHasRestParameter())

{

InitializeRestParam(newInstance, prestDest);

}

Js::RegSlot envReg = executeFunction->GetEnvReg();

if (envReg != Js::Constants::NoRegister && envReg < executeFunction->GetConstantCount())

{

Assert(this->executeFunction->GetThisRegForEventHandler() == Constants::NoRegister);

// The correct FD (possibly distinct from the one on the function) is passed in the constant table.

this->function->SetEnvironment((Js::FrameDisplay\*)newInstance->GetNonVarReg(envReg));

}

return newInstance;

}

template <class Fn>

void InterpreterStackFrame::Setup::InitializeParams(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest)

{

ArgSlot requiredInParamCount = executeFunction->GetInParamsCount();

Assert(requiredInParamCount > 1);

if (this->inSlotsCount >= requiredInParamCount)

{

Var \* pArg = &newInstance->m\_localSlots[executeFunction->GetConstantCount()];

Var \* paGivenSrc = this->inParams + 1;

ArgSlot paramIndex = 1;

do

{

Var src = \*paGivenSrc++;

callback(src, paramIndex);

\*pArg++ = src;

paramIndex++;

}

while (paramIndex < requiredInParamCount);

\*pprestDest = pArg;

}

else

{

InitializeParamsAndUndef(newInstance, callback, pprestDest);

}

}

template <class Fn>

void InterpreterStackFrame::Setup::InitializeParamsAndUndef(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest)

{

Var \* pArg = &newInstance->m\_localSlots[executeFunction->GetConstantCount()];

Var \* paGivenSrc = this->inParams + 1;

ArgSlot requiredInParamCount = executeFunction->GetInParamsCount();

ArgSlot paramIndex = 1;

while (paramIndex < this->inSlotsCount)

{

Var src = \*paGivenSrc++;

callback(src, paramIndex);

\*pArg++ = src;

paramIndex++;

}

Var varUndef = executeFunction->GetScriptContext()->GetLibrary()->GetUndefined();

do

{

callback(varUndef, paramIndex);

\*pArg++ = varUndef;

paramIndex++;

}

while (paramIndex < requiredInParamCount);

\*pprestDest = pArg;

}

void InterpreterStackFrame::Setup::InitializeRestParam(InterpreterStackFrame \* newInstance, Var \*dest)

{

Var \*src = this->inParams + executeFunction->GetInParamsCount();

if (this->inSlotsCount > executeFunction->GetInParamsCount())

{

// Create the rest array and copy the args directly into the contiguous head segment.

int excess = this->inSlotsCount - executeFunction->GetInParamsCount();

\*dest = JavascriptArray::OP\_NewScArray(excess, executeFunction->GetScriptContext());

JavascriptArray \*array = static\_cast<JavascriptArray \*>(\*dest);

Var \*elements = ((SparseArraySegment<Var>\*)array->GetHead())->elements;

js\_memcpy\_s(elements, excess \* sizeof(Var), src, excess \* sizeof(Var));

}

else

{

// Rest is an empty array when there are no excess parameters.

\*dest = JavascriptArray::OP\_NewScArray(0, executeFunction->GetScriptContext());

}

}

FrameDisplay \* InterpreterStackFrame::GetEnvForEvalCode()

{

FrameDisplay \*pScope;

if (m\_functionBody->GetIsStrictMode() && m\_functionBody->GetIsGlobalFunc())

{

pScope = this->GetLocalFrameDisplay();

}

else

{

pScope = (FrameDisplay\*)this->LdEnv();

}

return pScope;

}

void InterpreterStackFrame::InitializeClosures()

{

FunctionBody \*executeFunction = this->function->GetFunctionBody();

Var environment;

RegSlot thisRegForEventHandler = executeFunction->GetThisRegForEventHandler();

if (thisRegForEventHandler != Constants::NoRegister)

{

Var varThis = OP\_ArgIn0();

SetReg(thisRegForEventHandler, varThis);

environment = JavascriptOperators::OP\_LdHandlerScope(varThis, GetScriptContext());

this->SetEnv((FrameDisplay\*)environment);

}

else

{

environment = this->LdEnv();

}

RegSlot closureReg = executeFunction->GetLocalClosureReg();

if (closureReg != Js::Constants::NoRegister)

{

Assert(closureReg >= executeFunction->GetConstantCount());

if (executeFunction->HasScopeObject())

{

Js::RegSlot funcExprScopeReg = executeFunction->GetFuncExprScopeReg();

if (funcExprScopeReg != Constants::NoRegister)

{

// t0 = NewPseudoScope

// t1 = LdFrameDisplay t0 env

Var funcExprScope = JavascriptOperators::OP\_NewPseudoScope(GetScriptContext());

SetReg(funcExprScopeReg, funcExprScope);

environment = OP\_LdFrameDisplay(funcExprScope, environment, GetScriptContext());

}

this->NewScopeObject();

}

else

{

this->NewScopeSlots();

}

this->SetNonVarReg(closureReg, nullptr);

}

Js::RegSlot frameDisplayReg = executeFunction->GetLocalFrameDisplayReg();

if (frameDisplayReg != Js::Constants::NoRegister && closureReg != Js::Constants::NoRegister)

{

Assert(frameDisplayReg >= executeFunction->GetConstantCount());

void \*argHead = this->GetLocalClosure();

this->SetLocalFrameDisplay(this->NewFrameDisplay(argHead, environment));

this->SetNonVarReg(frameDisplayReg, nullptr);

}

this->closureInitDone = true;

}

#ifdef \_M\_IX86

#ifndef TEMP\_DISABLE\_ASMJS

int InterpreterStackFrame::GetAsmJsArgSize(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* func = stack->functionObject;

AsmJsFunctionInfo\* asmInfo = func->GetFunctionBody()->GetAsmJsFunctionInfo();

uint argSize = (uint)(asmInfo->GetArgByteSize());

argSize = ::Math::Align<int32>(argSize, 8);

// 2 \* sizeof(Var) is for functionObject, and another push that DynamicInterpreterThunk does

return argSize + 2 \* sizeof(Var);

}

int InterpreterStackFrame::GetDynamicRetType(AsmJsCallStackLayout\* stack)

{

return GetRetType(stack->functionObject);

}

int InterpreterStackFrame::GetRetType(JavascriptFunction\* func)

{

AsmJsFunctionInfo\* asmInfo = func->GetFunctionBody()->GetAsmJsFunctionInfo();

return asmInfo->GetReturnType().which();

}

DWORD InterpreterStackFrame::GetAsmIntDbValOffSet(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* func = stack->functionObject;

ScriptContext\* scriptContext = func->GetScriptContext();

return (DWORD)scriptContext + ScriptContext::GetAsmIntDbValOffset();

}

DWORD InterpreterStackFrame::GetAsmSimdValOffSet(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* func = stack->functionObject;

ScriptContext\* scriptContext = func->GetScriptContext();

return (DWORD)scriptContext + ScriptContext::GetAsmSimdValOffset();

}

#ifdef ASMJS\_PLAT

/\*

AsmInterpreterThunk

-------------------

This is the entrypoint for all Asm Interpreter calls (external and internal)

TODO - Make this a dynamic Interpreter thunk to support ETW

Functionality:

1) Prolog

2) call AsmInterpreter passing the function object

3) Get The return type

4) Check for Double or Float return type

5) If true then retrieve the value stored at a constant offset from the ScriptContext

6) Get Argument Size for callee cleanup

7) EpiLog

a) Retrieve the frame pointer

b) Store the return address in register (edx)

c) Clean the arguments based on the arguments size

d) push the return address back into the stack

\*/

\_\_declspec(naked)

void InterpreterStackFrame::InterpreterAsmThunk(AsmJsCallStackLayout\* layout)

{

enum {

Void = AsmJsRetType::Void,

Signed = AsmJsRetType::Signed,

Float = AsmJsRetType::Float,

Double = AsmJsRetType::Double,

Int32x4 = AsmJsRetType::Int32x4,

Float32x4 = AsmJsRetType::Float32x4,

Float64x2 = AsmJsRetType::Float64x2

};

//Prolog

\_\_asm

{

//Prologue

push ebp;

mov ebp, esp;

push layout; // push stack layout

call InterpreterStackFrame::AsmJsInterpreter;

push eax; // push the return value into the stack

push layout;

call InterpreterStackFrame::GetDynamicRetType;

cmp eax, Void;

je end;

cmp eax, Signed;

je end;

cmp eax, Float;

jne skipFloat;

// float

push layout;

call InterpreterStackFrame::GetAsmIntDbValOffSet;

cvtsd2ss xmm0, [eax];

jmp end;

skipFloat:

cmp eax, Double;

jne skipDouble;

// double

push layout;

call InterpreterStackFrame::GetAsmIntDbValOffSet;

movsd xmm0, [eax];

jmp end;

skipDouble:

// simd value

push layout;

call InterpreterStackFrame::GetAsmSimdValOffSet;

movups xmm0, [eax];

end:

push layout;

call InterpreterStackFrame::GetAsmJsArgSize;

mov ecx, eax;

pop eax; // pop the return value from AsmJsInterpreter to eax

// Epilog, callee cleanup

mov esp, ebp;

pop ebp;

// we need to move stack around in order to do callee cleanup

// unfortunately, we don't really have enough registers to do this cleanly

//

// we are rearranging the stack from this:

// 0x14 caller push scriptArg1

// 0x10 caller push functionObject

// 0x0C DynamicInterpreterThunk return address

// 0x08 DynamicInterpreterThunk push ebp

// 0x04 DynamicInterpreterThunk push functionObject

// 0x00 InterpreterAsmThunk return address <- stack pointer

// to this:

// 0x14 DynamicInterpreterThunk return address

// 0x10 DynamicInterpreterThunk push ebp

// 0x0C InterpreterAsmThunk return address <- stack pointer

push eax; // save eax

mov eax, esp;

add eax, ecx;

add eax, 0xC; // eax will be our stack destination. we need to move backwards because memory might overlap

mov edx, [esp+0x10];

mov [eax], edx; // move the dynamic interpreter thunk return location

sub eax, 0x4;

mov edx, [esp+0xC];

mov [eax], edx; // move the dynamic interpreter thunk "push ebp" location

// skip "push functionObject"

sub eax, 0x4;

mov edx, [esp+0x4];

mov [eax], edx; // move the return location

pop eax;

add esp, ecx; // cleanup arguments

ret;

}

}

#endif

#endif

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

#ifdef \_M\_IX86

\_\_declspec(naked)

Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

\_\_asm

{

push ebp

mov ebp, esp

push [esp+8] // push function object

call InterpreterStackFrame::EnsureDynamicInterpreterThunk;

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

push eax

mov ecx, eax

call[\_\_guard\_check\_icall\_fptr]

pop eax

#endif

pop ebp

jmp eax

}

}

#endif

#endif

#if ENABLE\_PROFILE\_INFO

JavascriptMethod InterpreterStackFrame::EnsureDynamicInterpreterThunk(Js::ScriptFunction \* function)

{

#if DYNAMIC\_INTERPRETER\_THUNK

Assert(function);

Js::FunctionBody \*functionBody = function->GetFunctionBody();

JavascriptMethod entrypoint = functionBody->EnsureDynamicInterpreterThunk(function->GetFunctionEntryPointInfo());

Assert(!IsDelayDynamicInterpreterThunk(functionBody->GetDirectEntryPoint(function->GetEntryPointInfo())));

if (function->GetEntryPoint() == InterpreterStackFrame::DelayDynamicInterpreterThunk)

{

// If we are not profiling, or the function object is not cross site, this is the direct entry point.

// Change the entry point on the object

Assert(functionBody->GetDirectEntryPoint(function->GetEntryPointInfo()) == entrypoint);

function->ChangeEntryPoint(function->GetEntryPointInfo(), entrypoint);

}

// Return the original entry point to be called

return entrypoint;

#else

return function->GetEntryPoint();

#endif

}

#endif

bool InterpreterStackFrame::IsDelayDynamicInterpreterThunk(void \* entryPoint)

{

return

#if DYNAMIC\_INTERPRETER\_THUNK

#if \_M\_X64

entryPoint == InterpreterStackFrame::AsmJsDelayDynamicInterpreterThunk ||

#endif

entryPoint == InterpreterStackFrame::DelayDynamicInterpreterThunk;

#else

false;

#endif

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

\_\_declspec(thread) int InterpreterThunkStackCountTracker::s\_count = 0;

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

Var InterpreterStackFrame::InterpreterThunk(JavascriptCallStackLayout\* layout)

{

Js::ScriptFunction \* function = Js::ScriptFunction::FromVar(layout->functionObject);

Js::ArgumentReader args(&layout->callInfo, layout->args);

void\* localReturnAddress = \_ReturnAddress();

void\* localAddressOfReturnAddress = \_AddressOfReturnAddress();

return InterpreterHelper(function, args, localReturnAddress, localAddressOfReturnAddress);

}

#else

#pragma optimize("", off)

Var InterpreterStackFrame::InterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

ARGUMENTS(args, callInfo);

void\* localReturnAddress = \_ReturnAddress();

void\* localAddressOfReturnAddress = \_AddressOfReturnAddress();

Assert(ScriptFunction::Is(function));

return InterpreterHelper(ScriptFunction::FromVar(function), args, localReturnAddress, localAddressOfReturnAddress);

}

#pragma optimize("", on)

#endif

Var InterpreterStackFrame::InterpreterHelper(ScriptFunction\* function, ArgumentReader args, void\* returnAddress, void\* addressOfReturnAddress, const bool isAsmJs)

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// Support for simulating partially initialized interpreter stack frame.

InterpreterThunkStackCountTracker tracker;

if (CONFIG\_ISENABLED(InjectPartiallyInitializedInterpreterFrameErrorFlag) &&

CONFIG\_FLAG(InjectPartiallyInitializedInterpreterFrameError) == InterpreterThunkStackCountTracker::GetCount())

{

switch (CONFIG\_FLAG(InjectPartiallyInitializedInterpreterFrameErrorType))

{

case 0:

DebugBreak();

break;

case 1:

Js::JavascriptError::MapAndThrowError(function->GetScriptContext(), VBSERR\_InternalError);

break;

default:

DebugBreak();

}

}

#endif

ScriptContext\* functionScriptContext = function->GetScriptContext();

ThreadContext \* threadContext = functionScriptContext->GetThreadContext();

Assert(!threadContext->IsDisableImplicitException());

functionScriptContext->VerifyAlive(!function->IsExternal());

Assert(threadContext->IsScriptActive());

Assert(threadContext->IsInScript());

FunctionBody\* executeFunction = JavascriptFunction::FromVar(function)->GetFunctionBody();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (!isAsmJs && executeFunction->IsByteCodeDebugMode() != functionScriptContext->IsInDebugMode()) // debug mode mismatch

{

if (executeFunction->GetUtf8SourceInfo()->GetIsLibraryCode())

{

Assert(!executeFunction->IsByteCodeDebugMode()); // Library script byteCode is never in debug mode

}

else

{

Throw::FatalInternalError();

}

}

#endif

if (executeFunction->interpretedCount == 0)

{

executeFunction->TraceInterpreterExecutionMode();

}

class AutoRestore

{

private:

ThreadContext \*const threadContext;

const uint8 savedLoopDepth;

public:

AutoRestore(ThreadContext \*const threadContext, FunctionBody \*const executeFunction)

: threadContext(threadContext),

savedLoopDepth(threadContext->LoopDepth())

{

if (savedLoopDepth != 0 && !executeFunction->GetIsAsmJsFunction())

{

executeFunction->SetWasCalledFromLoop();

}

}

~AutoRestore()

{

threadContext->SetLoopDepth(savedLoopDepth);

}

} autoRestore(threadContext, executeFunction);

#if ENABLE\_PROFILE\_INFO

DynamicProfileInfo \* dynamicProfileInfo = nullptr;

const bool doProfile = executeFunction->GetInterpreterExecutionMode(false) == ExecutionMode::ProfilingInterpreter ||

functionScriptContext->IsInDebugMode() && DynamicProfileInfo::IsEnabled(executeFunction);

if (doProfile)

{

#if !DYNAMIC\_INTERPRETER\_THUNK

executeFunction->EnsureDynamicProfileInfo();

#endif

dynamicProfileInfo = executeFunction->GetDynamicProfileInfo();

threadContext->ClearImplicitCallFlags();

}

#else

const bool doProfile = false;

#endif

executeFunction->interpretedCount++;

#ifdef BGJIT\_STATS

functionScriptContext->interpretedCount++;

functionScriptContext->maxFuncInterpret = max(functionScriptContext->maxFuncInterpret, executeFunction->interpretedCount);

#endif

AssertMsg(!executeFunction->IsDeferredParseFunction(),

"Non-intrinsic functions must provide byte-code to execute");

bool fReleaseAlloc = false;

InterpreterStackFrame\* newInstance = nullptr;

Var\* allocation = nullptr;

if (!isAsmJs && executeFunction->IsGenerator())

{

// If the FunctionBody is a generator then this call is being made by one of the three

// generator resuming methods: next(), throw(), or return(). They all pass the generator

// object as the first of two arguments. The real user arguments are obtained from the

// generator object. The second argument is the ResumeYieldData which is only needed

// when resuming a generator and so it only used here if a frame already exists on the

// generator object.

AssertMsg(args.Info.Count == 2, "Generator ScriptFunctions should only be invoked by generator APIs with the pair of arguments they pass in -- the generator object and a ResumeYieldData pointer");

JavascriptGenerator\* generator = JavascriptGenerator::FromVar(args[0]);

newInstance = generator->GetFrame();

if (newInstance != nullptr)

{

ResumeYieldData\* resumeYieldData = static\_cast<ResumeYieldData\*>(args[1]);

newInstance->SetNonVarReg(executeFunction->GetYieldRegister(), resumeYieldData);

// The debugger relies on comparing stack addresses of frames to decide when a step\_out is complete so

// give the InterpreterStackFrame a legit enough stack address to make this comparison work.

newInstance->m\_stackAddress = reinterpret\_cast<DWORD\_PTR>(&generator);

}

else

{

//

// Allocate a new InterpreterStackFrame instance on the recycler heap.

// It will live with the JavascriptGenerator object.

//

Arguments generatorArgs = generator->GetArguments();

InterpreterStackFrame::Setup setup(function, generatorArgs);

size\_t varAllocCount = setup.GetAllocationVarCount();

size\_t varSizeInBytes = varAllocCount \* sizeof(Var);

DWORD\_PTR stackAddr = reinterpret\_cast<DWORD\_PTR>(&generator); // as mentioned above, use any stack address from this frame to ensure correct debugging functionality

Var loopHeaderArray = executeFunction->GetHasAllocatedLoopHeaders() ? executeFunction->GetLoopHeaderArrayPtr() : nullptr;

allocation = RecyclerNewPlus(functionScriptContext->GetRecycler(), varSizeInBytes, Var);

AnalysisAssert(allocation);

#if DBG

// Allocate invalidVar on GC instead of stack since this InterpreterStackFrame will out live the current real frame

Js::RecyclableObject\* invalidVar = (Js::RecyclableObject\*)RecyclerNewPlusLeaf(functionScriptContext->GetRecycler(), sizeof(Js::RecyclableObject), Var);

AnalysisAssert(invalidVar);

memset(invalidVar, 0xFE, sizeof(Js::RecyclableObject));

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns(), doProfile, loopHeaderArray, stackAddr, invalidVar);

#else

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns(), doProfile, loopHeaderArray, stackAddr);

#endif

newInstance->m\_reader.Create(executeFunction);

generator->SetFrame(newInstance);

}

}

else

{

InterpreterStackFrame::Setup setup(function, args);

size\_t varAllocCount = setup.GetAllocationVarCount();

size\_t varSizeInBytes = varAllocCount \* sizeof(Var);

//

// Allocate a new InterpreterStackFrame instance on the interpreter's virtual stack.

//

DWORD\_PTR stackAddr;

// If the locals area exceeds a certain limit, allocate it from a private arena rather than

// this frame. The current limit is based on an old assert on the number of locals we would allow here.

if (varAllocCount > InterpreterStackFrame::LocalsThreshold)

{

ArenaAllocator \*tmpAlloc = nullptr;

fReleaseAlloc = functionScriptContext->EnsureInterpreterArena(&tmpAlloc);

allocation = (Var\*)tmpAlloc->Alloc(varSizeInBytes);

stackAddr = reinterpret\_cast<DWORD\_PTR>(&allocation); // use a stack address so the debugger stepping logic works (step-out, for example, compares stack depths to determine when to complete the step)

}

else

{

PROBE\_STACK\_PARTIAL\_INITIALIZED\_INTERPRETER\_FRAME(functionScriptContext, Js::Constants::MinStackInterpreter + varSizeInBytes);

allocation = (Var\*)\_alloca(varSizeInBytes);

stackAddr = reinterpret\_cast<DWORD\_PTR>(allocation);

}

/\*

\* If the function has any loop headers, we allocate an array for the loop headers wrappers, and

\* reference the wrappers in the array. We then push the pointer to the array onto the stack itself.

\* We do this so that while the function is being interpreted, we don't want the jitted loop

\* bodies to be collected, even if the loop body isn't being executed. The loop body will

\* get collected when the function has been JITted, and when the function exits the interpreter.

\* The array contains nulls if the loop body isn't jitted (or hasn't been jitted yet) but

\* it's cheaper to just copy them all into the recycler array rather than just the ones that

\* have been jitted.

\*/

Var loopHeaderArray = nullptr;

if (executeFunction->GetHasAllocatedLoopHeaders())

{

// Loop header array is recycler allocated, so we push it on the stack

// When we scan the stack, we'll recognize it as a recycler allocated

// object, and mark it's contents and keep the individual loop header

// wrappers alive

loopHeaderArray = executeFunction->GetLoopHeaderArrayPtr();

}

#if DBG

Js::RecyclableObject \* invalidStackVar = (Js::RecyclableObject\*)\_alloca(sizeof(Js::RecyclableObject));

memset(invalidStackVar, 0xFE, sizeof(Js::RecyclableObject));

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns() && !isAsmJs, doProfile, loopHeaderArray, stackAddr, invalidStackVar);

#else

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns() && !isAsmJs, doProfile, loopHeaderArray, stackAddr);

#endif

newInstance->m\_reader.Create(executeFunction);

}

//

// Execute the function's byte-code, returning the return-value:

// - Mark that the function is current executing and may not be modified.

//

executeFunction->BeginExecution();

Var aReturn = nullptr;

{

if (!isAsmJs && functionScriptContext->IsInDebugMode())

{

#if DYNAMIC\_INTERPRETER\_THUNK

PushPopFrameHelper pushPopFrameHelper(newInstance, returnAddress, addressOfReturnAddress);

aReturn = newInstance->DebugProcess();

#else

aReturn = newInstance->DebugProcessThunk(\_ReturnAddress(), \_AddressOfReturnAddress());

#endif

}

else

{

#if DYNAMIC\_INTERPRETER\_THUNK

PushPopFrameHelper pushPopFrameHelper(newInstance, returnAddress, addressOfReturnAddress);

aReturn = newInstance->Process();

#else

aReturn = newInstance->ProcessThunk(\_ReturnAddress(), \_AddressOfReturnAddress());

#endif

}

}

executeFunction->EndExecution();

if (fReleaseAlloc)

{

functionScriptContext->ReleaseInterpreterArena();

}

#if ENABLE\_PROFILE\_INFO

if (doProfile)

{

dynamicProfileInfo->RecordImplicitCallFlags(threadContext->GetImplicitCallFlags());

}

#endif

if (isAsmJs)

{

return newInstance;

}

return aReturn;

}

#ifndef TEMP\_DISABLE\_ASMJS

#if \_M\_IX86

int InterpreterStackFrame::AsmJsInterpreter(AsmJsCallStackLayout\* stack)

{

ScriptFunction \* function = (ScriptFunction\*)stack->functionObject;

Var\* paramsAddr = stack->args;

int flags = CallFlags\_Value;

ArgSlot nbArgs = UInt16Math::Add(function->GetFunctionBody()->GetAsmJsFunctionInfo()->GetArgCount(), 1);

CallInfo callInfo((CallFlags)flags, nbArgs);

ArgumentReader args(&callInfo, paramsAddr);

void\* returnAddress = \_ReturnAddress();

void\* addressOfReturnAddress = \_AddressOfReturnAddress();

#if ENABLE\_PROFILE\_INFO

function->GetFunctionBody()->EnsureDynamicProfileInfo();

#endif

InterpreterStackFrame\* newInstance = (InterpreterStackFrame\*)InterpreterHelper(function, args, returnAddress, addressOfReturnAddress, true);

//Handle return value

AsmJsRetType::Which retType = (AsmJsRetType::Which) GetRetType(function);

int retVal = 0;

switch (retType)

{

case AsmJsRetType::Int32x4:

case AsmJsRetType::Float32x4:

case AsmJsRetType::Float64x2:

if (function->GetScriptContext()->GetConfig()->IsSimdjsEnabled())

{

function->GetScriptContext()->retAsmSimdVal = newInstance->m\_localSimdSlots[0];

break;

}

Assert(UNREACHED);

// double return

case AsmJsRetType::Double:

function->GetScriptContext()->retAsmIntDbVal = newInstance->m\_localDoubleSlots[0];

break;

// float return

case AsmJsRetType::Float:

function->GetScriptContext()->retAsmIntDbVal = (double)newInstance->m\_localFloatSlots[0];

break;

// signed or void return

case AsmJsRetType::Signed:

case AsmJsRetType::Void:

retVal = newInstance->m\_localIntSlots[0];

break;

default:

Assume(false);

}

return retVal;

}

#elif \_M\_X64

typedef double(\*AsmJsInterpreterDoubleEP)(AsmJsCallStackLayout\*, void \*);

typedef float(\*AsmJsInterpreterFloatEP)(AsmJsCallStackLayout\*, void \*);

typedef int(\*AsmJsInterpreterIntEP)(AsmJsCallStackLayout\*, void \*);

void \* InterpreterStackFrame::GetAsmJsInterpreterEntryPoint(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* function = stack->functionObject;

void \* entryPoint = nullptr;

switch (function->GetFunctionBody()->GetAsmJsFunctionInfo()->GetReturnType().which())

{

case Js::AsmJsRetType::Double:

{

entryPoint = (AsmJsInterpreterDoubleEP)Js::InterpreterStackFrame::AsmJsInterpreter < double > ;

break;

}

case Js::AsmJsRetType::Float:

{

entryPoint = (AsmJsInterpreterFloatEP)Js::InterpreterStackFrame::AsmJsInterpreter < float > ;

break;

}

case Js::AsmJsRetType::Signed:

case Js::AsmJsRetType::Void:

{

entryPoint = (AsmJsInterpreterIntEP)Js::InterpreterStackFrame::AsmJsInterpreter < int > ;

break;

}

case Js::AsmJsRetType::Int32x4:

case Js::AsmJsRetType::Float32x4:

case Js::AsmJsRetType::Float64x2:

{

entryPoint = Js::InterpreterStackFrame::AsmJsInterpreterSimdJs;

break;

}

default:

Assume(UNREACHED);

}

return entryPoint;

}

template<>

int InterpreterStackFrame::GetAsmJsRetVal<int>(InterpreterStackFrame\* instance)

{

return instance->m\_localIntSlots[0];

}

template<>

double InterpreterStackFrame::GetAsmJsRetVal<double>(InterpreterStackFrame\* instance)

{

return instance->m\_localDoubleSlots[0];

}

template<>

float InterpreterStackFrame::GetAsmJsRetVal<float>(InterpreterStackFrame\* instance)

{

return instance->m\_localFloatSlots[0];

}

template<>

X86SIMDValue InterpreterStackFrame::GetAsmJsRetVal<X86SIMDValue>(InterpreterStackFrame\* instance)

{

return X86SIMDValue::ToX86SIMDValue(instance->m\_localSimdSlots[0]);

}

template<typename T>

T InterpreterStackFrame::AsmJsInterpreter(AsmJsCallStackLayout\* layout)

{

Js::ScriptFunction \* function = Js::ScriptFunction::FromVar(layout->functionObject);

int flags = CallFlags\_Value;

ArgSlot nbArgs = UInt16Math::Add(function->GetFunctionBody()->GetAsmJsFunctionInfo()->GetArgCount(), 1);

CallInfo callInfo((CallFlags)flags, nbArgs);

ArgumentReader args(&callInfo, (Var\*)layout->args);

void\* returnAddress = \_ReturnAddress();

void\* addressOfReturnAddress = \_AddressOfReturnAddress();

function->GetFunctionBody()->EnsureDynamicProfileInfo();

InterpreterStackFrame\* newInstance = (InterpreterStackFrame\*)InterpreterHelper(function, args, returnAddress, addressOfReturnAddress, true);

return GetAsmJsRetVal<T>(newInstance);

}

\_\_m128 InterpreterStackFrame::AsmJsInterpreterSimdJs(AsmJsCallStackLayout\* layout)

{

return AsmJsInterpreter<X86SIMDValue>(layout).m128\_value;

}

#endif

#endif

///----------------------------------------------------------------------------

///

/// InterpreterStackFrame::SetOut()

///

/// SetOut() change the Var value stored in the specified "out parameter"

/// register.

///

///----------------------------------------------------------------------------

inline void InterpreterStackFrame::SetOut(ArgSlot outRegisterID, Var aValue)

{

//

// The "out" parameter slots are located at the end of the local register range, counting

// forwards. This results in the "in" parameter slots being disjoint from the rest of the

// InterpreterStackFrame.

// ..., InterpreterStackFrame A, Locals A[], ..., Out A:0, Out A:1, Out A:2, ...

// | In B:0, In B:1, ..., InterpreterStackFrame B, Locals B[], ...

// (current 'this') |

// (new 'this' after call)

//

Assert(m\_outParams + outRegisterID < m\_outSp);

m\_outParams[outRegisterID] = aValue;

}

inline void InterpreterStackFrame::SetOut(ArgSlot\_OneByte outRegisterID, Var aValue)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

m\_outParams[outRegisterID] = aValue;

}

inline void InterpreterStackFrame::OP\_SetOutAsmDb( RegSlot outRegisterID, double val )

{

Assert( m\_outParams + outRegisterID < m\_outSp );

m\_outParams[outRegisterID] = JavascriptNumber::New( val, scriptContext );

}

inline void InterpreterStackFrame::OP\_SetOutAsmInt( RegSlot outRegisterID, int val )

{

Assert( m\_outParams + outRegisterID < m\_outSp );

m\_outParams[outRegisterID] = JavascriptNumber::ToVar( val, scriptContext );

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmFlt(RegSlot outRegisterID, float val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(float\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmInt(RegSlot outRegisterID, int val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(int\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmDb(RegSlot outRegisterID, double val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(double\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmSimd(RegSlot outRegisterID, AsmJsSIMDValue val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(AsmJsSIMDValue\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::PushOut(Var aValue)

{

\*m\_outSp++ = aValue;

}

inline void InterpreterStackFrame::PopOut(ArgSlot argCount)

{

m\_outSp -= (argCount+1);

m\_outParams = (Var\*)\*m\_outSp;

AssertMsg(m\_localSlots + this->m\_functionBody->GetLocalsCount() <= m\_outSp &&

m\_outSp < (m\_localSlots + this->m\_functionBody->GetLocalsCount() + this->m\_functionBody->GetOutParamsDepth()),

"out args Stack pointer not in range after Pop");

}

void InterpreterStackFrame::ResetOut()

{

//

// Reset the m\_outParams and m\_outSp

//

m\_outParams = m\_localSlots + this->m\_functionBody->GetLocalsCount();

m\_outSp = m\_outParams;

}

\_\_declspec(noinline)

Var InterpreterStackFrame::DebugProcessThunk(void\* returnAddress, void\* addressOfReturnAddress)

{

PushPopFrameHelper pushPopFrameHelper(this, returnAddress, addressOfReturnAddress);

return this->DebugProcess();

}

//

// Under debug mode allow the exception to be swallowed and execution to continue

// if the debugger has specified that behavior.

//

Var InterpreterStackFrame::DebugProcess()

{

Assert(this->returnAddress != nullptr);

while (true)

{

JavascriptExceptionObject \*exception = nullptr;

try

{

return this->ProcessWithDebugging();

}

catch (JavascriptExceptionObject \*exception\_)

{

Assert(exception\_);

exception = exception\_;

}

if (exception)

{

bool skipException = false;

if (exception != scriptContext->GetThreadContext()->GetPendingSOErrorObject()

&& exception != scriptContext->GetThreadContext()->GetPendingOOMErrorObject())

{

skipException = exception->IsDebuggerSkip();

}

if (skipException)

{

// If we are going to swallow the exception then advance to the beginning of the next user statement

if (exception->IsIgnoreAdvanceToNextStatement()

|| this->scriptContext->GetDebugContext()->GetProbeContainer()->AdvanceToNextUserStatement(this->m\_functionBody, &this->m\_reader))

{

// We must fix up the return value to at least be undefined:

this->SetReg((RegSlot)0,this->scriptContext->GetLibrary()->GetUndefined());

// If we recover from the exception, there may be a chance the out pointers in the InterpreterStackframe are not in a proper state.

// Reset them to correct the stack.

ResetOut();

// If we can successfully advance then continuing processing

continue;

}

}

exception = exception->CloneIfStaticExceptionObject(scriptContext);

throw exception;

}

}

}

template<>

OpCode InterpreterStackFrame::ReadByteOp<OpCode>(const byte \*& ip

#if DBG\_DUMP

, bool isExtended /\*= false\*/

#endif

)

{

#if DBG || DBG\_DUMP

//

// For debugging byte-code, store the current offset before the instruction is read:

// - We convert this to "void \*" to encourage the debugger to always display in hex,

// which matches the displayed offsets used by ByteCodeDumper.

//

this->DEBUG\_currentByteOffset = (void \*) m\_reader.GetCurrentOffset();

#endif

OpCode op = ByteCodeReader::ReadByteOp(ip);

#if DBG\_DUMP

this->scriptContext->byteCodeHistogram[(int)op]++;

if (PHASE\_TRACE(Js::InterpreterPhase, this->m\_functionBody))

{

Output::Print(L"%d.%d:Executing %s at offset 0x%X\n", this->m\_functionBody->GetSourceContextId(), this->m\_functionBody->GetLocalFunctionId(), Js::OpCodeUtil::GetOpCodeName((Js::OpCode)(op+((int)isExtended<<8))), DEBUG\_currentByteOffset);

}

#endif

return op;

}

#ifndef TEMP\_DISABLE\_ASMJS

template<>

OpCodeAsmJs InterpreterStackFrame::ReadByteOp<OpCodeAsmJs>(const byte \*& ip

#if DBG\_DUMP

, bool isExtended /\*= false\*/

#endif

)

{

#if DBG || DBG\_DUMP

//

// For debugging byte-code, store the current offset before the instruction is read:

// - We convert this to "void \*" to encourage the debugger to always display in hex,

// which matches the displayed offsets used by ByteCodeDumper.

//

this->DEBUG\_currentByteOffset = (void \*) m\_reader.GetCurrentOffset();

#endif

OpCodeAsmJs op = (OpCodeAsmJs)ByteCodeReader::ReadByteOp(ip);

#if DBG\_DUMP

if (PHASE\_TRACE(Js::AsmjsInterpreterPhase, this->m\_functionBody))

{

Output::Print(L"%d.%d:Executing %s at offset 0x%X\n", this->m\_functionBody->GetSourceContextId(), this->m\_functionBody->GetLocalFunctionId(), Js::OpCodeUtilAsmJs::GetOpCodeName((Js::OpCodeAsmJs)(op+((int)isExtended<<8))), DEBUG\_currentByteOffset);

}

#endif

return op;

}

#endif

\_\_declspec(noinline)

Var InterpreterStackFrame::ProcessThunk(void\* address, void\* addressOfReturnAddress)

{

PushPopFrameHelper pushPopFrameHelper(this, address, addressOfReturnAddress);

return this->Process();

}

template<> uint32 InterpreterStackFrame::LogSizeOf<uint8>(){return 0;}

template<> uint32 InterpreterStackFrame::LogSizeOf<int8>(){return 0;}

template<> uint32 InterpreterStackFrame::LogSizeOf<uint16>(){return 1;}

template<> uint32 InterpreterStackFrame::LogSizeOf<int16>(){return 1;}

template<> uint32 InterpreterStackFrame::LogSizeOf<uint32>(){return 2;}

template<> uint32 InterpreterStackFrame::LogSizeOf<int32>(){return 2;}

template<> uint32 InterpreterStackFrame::LogSizeOf<float>(){return 2;}

template<> uint32 InterpreterStackFrame::LogSizeOf<double>(){return 3;}

Var InterpreterStackFrame::ProcessAsmJsModule()

{

#ifdef ASMJS\_PLAT

Js::FunctionBody\* asmJsModuleFunctionBody = GetFunctionBody();

AsmJsModuleInfo\* info = asmJsModuleFunctionBody->GetAsmJsModuleInfo();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Configuration::Global.flags.ForceAsmJsLinkFail)

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Forcing link failure");

return this->ProcessLinkFailedAsmJsModule();

}

#endif

if( m\_inSlotsCount != info->GetArgInCount() + 1 )

{

// Error reparse without asm.js

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Invalid module argument count");

return this->ProcessLinkFailedAsmJsModule();

}

const AsmJsModuleMemory& moduleMemory = info->GetModuleMemory();

Var\* moduleMemoryPtr = RecyclerNewArray( scriptContext->GetRecycler(), Var, moduleMemory.mMemorySize );

Var\* arrayBufferPtr = moduleMemoryPtr + moduleMemory.mArrayBufferOffset;

Assert(moduleMemory.mArrayBufferOffset == AsmJsModuleMemory::MemoryTableBeginOffset);

Var\* stdLibPtr = moduleMemoryPtr + moduleMemory.mStdLibOffset;

int\* localIntSlots = (int\*)(moduleMemoryPtr + moduleMemory.mIntOffset);

float\* localFloatSlots = (float\*)(moduleMemoryPtr + moduleMemory.mFloatOffset);

double\* localDoubleSlots = (double\*)(moduleMemoryPtr + moduleMemory.mDoubleOffset);

Var\* localFunctionImports = moduleMemoryPtr + moduleMemory.mFFIOffset ;

Var\* localModuleFunctions = moduleMemoryPtr + moduleMemory.mFuncOffset ;

Var\*\* localFunctionTables = (Var\*\*)(moduleMemoryPtr + moduleMemory.mFuncPtrOffset) ;

AsmJsSIMDValue\* localSimdSlots = nullptr;

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

localSimdSlots = ((AsmJsSIMDValue\*)moduleMemoryPtr) + moduleMemory.mSimdOffset; // simdOffset is in SIMDValues

}

#if 0

// Align SIMD regs to 128 bits.

// We only have space to align if there are any SIMD variables. Otherwise, leave unaligned.

if (info->GetSimdRegCount())

{

AssertMsg((moduleMemory.mMemorySize / SIMD\_SLOTS\_SPACE) - moduleMemory.mSimdOffset >= 1, "Not enough space in module memory to align SIMD vars");

localSimdSlots = (AsmJsSIMDValue\*)::Math::Align<int>((int)localSimdSlots, sizeof(AsmJsSIMDValue));

}

#endif

ThreadContext\* threadContext = this->scriptContext->GetThreadContext();

\*stdLibPtr = (m\_inSlotsCount > 1) ? m\_inParams[1] : nullptr;

Var foreign = (m\_inSlotsCount > 2) ? m\_inParams[2] : nullptr;

\*arrayBufferPtr = (m\_inSlotsCount > 3) ? m\_inParams[3] : nullptr;

//cache the current state of the disable implicit call flag

DisableImplicitFlags prevDisableImplicitFlags = threadContext->GetDisableImplicitFlags();

ImplicitCallFlags saveImplicitcallFlags = threadContext->GetImplicitCallFlags();

// Disable implicit calls to check if any of the VarImport or Function Import leads to implicit calls

threadContext->DisableImplicitCall();

threadContext->SetImplicitCallFlags(ImplicitCallFlags::ImplicitCall\_None);

bool checkParamResult = ASMLink::CheckParams(this->scriptContext, info, \*stdLibPtr, foreign, \*arrayBufferPtr);

if (!checkParamResult)

{

// don't need to print, because checkParams will do it for us

goto linkFailure;

}

else if(this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Params have side effects");

return this->ProcessLinkFailedAsmJsModule();

}

// Initialize Variables

for (int i = 0; i < info->GetVarCount(); i++)

{

const auto& var = info->GetVar( i );

const AsmJsVarType type(var.type);

if(type.isInt() )

{

localIntSlots[var.location] = var.initialiser.intInit;

}

else if (type.isFloat())

{

localFloatSlots[var.location] = var.initialiser.floatInit;

}

else if (type.isDouble())

{

localDoubleSlots[var.location] = var.initialiser.doubleInit;

}

else if (scriptContext->GetConfig()->IsSimdjsEnabled() && type.isSIMD())

{

// e.g. var g = f4(0.0, 0.0, 0.0, 0.0);

localSimdSlots[var.location] = var.initialiser.simdInit;

}

else {

Assert(UNREACHED);

}

}

// Load constant variables

for( int i = 0; i < info->GetVarImportCount(); i++ )

{

const auto& import = info->GetVarImport( i );

const AsmJsVarType type(import.type);

// this might throw, but it would anyway in non-asm.js

Var value = JavascriptOperators::OP\_GetProperty( foreign, import.field, scriptContext );

// check if there is implicit call and if there is implicit call then clear the disableimplicitcall flag

if (this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Accessing var import %s has side effects", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

return this->ProcessLinkFailedAsmJsModule();

}

if (CONFIG\_FLAG(AsmJsEdge))

{

// emscripten had a bug which caused this check to fail in some circumstances, so this check fails for some demos

if (!TaggedNumber::Is(value) && (!RecyclableObject::Is(value) || DynamicType::Is(RecyclableObject::FromVar(value)->GetTypeId())))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Var import %s must be primitive", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

goto linkFailure;

}

}

if(type.isInt() )

{

int val = JavascriptMath::ToInt32( value, scriptContext );

localIntSlots[import.location] = val;

}

else if (type.isFloat())

{

float val = (float)JavascriptConversion::ToNumber(value, scriptContext);

localFloatSlots[import.location] = val;

}

else if (type.isDouble())

{

double val = JavascriptConversion::ToNumber( value, scriptContext );

localDoubleSlots[import.location] = val;

}

else if (scriptContext->GetConfig()->IsSimdjsEnabled() && type.isSIMD())

{

// e.g. var g = f4(imports.v);

bool valid = false;

AsmJsSIMDValue val;

val.Zero();

switch (type.which())

{

case AsmJsVarType::Int32x4:

valid = JavascriptSIMDInt32x4::Is(value);

val = ((JavascriptSIMDInt32x4\*)value)->GetValue();

break;

case AsmJsVarType::Float32x4:

valid = JavascriptSIMDFloat32x4::Is(value);

val = ((JavascriptSIMDFloat32x4\*)value)->GetValue();

break;

case AsmJsVarType::Float64x2:

valid = JavascriptSIMDFloat64x2::Is(value);

val = ((JavascriptSIMDFloat64x2\*)value)->GetValue();

break;

default:

Assert(UNREACHED);

};

if (!valid)

{

// link failure of SIMD values imports.

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Foreign var import %s is not SIMD type", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

goto linkFailure;

}

localSimdSlots[import.location] = val;

}

// check for implicit call after converting to number

if (this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

// Runtime error

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Accessing var import %s has side effects", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

return this->ProcessLinkFailedAsmJsModule();

}

}

// Load external functions

for( int i = 0; i < info->GetFunctionImportCount(); i++ )

{

const auto& import = info->GetFunctionImport( i );

// this might throw, but it would anyway in non-asm.js

Var importFunc = JavascriptOperators::OP\_GetProperty( foreign, import.field, scriptContext );

// check if there is implicit call and if there is implicit call then clear the disableimplicitcall flag

if (this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Accessing foreign function import %s has side effects", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

return this->ProcessLinkFailedAsmJsModule();

}

if( !JavascriptFunction::Is( importFunc ) )

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Foreign function import %s is not a function", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

goto linkFailure;

}

localFunctionImports[import.location] = importFunc;

}

if (\*arrayBufferPtr)

{

(\*(ArrayBuffer\*\*)arrayBufferPtr)->SetIsAsmJsBuffer();

}

threadContext->SetDisableImplicitFlags(prevDisableImplicitFlags);

threadContext->SetImplicitCallFlags(saveImplicitcallFlags);

FrameDisplay\* pDisplay = RecyclerNewPlus(scriptContext->GetRecycler(), sizeof(void\*), FrameDisplay, 1);

pDisplay->SetItem( 0, moduleMemoryPtr );

for (int i = 0; i < info->GetFunctionCount(); i++)

{

const auto& modFunc = info->GetFunction(i);

// TODO: add more runtime checks here

auto proxy = m\_functionBody->GetNestedFuncReference(i);

AsmJsScriptFunction\* scriptFuncObj = (AsmJsScriptFunction\*)ScriptFunction::OP\_NewScFunc(pDisplay, (FunctionProxy\*\*)proxy);

localModuleFunctions[modFunc.location] = scriptFuncObj;

if (i == 0 && info->GetUsesChangeHeap())

{

scriptFuncObj->GetDynamicType()->SetEntryPoint(AsmJsChangeHeapBuffer);

}

else

{

if (scriptFuncObj->GetDynamicType()->GetEntryPoint() == DefaultDeferredDeserializeThunk)

{

JavascriptFunction::DeferredDeserialize(scriptFuncObj);

}

scriptFuncObj->GetDynamicType()->SetEntryPoint(AsmJsExternalEntryPoint);

scriptFuncObj->GetFunctionBody()->GetAsmJsFunctionInfo()->SetModuleFunctionBody(asmJsModuleFunctionBody);

}

scriptFuncObj->SetModuleMemory(moduleMemoryPtr);

if (!info->IsRuntimeProcessed())

{

// don't reset entrypoint upon relinking

FunctionEntryPointInfo\* entypointInfo = (FunctionEntryPointInfo\*)scriptFuncObj->GetEntryPointInfo();

entypointInfo->SetIsAsmJSFunction(true);

entypointInfo->SetModuleAddress((uintptr\_t)moduleMemoryPtr);

#if DYNAMIC\_INTERPRETER\_THUNK

if (!PHASE\_ON1(AsmJsJITTemplatePhase))

{

entypointInfo->address = AsmJsDefaultEntryThunk;

}

#endif

}

}

// Initialize function table arrays

for( int i = 0; i < info->GetFunctionTableCount(); i++ )

{

const auto& modFuncTable = info->GetFunctionTable( i );

Var\* funcTableArray = RecyclerNewArray( scriptContext->GetRecycler(), Var, modFuncTable.size );

for (uint j = 0; j < modFuncTable.size ; j++)

{

// get the module function index

const RegSlot index = modFuncTable.moduleFunctionIndex[j];

// assign the module function pointer to the array

Var functionPtr = localModuleFunctions[index];

funcTableArray[j] = functionPtr;

}

localFunctionTables[i] = funcTableArray;

}

// Do MTJRC/MAIC:0 check

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if ((PHASE\_ON1(Js::AsmJsJITTemplatePhase) && CONFIG\_FLAG(MaxTemplatizedJitRunCount) == 0) || (!PHASE\_ON1(Js::AsmJsJITTemplatePhase) && CONFIG\_FLAG(MaxAsmJsInterpreterRunCount) == 0))

{

if (PHASE\_TRACE1(AsmjsEntryPointInfoPhase))

{

Output::Print(L"%s Scheduling For Full JIT at callcount:%d\n", asmJsModuleFunctionBody->GetDisplayName(), 0);

Output::Flush();

}

for (int i = 0; i < info->GetFunctionCount(); i++)

{

ScriptFunction\* functionObj = (ScriptFunction\*)localModuleFunctions[i];

AnalysisAssert(functionObj != nullptr);

// don't want to generate code for APIs like changeHeap

if (functionObj->GetEntryPoint() == Js::AsmJsExternalEntryPoint)

{

GenerateFunction(asmJsModuleFunctionBody->GetScriptContext()->GetNativeCodeGenerator(), functionObj->GetFunctionBody(), functionObj);

}

}

}

#endif

info->SetIsRuntimeProcessed(true);

// create export object

if( info->GetExportsCount() )

{

Var newObj = JavascriptOperators::NewScObjectLiteral( GetScriptContext(), info->GetExportsIdArray(),

this->GetFunctionBody()->GetObjectLiteralTypeRef( 0 ) );

for( int i = 0; i < info->GetExportsCount(); i++ )

{

auto ex = info->GetExport( i );

Var func = localModuleFunctions[\*ex.location];

JavascriptOperators::OP\_InitProperty( newObj, \*ex.id, func );

}

SetReg( (RegSlot) 0, newObj );

return newObj;

}

// export only 1 function

Var exportFunc = localModuleFunctions[info->GetExportFunctionIndex()];

SetReg((RegSlot)0, exportFunc);

return exportFunc;

linkFailure:

threadContext->SetDisableImplicitFlags(prevDisableImplicitFlags);

threadContext->SetImplicitCallFlags(saveImplicitcallFlags);

return this->ProcessLinkFailedAsmJsModule();

}

Var InterpreterStackFrame::ProcessLinkFailedAsmJsModule()

{

AsmJSCompiler::OutputError(this->scriptContext, L"asm.js linking failed.");

Js::FunctionBody\* asmJsModuleFunctionBody = GetFunctionBody();

AsmJsModuleInfo\* info = asmJsModuleFunctionBody->GetAsmJsModuleInfo();

// do not support relinking with failed relink

if (info->IsRuntimeProcessed())

{

Js::Throw::OutOfMemory();

}

ScriptFunction \* funcObj = GetJavascriptFunction();

ScriptFunction::ReparseAsmJsModule(&funcObj);

const bool doProfile =

funcObj->GetFunctionBody()->GetInterpreterExecutionMode(false) == ExecutionMode::ProfilingInterpreter ||

GetScriptContext()->IsInDebugMode() && DynamicProfileInfo::IsEnabled(funcObj->GetFunctionBody());

DynamicProfileInfo \* dynamicProfileInfo = nullptr;

if (doProfile)

{

dynamicProfileInfo = funcObj->GetFunctionBody()->GetDynamicProfileInfo();

funcObj->GetScriptContext()->GetThreadContext()->ClearImplicitCallFlags();

}

// after reparsing, we want to also use a new interpreter stack frame, as it will have different characteristics than the asm.js version

InterpreterStackFrame::Setup setup(funcObj, m\_inParams, m\_inSlotsCount);

size\_t varAllocCount = setup.GetAllocationVarCount();

size\_t varSizeInBytes = varAllocCount \* sizeof(Var);

Var\* allocation = nullptr;

DWORD\_PTR stackAddr;

bool fReleaseAlloc = false;

if (varAllocCount > InterpreterStackFrame::LocalsThreshold)

{

ArenaAllocator \*tmpAlloc = nullptr;

fReleaseAlloc = GetScriptContext()->EnsureInterpreterArena(&tmpAlloc);

allocation = (Var\*)tmpAlloc->Alloc(varSizeInBytes);

// use a stack address so the debugger stepping logic works (step-out, for example, compares stack depths to determine when to complete the step)

// debugger stepping does not matter here, but it's worth being consistent with normal stack frame

stackAddr = reinterpret\_cast<DWORD\_PTR>(&allocation);

}

else

{

PROBE\_STACK\_PARTIAL\_INITIALIZED\_INTERPRETER\_FRAME(GetScriptContext(), Js::Constants::MinStackInterpreter + varSizeInBytes);

allocation = (Var\*)\_alloca(varSizeInBytes);

stackAddr = reinterpret\_cast<DWORD\_PTR>(allocation);

}

#if DBG

Js::RecyclableObject \* invalidStackVar = (Js::RecyclableObject\*)\_alloca(sizeof(Js::RecyclableObject));

memset(invalidStackVar, 0xFE, sizeof(Js::RecyclableObject));

InterpreterStackFrame \* newInstance = newInstance = setup.InitializeAllocation(allocation, funcObj->GetFunctionBody()->GetHasImplicitArgIns(), doProfile, nullptr, stackAddr, invalidStackVar);

#else

InterpreterStackFrame \* newInstance = newInstance = setup.InitializeAllocation(allocation, funcObj->GetFunctionBody()->GetHasImplicitArgIns(), doProfile, nullptr, stackAddr);

#endif

newInstance->m\_reader.Create(funcObj->GetFunctionBody());

// now that we have set up the new frame, let's interpret it!

funcObj->GetFunctionBody()->BeginExecution();

PushPopFrameHelper(newInstance, \_ReturnAddress(), \_AddressOfReturnAddress());

Var retVal = newInstance->ProcessUnprofiled();

if (doProfile)

{

dynamicProfileInfo->RecordImplicitCallFlags(GetScriptContext()->GetThreadContext()->GetImplicitCallFlags());

}

if (fReleaseAlloc)

{

GetScriptContext()->ReleaseInterpreterArena();

}

return retVal;

#else

Assert(UNREACHED);

return nullptr;

#endif

}

#if DBG\_DUMP

int AsmJsCallDepth = 0;

#endif

void InterpreterStackFrame::PrintStack(const int\* const intSrc, const float\* const fltSrc, const double\* const dblSrc, int intConstCount, int floatConstCount, int doubleConstCount, const wchar\_t\* state)

{

Output::Print(L"\n");

Output::Print(L"Interpreter Constant Stack Data(%s)\n", state);

Output::Print(L"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

Output::Print(L"Int Data\n");

Output::Print(L"--------\n");

for (int count = 0; count < intConstCount; count++)

{

Output::Print(L"Index:%d Value:%d \n", count, intSrc[count]);

}

Output::Print(L"\n");

Output::Print(L"Float Data\n");

Output::Print(L"----------\n");

for (int count = 0; count < floatConstCount; count++)

{

Output::Print(L"Index:%d Value:%f \n", count, fltSrc[count]);

}

Output::Print(L"\n");

Output::Print(L"Double Data\n");

Output::Print(L"-----------\n");

for (int count = 0; count < doubleConstCount; count++)

{

Output::Print(L"Index:%d Value:%g \n", count, dblSrc[count]);

}

Output::Print(L"\n");

}

#ifndef TEMP\_DISABLE\_ASMJS

// Function memory allocation should be done the same way as

// T AsmJsCommunEntryPoint(Js::ScriptFunction\* func, ...) (AsmJSJitTemplate.cpp)

// update any changes there

/\*

This function does the following fixup

Stack Before Stack After

============== ================

| VarConstants | | VarConstants |

|--------------| |-----------------

| IntConstants | | IntConstants |

|--------------| | ------------ |

| FloatConst | | Int Vars+Tmps |

|--------------| |----------------|

| DoubleConst | | FloatConst |

|--------------| | ---------- |

| Var&Temps | | Flt Vars+tmps |

|==============| |----------------|

| DoubleConst |

| ----------- |

| Dbl Vars+Tmps |

================

intSrc,FltSrc&DblSrc are pointers to the stack before the change

m\_localIntSlots,m\_localFloatSlots,m\_localDoubleSlots are pointers to the stack after the change

\*/

void InterpreterStackFrame::AlignMemoryForAsmJs()

{

FunctionBody \*const functionBody = GetFunctionBody();

ScriptFunction\* func = GetJavascriptFunction();

//schedule for codegen here only if TJ is collected

if (!functionBody->GetIsAsmJsFullJitScheduled() && !PHASE\_OFF(BackEndPhase, functionBody)

&& !PHASE\_OFF(FullJitPhase, functionBody) && !this->scriptContext->GetConfig()->IsNoNative())

{

int callCount = ++((FunctionEntryPointInfo\*)func->GetEntryPointInfo())->callsCount;

bool doSchedule = false;

const int minAsmJsInterpretRunCount = (int)CONFIG\_FLAG(MinAsmJsInterpreterRunCount);

if (callCount >= minAsmJsInterpretRunCount)

{

doSchedule = true;

}

if (doSchedule && !functionBody->GetIsAsmJsFullJitScheduled())

{

#if ENABLE\_NATIVE\_CODEGEN

if (PHASE\_TRACE1(AsmjsEntryPointInfoPhase))

{

Output::Print(L"Scheduling For Full JIT from Interpreter at callcount:%d\n", callCount);

}

GenerateFunction(functionBody->GetScriptContext()->GetNativeCodeGenerator(), functionBody, func);

#endif

functionBody->SetIsAsmJsFullJitScheduled(true);

}

}

AsmJsFunctionInfo\* info = functionBody->GetAsmJsFunctionInfo();

const int intConstCount = info->GetIntConstCount();

const int doubleConstCount = info->GetDoubleConstCount();

const int floatConstCount = info->GetFloatConstCount();

const int simdConstCount = info->GetSimdConstCount();

// Offset of doubles from (double\*)m\_localSlot

const int intOffset = info->GetIntByteOffset() / sizeof(int);

const int doubleOffset = info->GetDoubleByteOffset() / sizeof(double);

const int floatOffset = info->GetFloatByteOffset() / sizeof(float);

const int simdByteOffset = info->GetSimdByteOffset();// in bytes;

int\* intSrc = (int\*)(m\_localSlots + AsmJsFunctionMemory::RequiredVarConstants);

// Where all int value starts

m\_localIntSlots = ((int\*)m\_localSlots) + intOffset;

// where int arguments starts

// int\* intArgDst = m\_localIntSlots + intConstCount;

// Where float constants currently are

float\* floatSrc = (float\*)(intSrc + intConstCount);

// where all float value starts with the new layout

m\_localFloatSlots = ((float\*)m\_localSlots) + floatOffset;

// Where double arguments starts

// float\* floatArgDst = m\_localFloatSlots + floatConstCount;

// Where double constants currently are

double\* doubleSrc = (double\*)(floatSrc + floatConstCount);

// where all double value starts

m\_localDoubleSlots = ((double\*)m\_localSlots) + doubleOffset;

// Where double arguments starts

// double\* doubleArgDst = m\_localDoubleSlots + doubleConstCount;

AsmJsSIMDValue\* simdSrc = nullptr;

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

simdSrc = (AsmJsSIMDValue\*)(doubleSrc + doubleConstCount);

m\_localSimdSlots = (AsmJsSIMDValue\*)((char\*)m\_localSlots + simdByteOffset);

}

// Load module environment

FrameDisplay\* frame = this->function->GetEnvironment();

m\_localSlots[AsmJsFunctionMemory::ModuleEnvRegister] = frame->GetItem(0);

m\_localSlots[AsmJsFunctionMemory::ArrayBufferRegister] = (Var\*)frame->GetItem(0) + AsmJsModuleMemory::MemoryTableBeginOffset;

m\_localSlots[AsmJsFunctionMemory::ArraySizeRegister] = 0; // do not cache ArraySize in the interpreter

m\_localSlots[AsmJsFunctionMemory::ScriptContextBufferRegister] = functionBody->GetScriptContext();

if (PHASE\_TRACE1(AsmjsInterpreterStackPhase))

{

PrintStack(intSrc, floatSrc, doubleSrc, intConstCount, floatConstCount, doubleConstCount, L"Before Shuffling");

}

// Copying has to happen in that order in order not to overwrite constants

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

memcpy\_s(m\_localSimdSlots, simdConstCount\*sizeof(AsmJsSIMDValue), simdSrc, simdConstCount\*sizeof(AsmJsSIMDValue));

}

// Moving the double and floats to their slot position. We must move the doubles first so that we do not overwrite the doubles stack with floats

memcpy\_s(m\_localDoubleSlots, doubleConstCount\*sizeof(double), doubleSrc, doubleConstCount\*sizeof(double));

memcpy\_s(m\_localFloatSlots, floatConstCount\*sizeof(float), floatSrc, floatConstCount\*sizeof(float));

if (PHASE\_TRACE1(AsmjsInterpreterStackPhase))

{

PrintStack(m\_localIntSlots, m\_localFloatSlots, m\_localDoubleSlots, intConstCount, floatConstCount, doubleConstCount, L"After Shuffling");

}

int\* intArg;

double\* doubleArg;

float\* floatArg;

intArg = m\_localIntSlots + intConstCount;

doubleArg = m\_localDoubleSlots + doubleConstCount;

floatArg = m\_localFloatSlots + floatConstCount;

AsmJsSIMDValue\* simdArg = m\_localSimdSlots + simdConstCount;

// Move the arguments to the right location

ArgSlot argCount = info->GetArgCount();

#if \_M\_X64

uint homingAreaSize = 0;

#endif

uintptr argAddress = (uintptr)m\_inParams;

for (ArgSlot i = 0; i < argCount; i++)

{

#if \_M\_X64

// 3rd Argument should be at the end of the homing area.

Assert(i != 3 || argAddress == (uintptr)m\_inParams + homingAreaSize);

if (i < 3)

{

// for x64 we spill the first 3 floating point args below the rest of the arguments on the stack

// m\_inParams will be from DynamicInterpreterThunk's frame. Floats are in InterpreterAsmThunk's frame. Stack will be set up like so:

// DIT arg2 <- first scriptArg, m\_inParams points here

// DIT arg1

// padding

// IAT r9 home

// IAT r8 home

// IAT rdx home

// IAT rcx home

// IAT return address

// IAT push rbp

// IAT padding

// IAT xmm3 spill

// IAT xmm2 spill

// IAT xmm1 spill <- floatSpillAddress for arg1

// floats are spilled as xmmwords

uintptr floatSpillAddress = (uintptr)m\_inParams - MachPtr \* (15 - 2\*i);

if (info->GetArgType(i).isInt())

{

\*intArg = \*(int\*)argAddress;

++intArg;

homingAreaSize += MachPtr;

}

else if (info->GetArgType(i).isFloat())

{

\*floatArg = \*(float\*)floatSpillAddress;

++floatArg;

homingAreaSize += MachPtr;

}

else if (info->GetArgType(i).isDouble())

{

\*doubleArg = \*(double\*)floatSpillAddress;

++doubleArg;

homingAreaSize += MachPtr;

}

else

{

Assert(info->GetArgType(i).isSIMD());

\*simdArg = \*(AsmJsSIMDValue\*)floatSpillAddress;

++simdArg;

homingAreaSize += sizeof(AsmJsSIMDValue);

}

if (scriptContext->GetConfig()->IsSimdjsEnabled() && i == 2) // last argument ?

{

// If we have simd arguments, the homing area in m\_inParams can be larger than 3 64-bit slots. This is because SIMD values are unboxed there too.

// After unboxing, the homing area is overwritten by rdx, r8 and r9, and we read/skip 64-bit slots from the homing area (argAddress += MachPtr).

// After the last argument of the 3 is read, we need to advance argAddress to skip over the possible extra space and to the start of the rest of the arguments.

argAddress = (uintptr)m\_inParams + homingAreaSize;

}

else

{

argAddress += MachPtr;

}

}

else

#endif

if (info->GetArgType(i).isInt())

{

\*intArg = \*(int\*)argAddress;

++intArg;

argAddress += MachPtr;

}

else if (info->GetArgType(i).isFloat())

{

\*floatArg = \*(float\*)argAddress;

++floatArg;

argAddress += MachPtr;

}

else if (info->GetArgType(i).isDouble())

{

Assert(info->GetArgType(i).isDouble());

\*doubleArg = \*(double\*)argAddress;

++doubleArg;

argAddress += sizeof(double);

}

else if (scriptContext->GetConfig()->IsSimdjsEnabled() && info->GetArgType(i).isSIMD())

{

\*simdArg = \*(AsmJsSIMDValue\*)argAddress;

++simdArg;

argAddress += sizeof(AsmJsSIMDValue);

}

else

{

AssertMsg(UNREACHED, "Invalid function arg type.");

}

}

#if DBG\_DUMP

const bool tracingFunc = PHASE\_TRACE( AsmjsFunctionEntryPhase, functionBody );

if( tracingFunc )

{

if( AsmJsCallDepth )

{

Output::Print( L"%\*c", AsmJsCallDepth,' ');

}

Output::Print( L"Executing function %s", functionBody->GetDisplayName());

++AsmJsCallDepth;

}

#endif

#if DBG\_DUMP

if (tracingFunc)

{

Output::Print(L"){\n");

}

#endif

if( info->GetReturnType() == AsmJsRetType::Void )

{

m\_localSlots[0] = JavascriptOperators::OP\_LdUndef( scriptContext );

}

}

#endif

///----------------------------------------------------------------------------

///

/// InterpreterStackFrame::Process

///

/// Process() processes a single loop of execution for the current

/// JavascriptFunction being executed:

/// - Individual instructions are dispatched to specific handlers for different

/// OpCodes.

///

///----------------------------------------------------------------------------

#if ENABLE\_PROFILE\_INFO

#define INTERPRETERLOOPNAME ProcessProfiled

#define PROVIDE\_INTERPRETERPROFILE

#include "Interpreterloop.inl"

#undef PROVIDE\_INTERPRETERPROFILE

#undef INTERPRETERLOOPNAME

#endif

#define INTERPRETERLOOPNAME ProcessUnprofiled

#include "Interpreterloop.inl"

#undef INTERPRETERLOOPNAME

#ifndef TEMP\_DISABLE\_ASMJS

#define INTERPRETERLOOPNAME ProcessAsmJs

#define INTERPRETER\_ASMJS

#include "InterpreterProcessOpCodeAsmJs.h"

#include "Interpreterloop.inl"

#undef INTERPRETER\_ASMJS

#undef INTERPRETERLOOPNAME

#endif

// For now, always collect profile data when debugging,

// otherwise the backend will be confused if there's no profile data.

#define INTERPRETERLOOPNAME ProcessWithDebugging

#define PROVIDE\_DEBUGGING

#if ENABLE\_PROFILE\_INFO

#define PROVIDE\_INTERPRETERPROFILE

#endif

#include "Interpreterloop.inl"

#if ENABLE\_PROFILE\_INFO

#undef PROVIDE\_INTERPRETERPROFILE

#endif

#undef PROVIDE\_DEBUGGING

#undef INTERPRETERLOOPNAME

Var InterpreterStackFrame::Process()

{

#if ENABLE\_PROFILE\_INFO

class AutoRestore

{

private:

InterpreterStackFrame \*const interpreterStackFrame;

const uint32 savedSwitchProfileModeOnLoopEndNumber;

const bool savedIsAutoProfiling;

const bool savedSwitchProfileMode;

public:

AutoRestore(InterpreterStackFrame \*const interpreterStackFrame)

: interpreterStackFrame(interpreterStackFrame),

savedIsAutoProfiling(interpreterStackFrame->isAutoProfiling),

savedSwitchProfileMode(interpreterStackFrame->switchProfileMode),

savedSwitchProfileModeOnLoopEndNumber(interpreterStackFrame->switchProfileModeOnLoopEndNumber)

{

}

~AutoRestore()

{

interpreterStackFrame->isAutoProfiling = savedIsAutoProfiling;

interpreterStackFrame->switchProfileMode = savedSwitchProfileMode;

interpreterStackFrame->switchProfileModeOnLoopEndNumber = savedSwitchProfileModeOnLoopEndNumber;

}

} autoRestore(this);

#endif

if ((m\_flags & Js::InterpreterStackFrameFlags\_FromBailOut) && !(m\_flags & InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode))

{

if (this->ehBailoutData)

{

m\_flags |= Js::InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode;

EHBailoutData \* topLevelEHBailoutData = this->ehBailoutData;

while (topLevelEHBailoutData->parent->nestingDepth != -1)

{

topLevelEHBailoutData->parent->child = topLevelEHBailoutData;

topLevelEHBailoutData = topLevelEHBailoutData->parent;

}

ProcessTryCatchBailout(topLevelEHBailoutData, this->ehBailoutData->nestingDepth);

m\_flags &= ~Js::InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode;

this->ehBailoutData = nullptr;

}

}

#ifndef TEMP\_DISABLE\_ASMJS

FunctionBody \*const functionBody = GetFunctionBody();

if( functionBody->GetIsAsmjsMode() )

{

AsmJsFunctionInfo\* asmInfo = functionBody->GetAsmJsFunctionInfo();

if (asmInfo)

{

AlignMemoryForAsmJs();

Var returnVar = ProcessAsmJs();

#if DBG\_DUMP

if( PHASE\_TRACE( AsmjsFunctionEntryPhase, functionBody ) )

{

--AsmJsCallDepth;

if( AsmJsCallDepth )

{

Output::Print( L"%\*c}", AsmJsCallDepth, ' ' );

}

else

{

Output::Print( L"}" );

}

switch( asmInfo->GetReturnType().which() )

{

case AsmJsRetType::Void:

break;

case AsmJsRetType::Signed:

Output::Print( L" = %d", JavascriptMath::ToInt32( returnVar, scriptContext ) );

break;

case AsmJsRetType::Float:

case AsmJsRetType::Double:

Output::Print( L" = %.4f", JavascriptConversion::ToNumber( returnVar, scriptContext ) );

break;

default:

break;

}

Output::Print( L";\n" );

}

#endif

return returnVar;

}

else

{

Assert(functionBody->GetAsmJsModuleInfo());

return ProcessAsmJsModule();

}

}

#endif

#if ENABLE\_PROFILE\_INFO

switchProfileMode = false;

switchProfileModeOnLoopEndNumber = 0u - 1;

#endif

ByteCodeDumper::Dump(functionBody);

#if ENABLE\_PROFILE\_INFO

const ExecutionMode interpreterExecutionMode =

functionBody->GetInterpreterExecutionMode(!!(GetFlags() & InterpreterStackFrameFlags\_FromBailOut));

if(interpreterExecutionMode == ExecutionMode::ProfilingInterpreter)

{

isAutoProfiling = false;

return ProcessProfiled();

}

Assert(

interpreterExecutionMode == ExecutionMode::Interpreter ||

interpreterExecutionMode == ExecutionMode::AutoProfilingInterpreter);

isAutoProfiling = interpreterExecutionMode == ExecutionMode::AutoProfilingInterpreter;

Var result;

while(true)

{

Assert(!switchProfileMode);

result = ProcessUnprofiled();

Assert(!(switchProfileMode && result));

if(switchProfileMode)

{

switchProfileMode = false;

}

else

{

break;

}

Assert(isAutoProfiling);

#if DBG\_DUMP

if(PHASE\_TRACE(InterpreterAutoProfilePhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"InterpreterAutoProfile - Func %s - Started profiling\n", functionBody->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

#endif

Assert(!switchProfileMode);

result = ProcessProfiled();

Assert(!(switchProfileMode && result));

if(switchProfileMode)

{

switchProfileMode = false;

}

else

{

break;

}

Assert(isAutoProfiling);

#if DBG\_DUMP

if(PHASE\_TRACE(InterpreterAutoProfilePhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"InterpreterAutoProfile - Func %s - Stopped profiling\n", functionBody->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

#endif

}

return result;

#else

return ProcessUnprofiled();

#endif

}

template <class T>

void InterpreterStackFrame::OP\_GetMethodProperty(unaligned T \*playout)

{

Var varInstance = GetReg(playout->Instance);

OP\_GetMethodProperty(varInstance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetLocalMethodProperty(unaligned T \*playout)

{

OP\_GetMethodProperty(this->localClosure, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetMethodProperty(Var varInstance, unaligned T \*playout)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(varInstance);

#endif

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

RecyclableObject\* obj = NULL;

if (RecyclableObject::Is(varInstance))

{

obj = RecyclableObject::FromVar(varInstance);

if ((propertyId == PropertyIds::apply || propertyId == PropertyIds::call) && ScriptFunction::Is(obj))

{

// If the property being loaded is "apply"/"call", make an optimistic assumption that apply/call is not overridden and

// undefer the function right here if it was defer parsed before. This is required so that the load of "apply"/"call"

// happens from the same "type". Otherwise, we will have a polymorphic cache for load of "apply"/"call".

ScriptFunction \*fn = ScriptFunction::FromVar(obj);

if(fn->GetType()->GetEntryPoint() == JavascriptFunction::DeferredParsingThunk)

{

JavascriptFunction::DeferredParse(&fn);

}

}

}

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var aValue;

if (obj &&

CacheOperators::TryGetProperty<true, true, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &aValue, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, aValue);

return;

}

OP\_GetMethodProperty\_NoFastPath(varInstance, playout);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetMethodProperty\_NoFastPath(Var instance, unaligned T \*playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetMethod<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(instance, propertyId, value, true);

}

#endif

SetReg(playout->Value, value);

}

template <class T>

void InterpreterStackFrame::OP\_GetRootMethodProperty(unaligned T \*playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

Js::Var instance = this->GetRootObject();

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

DynamicObject \*obj = DynamicObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var aValue;

if (CacheOperators::TryGetProperty<true, true, false, false, false, false, true, false, false>(

obj, true, obj, propertyId, &aValue, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, aValue);

return;

}

OP\_GetRootMethodProperty\_NoFastPath(playout);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetRootMethodProperty\_NoFastPath(unaligned T \*playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var rootInstance = this->GetRootObject();

Var value = JavascriptOperators::PatchGetRootMethod<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

DynamicObject::FromVar(rootInstance),

propertyId

);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(rootInstance, propertyId, value, true);

}

#endif

SetReg(playout->Value, value);

}

template <class T>

void InterpreterStackFrame::OP\_GetMethodPropertyScoped(unaligned T \*playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

Var varInstance = GetReg(playout->Instance);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

RecyclableObject\* obj = NULL;

if (RecyclableObject::Is(varInstance))

{

obj = RecyclableObject::FromVar(varInstance);

}

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var aValue;

if (obj &&

CacheOperators::TryGetProperty<true, true, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &aValue, GetScriptContext(), nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, aValue);

return;

}

OP\_GetMethodPropertyScoped\_NoFastPath(playout);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetMethodPropertyScoped\_NoFastPath(unaligned T \*playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Js::Var instance = GetReg(playout->Instance);

Js::Var value = JavascriptOperators::PatchScopedGetMethod<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(instance, propertyId, value, true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetMethodProperty(unaligned T \*playout)

{

ProfiledGetProperty<T, false, true, false>(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetLocalMethodProperty(unaligned T \*playout)

{

ProfiledGetProperty<T, false, true, false>(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetRootMethodProperty(unaligned T \*playout)

{

ProfiledGetProperty<T, true, true, false>(playout, GetRootObject());

}

RecyclableObject \*

InterpreterStackFrame::OP\_CallGetFunc(Var target)

{

return JavascriptOperators::GetCallableObjectOrThrow(target, GetScriptContext());

}

void InterpreterStackFrame::OP\_AsmStartCall( const unaligned OpLayoutStartCall \* playout )

{

OP\_StartCall( playout->ArgCount/sizeof(Var) );

m\_outParams[0] = scriptContext->GetLibrary()->GetUndefined();

}

void InterpreterStackFrame::OP\_StartCall(const unaligned OpLayoutStartCall \* playout)

{

OP\_StartCall(playout->ArgCount);

}

void InterpreterStackFrame::OP\_StartCall(uint outParamCount)

{

// Save the outParams for the current callsite on the outparam stack

PushOut(m\_outParams);

// Update outParams for the indicated callsite

m\_outParams = m\_outSp;

m\_outSp += outParamCount;

AssertMsg(m\_localSlots + this->m\_functionBody->GetLocalsCount() < m\_outSp &&

m\_outSp <= (m\_localSlots + this->m\_functionBody->GetLocalsCount() + this->m\_functionBody->GetOutParamsDepth()),

"out args Stack pointer not in range after Push");

}

#ifndef TEMP\_DISABLE\_ASMJS

#if \_M\_X64

void InterpreterStackFrame::OP\_CallAsmInternal(RecyclableObject \* function)

{

AsmJsFunctionInfo\* asmInfo = ((ScriptFunction\*)function)->GetFunctionBody()->GetAsmJsFunctionInfo();

uint argsSize = asmInfo->GetArgByteSize();

ScriptFunction\* scriptFunc = (ScriptFunction\*)function;

ScriptContext \* scriptContext = function->GetScriptContext();

PROBE\_STACK\_CALL(scriptContext, function, argsSize);

Js::FunctionEntryPointInfo\* entrypointInfo = (Js::FunctionEntryPointInfo\*)scriptFunc->GetEntryPointInfo();

switch (asmInfo->GetReturnType().which())

{

case AsmJsRetType::Void:

case AsmJsRetType::Signed:

m\_localIntSlots[0] = JavascriptFunction::CallAsmJsFunction<int>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

break;

case AsmJsRetType::Double:

m\_localDoubleSlots[0] = JavascriptFunction::CallAsmJsFunction<double>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

break;

case AsmJsRetType::Float:

m\_localFloatSlots[0] = JavascriptFunction::CallAsmJsFunction<float>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

break;

case AsmJsRetType::Float32x4:

case AsmJsRetType::Int32x4:

case AsmJsRetType::Float64x2:

X86SIMDValue simdVal;

simdVal.m128\_value = JavascriptFunction::CallAsmJsFunction<\_\_m128>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

m\_localSimdSlots[0] = X86SIMDValue::ToSIMDValue(simdVal);

break;

}

Assert((uint)((ArgSlot)asmInfo->GetArgCount() + 1) == (uint)(asmInfo->GetArgCount() + 1));

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

PopOut((ArgSlot)(asmInfo->GetArgByteSize() / sizeof(Var)) + 1);

}

else

{

PopOut((ArgSlot)asmInfo->GetArgCount() + 1);

}

Assert(function);

}

#elif \_M\_IX86

void InterpreterStackFrame::OP\_CallAsmInternal(RecyclableObject \* function)

{

enum {

Void = AsmJsRetType::Void,

Signed = AsmJsRetType::Signed,

Float = AsmJsRetType::Float,

Double = AsmJsRetType::Double,

Int32x4 = AsmJsRetType::Int32x4,

Float32x4 = AsmJsRetType::Float32x4,

Float64x2 = AsmJsRetType::Float64x2

};

AsmJsFunctionInfo\* asmInfo = ((ScriptFunction\*)function)->GetFunctionBody()->GetAsmJsFunctionInfo();

Assert((uint)((ArgSlot)asmInfo->GetArgCount() + 1) == (uint)(asmInfo->GetArgCount() + 1));

uint argsSize = asmInfo->GetArgByteSize();

uint alignedSize = ::Math::Align<int32>(argsSize, 8);

ScriptFunction\* scriptFunc = (ScriptFunction\*)function;

ScriptContext \* scriptContext = function->GetScriptContext();

PROBE\_STACK\_CALL(scriptContext, function, alignedSize);

Js::FunctionEntryPointInfo\* entrypointInfo = (Js::FunctionEntryPointInfo\*)scriptFunc->GetEntryPointInfo();

int retIntVal = NULL;

float retFloatVal = NULL;

double retDoubleVal = NULL;

AsmJsSIMDValue retSimdVal;

retSimdVal.Zero();

AsmJsRetType::Which retType = (AsmJsRetType::Which) GetRetType(scriptFunc);

void \*data = nullptr;

JavascriptMethod entryPoint = (JavascriptMethod)entrypointInfo->address;

void \*savedEsp = nullptr;

\_\_asm

{

// Save ESP

mov savedEsp, esp;

mov eax, alignedSize;

// Make sure we don't go beyond guard page

cmp eax, 0x1000;

jge alloca\_probe;

sub esp, eax;

jmp dbl\_align;

alloca\_probe :

// Use alloca to allocate more then a page size

// Alloca assumes eax, contains size, and adjust ESP while

// probing each page.

call \_alloca\_probe\_16;

dbl\_align :

and esp,-8

mov data, esp;

}

{

Var\* outParam = m\_outParams + 1;

void\* dest = (void\*)data;

memmove(dest, outParam, argsSize);

}

// call variable argument function provided in entryPoint

\_\_asm

{

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

mov ecx, entryPoint

call[\_\_guard\_check\_icall\_fptr]

; no need to restore ecx('call entryPoint' is a \_\_cdecl call)

#endif

push function;

call entryPoint;

mov ebx, retType;

cmp ebx, Void;

je VoidLabel;

cmp ebx, Signed;

je SignedLabel;

cmp ebx, Float;

je FloatLabel;

cmp ebx, Double;

je DoubleLabel;

// simd

movups retSimdVal, xmm0;

jmp end

VoidLabel:

SignedLabel:

mov retIntVal, eax;

jmp end;

FloatLabel:

movss retFloatVal, xmm0;

jmp end;

DoubleLabel:

movsd retDoubleVal, xmm0;

end:

// Restore ESP

mov esp, savedEsp;

}

switch (retType)

{

case AsmJsRetType::Int32x4:

case AsmJsRetType::Float32x4:

case AsmJsRetType::Float64x2:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

m\_localSimdSlots[0] = retSimdVal;

break;

}

Assert(UNREACHED);

case AsmJsRetType::Double:

m\_localDoubleSlots[0] = retDoubleVal;

break;

case AsmJsRetType::Float:

m\_localFloatSlots[0] = retFloatVal;

break;

case AsmJsRetType::Signed:

case AsmJsRetType::Void:

m\_localIntSlots[0] = retIntVal;

break;

default:

Assume(false);

}

PopOut((uint)((ArgSlot)argsSize/sizeof(Var)) + 1);

Assert(function);

}

#else

void InterpreterStackFrame::OP\_CallAsmInternal(RecyclableObject \* function)

{

\_\_debugbreak();

}

#endif

#endif

template <class T>

void InterpreterStackFrame::OP\_AsmCall(const unaligned T\* playout)

{

OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), CallFlags\_None);

AsmJsModuleInfo::EnsureHeapAttached(this->function);

}

template <class T>

void InterpreterStackFrame::OP\_CallCommon(const unaligned T \* playout, RecyclableObject \* function, unsigned flags, const Js::AuxArray<uint32> \*spreadIndices)

{

// Always save and restore implicit call flags when calling out

// REVIEW: Can we avoid it if we don't collect dynamic profile info?

ThreadContext \* threadContext = scriptContext->GetThreadContext();

Js::ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

#if DBG

if (scriptContext->IsInDebugMode())

{

JavascriptFunction::CheckValidDebugThunk(scriptContext, function);

}

#endif

if (playout->Return == Js::Constants::NoRegister)

{

flags |= CallFlags\_NotUsed;

Arguments args(CallInfo((CallFlags)flags, playout->ArgCount), m\_outParams);

AssertMsg(args.Info.Flags == flags, "Flags don't fit into the CallInfo field?");

if (spreadIndices != nullptr)

{

JavascriptFunction::CallSpreadFunction(function, function->GetEntryPoint(), args, spreadIndices);

}

else

{

JavascriptFunction::CallFunction<true>(function, function->GetEntryPoint(), args);

}

}

else

{

flags |= CallFlags\_Value;

Arguments args(CallInfo((CallFlags)flags, playout->ArgCount), m\_outParams);

AssertMsg(args.Info.Flags == flags, "Flags don't fit into the CallInfo field?");

if (spreadIndices != nullptr)

{

SetReg((RegSlot)playout->Return, JavascriptFunction::CallSpreadFunction(function, function->GetEntryPoint(), args, spreadIndices));

}

else

{

SetReg((RegSlot)playout->Return, JavascriptFunction::CallFunction<true>(function, function->GetEntryPoint(), args));

}

}

threadContext->SetImplicitCallFlags(savedImplicitCallFlags);

PopOut(playout->ArgCount);

}

template <class T>

void InterpreterStackFrame::OP\_CallCommonI(const unaligned T \* playout, RecyclableObject \* function, unsigned flags)

{

OP\_CallCommon(playout, function, flags); // CallCommon doesn't do anything with Member

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::OP\_ProfileCallCommon(const unaligned T \* playout, RecyclableObject \* function, unsigned flags, ProfileId profileId, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices)

{

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

FunctionInfo\* functionInfo = function->GetTypeId() == TypeIds\_Function?

JavascriptFunction::FromVar(function)->GetFunctionInfo() : nullptr;

dynamicProfileInfo->RecordCallSiteInfo(functionBody, profileId, functionInfo, functionInfo ? static\_cast<JavascriptFunction\*>(function) : nullptr, playout->ArgCount, false, inlineCacheIndex);

OP\_CallCommon<T>(playout, function, flags, spreadIndices);

if (playout->Return != Js::Constants::NoRegister)

{

dynamicProfileInfo->RecordReturnTypeOnCallSiteInfo(functionBody, profileId, GetReg((RegSlot)playout->Return));

}

}

template <class T>

void InterpreterStackFrame::OP\_ProfileReturnTypeCallCommon(const unaligned T \* playout, RecyclableObject \* function, unsigned flags, ProfileId profileId, const Js::AuxArray<uint32> \*spreadIndices)

{

OP\_CallCommon<T>(playout, function, flags, spreadIndices);

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

if (playout->Return != Js::Constants::NoRegister)

{

dynamicProfileInfo->RecordReturnType(functionBody, profileId, GetReg((RegSlot)playout->Return));

}

}

#endif

template <class T>

void InterpreterStackFrame::OP\_CallPutCommon(const unaligned T \*playout, RecyclableObject \* function)

{

Arguments args(CallInfo(CallFlags\_None, playout->ArgCount), m\_outParams);

SetReg((RegSlot)playout->Return, function->InvokePut(args));

PopOut(playout->ArgCount);

}

template <class T>

void InterpreterStackFrame::OP\_CallPutCommonI(const unaligned T \*playout, RecyclableObject \* function)

{

OP\_CallPutCommon(playout, function);

}

template <class T>

void InterpreterStackFrame::OP\_GetRootProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

Js::Var instance = this->GetRootObject();

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

DynamicObject \* obj = DynamicObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if(CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, true, obj, propertyId, &value, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, value);

return;

}

OP\_GetRootProperty\_NoFastPath(playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetRootPropertyForTypeOf(unaligned T\* playout)

{

Var rootInstance = GetRootObject();

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetRootValueForTypeOf<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

DynamicObject::FromVar(rootInstance),

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(rootInstance, propertyId, value, /\*successful:\*/true);

}

#endif

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetRootProperty\_NoFastPath(unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var rootInstance = this->GetRootObject();

Var value = JavascriptOperators::PatchGetRootValue<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

DynamicObject::FromVar(rootInstance),

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(rootInstance, propertyId, value, /\*successful:\*/true);

}

#endif

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::UpdateFldInfoFlagsForGetSetInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj)

{

RecyclableObject \*callee = nullptr;

//TODO: Setter case once we stop sharing inline caches for these callsites.

if ((cacheType & (CacheType\_Getter | CacheType\_Setter)) && GetInlineCache(inlineCacheIndex)->GetGetterSetter(obj->GetType(), &callee))

{

const auto functionBody = this->m\_functionBody;

bool canInline = dynamicProfileInfo->RecordLdFldCallSiteInfo(functionBody, callee, false /\*callApplyTarget\*/);

if (canInline)

{

//updates this fldInfoFlags passed by reference.

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_InlineCandidate);

}

}

}

template <class T>

void InterpreterStackFrame::UpdateFldInfoFlagsForCallApplyInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj)

{

RecyclableObject \*callee = nullptr;

if (!(fldInfoFlags & FldInfo\_Polymorphic) && GetInlineCache(inlineCacheIndex)->GetCallApplyTarget(obj, &callee))

{

const auto functionBody = this->m\_functionBody;

bool canInline = dynamicProfileInfo->RecordLdFldCallSiteInfo(functionBody, callee, true /\*callApplyTarget\*/);

if (canInline)

{

//updates this fldInfoFlags passed by reference.

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_InlineCandidate);

}

}

}

template <class T, bool Root, bool Method, bool CallApplyTarget>

void InterpreterStackFrame::ProfiledGetProperty(unaligned T\* playout, const Var instance)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = ProfilingHelpers::ProfiledLdFld<Root, Method, CallApplyTarget>(

instance,

propertyId,

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetFunctionBody(),

instance);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, /\*successful:\*/true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetRootProperty(unaligned T\* playout)

{

ProfiledGetProperty<T, true, false, false>(playout, GetRootObject());

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetRootPropertyForTypeOf(unaligned T\* playout)

{

Var rootInstance = GetRootObject();

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = ProfilingHelpers::ProfiledLdFldForTypeOf<true, false, false>(

rootInstance,

propertyId,

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetFunctionBody());

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(rootInstance, propertyId, value, /\*successful:\*/true);

}

#endif

}

#endif

template <class T>

void InterpreterStackFrame::OP\_GetPropertyForTypeOf(unaligned T\* playout)

{

Var instance = GetReg(playout->Instance);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetValueForTypeOf<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, /\*successful:\*/true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_GetProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

OP\_GetProperty(instance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetLocalProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Var instance = this->localClosure;

OP\_GetProperty(instance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetProperty(Var instance, unaligned T\* playout)

{

InlineCache \*inlineCache = GetInlineCache(playout->inlineCacheIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (RecyclableObject::Is(instance))

{

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &value, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, value);

return;

}

}

OP\_GetProperty\_NoFastPath(instance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetSuperProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

Var thisInstance = GetReg(playout->Value2);

InlineCache \*inlineCache = GetInlineCache(playout->PropertyIdIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->PropertyIdIndex);

if (RecyclableObject::Is(instance) && RecyclableObject::Is(thisInstance))

{

RecyclableObject\* superObj = RecyclableObject::FromVar(instance);

RecyclableObject\* thisObj = RecyclableObject::FromVar(thisInstance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->PropertyIdIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

thisObj, false, superObj, propertyId, &value, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, value);

return;

}

}

SetReg(

playout->Value,

JavascriptOperators::PatchGetValueWithThisPtr<false>(

GetFunctionBody(),

GetInlineCache(playout->PropertyIdIndex),

playout->PropertyIdIndex,

GetReg(playout->Instance),

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetReg(playout->Value2)));

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetProperty\_NoFastPath(Var instance, unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetValue<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, true);

}

#endif

SetReg(playout->Value, value);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetProperty(unaligned T\* playout)

{

ProfiledGetProperty<T, false, false, false>(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetLocalProperty(unaligned T\* playout)

{

ProfiledGetProperty<T, false, false, false>(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetSuperProperty(unaligned T\* playout)

{

SetReg(

playout->Value,

ProfilingHelpers::ProfiledLdFld<false, false, false>(

GetReg(playout->Instance),

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetInlineCache(playout->PropertyIdIndex),

playout->PropertyIdIndex,

GetFunctionBody(),

GetReg(playout->Value2)));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetPropertyForTypeOf(unaligned T\* playout)

{

Var instance = GetReg(playout->Instance);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = ProfilingHelpers::ProfiledLdFldForTypeOf<false, false, false>(

instance,

propertyId,

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetFunctionBody()

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetPropertyCallApplyTarget(unaligned T\* playout)

{

ProfiledGetProperty<T, false, false, true>(playout, GetReg(playout->Instance));

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_GetPropertyScoped(const unaligned OpLayoutT\_ElementP<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Get the property, using a scope stack rather than an individual instance.

// Use the cache

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

FrameDisplay \*pScope = this->GetEnvForEvalCode();

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

int length = pScope->GetLength();

if ( 1 == length )

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(0);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &value, scriptContext, nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, value);

return;

}

}

OP\_GetPropertyScoped\_NoFastPath(playout);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <typename T>

void InterpreterStackFrame::OP\_GetPropertyForTypeOfScoped(const unaligned OpLayoutT\_ElementP<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Get the property, using a scope stack rather than an individual instance.

// Use the cache

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

FrameDisplay \*pScope = this->GetEnvForEvalCode();

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

int length = pScope->GetLength();

if (1 == length)

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(0);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &value, scriptContext, nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, value);

return;

}

}

SetReg(

playout->Value,

JavascriptOperators::PatchGetPropertyForTypeOfScoped<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetEnvForEvalCode(),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(Js::FunctionBody::RootObjectRegSlot)));

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <typename T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetPropertyScoped\_NoFastPath(const unaligned OpLayoutT\_ElementP<T>\* playout)

{

// Implicit root object as default instance

Var defaultInstance = GetReg(Js::FunctionBody::RootObjectRegSlot);

// PatchGetPropertyScoped doesn't update type and slotIndex if the scope is not an array of length 1.

SetReg(

playout->Value,

JavascriptOperators::PatchGetPropertyScoped<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetEnvForEvalCode(),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

defaultInstance));

}

template <class T>

void InterpreterStackFrame::OP\_SetPropertyScoped(unaligned T\* playout, PropertyOperationFlags flags)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Set the property, using a scope stack rather than an individual instance.

// Use the cache

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

FrameDisplay \*pScope = this->GetEnvForEvalCode();

InlineCache \*inlineCache = GetInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

Var value = GetReg(playout->Value);

DynamicObject \*obj;

int length = pScope->GetLength();

if ( 1 == length )

{

obj = (DynamicObject\*)pScope->GetItem(0);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

if (CacheOperators::TrySetProperty<true, false, false, false, false, true, false, false>(

obj, false, propertyId, value, scriptContext, flags, nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return;

}

}

OP\_SetPropertyScoped\_NoFastPath(playout, flags);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_SetPropertyScoped\_NoFastPath(unaligned T\* playout, PropertyOperationFlags flags)

{

// Implicit root object as default instance

Var defaultInstance = GetReg(Js::FunctionBody::RootObjectRegSlot);

// PatchSetPropertyScoped doesn't update type and slotIndex if the scope is not an array of length 1.

JavascriptOperators::PatchSetPropertyScoped<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetEnvForEvalCode(),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(playout->Value),

defaultInstance,

flags);

}

template <class T>

void InterpreterStackFrame::OP\_SetPropertyScopedStrict(unaligned T\* playout)

{

OP\_SetPropertyScoped(playout, PropertyOperation\_StrictMode);

}

template <class T>

void InterpreterStackFrame::OP\_ConsoleSetPropertyScoped(unaligned T\* playout)

{

OP\_SetPropertyScoped(playout, PropertyOperation\_AllowUndeclInConsoleScope);

}

template <class T>

\_\_inline bool InterpreterStackFrame::TrySetPropertyLocalFastPath(unaligned T\* playout, PropertyId pid, Var instance, InlineCache\*& inlineCache, PropertyOperationFlags flags)

{

Assert(!TaggedNumber::Is(instance));

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

return

CacheOperators::TrySetProperty<true, false, false, false, false, true, false, false>(

obj,

!!(flags & PropertyOperation\_Root),

pid,

GetReg(playout->Value),

GetScriptContext(),

flags,

nullptr,

&info);

}

template <class T>

\_\_inline void InterpreterStackFrame::DoSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

// Same fast path as in the backend.

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

InlineCache \*inlineCache;

if (!TaggedNumber::Is(instance)

&& TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

if(GetJavascriptFunction()->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

return;

}

DoSetProperty\_NoFastPath(playout, instance, flags);

}

template <class T>

\_\_inline void InterpreterStackFrame::DoSetSuperProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

DoSetSuperProperty\_NoFastPath(playout, instance, flags);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::DoSetProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

InlineCache \*const inlineCache = GetInlineCache(playout->inlineCacheIndex);

const auto PatchPutRootValue = &JavascriptOperators::PatchPutRootValueNoLocalFastPath<false, InlineCache>;

const auto PatchPutValue = &JavascriptOperators::PatchPutValueNoLocalFastPath<false, InlineCache>;

const auto PatchPut = flags & PropertyOperation\_Root ? PatchPutRootValue : PatchPutValue;

PatchPut(

GetFunctionBody(),

inlineCache,

playout->inlineCacheIndex,

instance,

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(playout->Value),

flags);

if(!TaggedNumber::Is(instance) && GetJavascriptFunction()->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::DoSetSuperProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

InlineCache \*const inlineCache = GetInlineCache(playout->PropertyIdIndex);

JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<false, InlineCache>(

GetFunctionBody(),

inlineCache,

playout->PropertyIdIndex,

instance,

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetReg(playout->Value),

GetReg(playout->Value2),

flags);

if (!TaggedNumber::Is(instance) && GetJavascriptFunction()->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

}

#if ENABLE\_PROFILE\_INFO

template <class T, bool Root>

void InterpreterStackFrame::ProfiledSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

Assert(!Root || flags & PropertyOperation\_Root);

ProfilingHelpers::ProfiledStFld<Root>(

instance,

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetReg(playout->Value),

flags,

GetJavascriptFunction(),

instance);

}

template <class T, bool Root>

void InterpreterStackFrame::ProfiledSetSuperProperty(unaligned T\* playout, Var instance, Var thisInstance, PropertyOperationFlags flags)

{

Assert(!Root || flags & PropertyOperation\_Root);

ProfilingHelpers::ProfiledStFld<Root>(

instance,

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetInlineCache(playout->PropertyIdIndex),

playout->PropertyIdIndex,

GetReg(playout->Value),

flags,

GetJavascriptFunction(),

thisInstance);

}

#endif

template <class T>

void InterpreterStackFrame::OP\_SetProperty(unaligned T\* playout)

{

DoSetProperty(playout, GetReg(playout->Instance), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_SetLocalProperty(unaligned T\* playout)

{

DoSetProperty(playout, this->localClosure, PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_SetSuperProperty(unaligned T\* playout)

{

DoSetSuperProperty(playout, GetReg(playout->Instance), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetProperty(unaligned T\* playout)

{

ProfiledSetProperty<T, false>(playout, GetReg(playout->Instance), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetLocalProperty(unaligned T\* playout)

{

ProfiledSetProperty<T, false>(playout, this->localClosure, PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetSuperProperty(unaligned T\* playout)

{

ProfiledSetSuperProperty<T, false>(playout, GetReg(playout->Instance), GetReg(playout->Value2), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_SetRootProperty(unaligned T\* playout)

{

DoSetProperty(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetRootProperty(unaligned T\* playout)

{

ProfiledSetProperty<T, true>(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_SetPropertyStrict(unaligned T\* playout)

{

DoSetProperty(playout, GetReg(playout->Instance), PropertyOperation\_StrictMode);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetPropertyStrict(unaligned T\* playout)

{

ProfiledSetProperty<T, false>(playout, GetReg(playout->Instance), PropertyOperation\_StrictMode);

}

template <class T>

void InterpreterStackFrame::OP\_SetRootPropertyStrict(unaligned T\* playout)

{

DoSetProperty(playout, this->GetRootObject(), PropertyOperation\_StrictModeRoot);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetRootPropertyStrict(unaligned T\* playout)

{

ProfiledSetProperty<T, true>(playout, this->GetRootObject(), PropertyOperation\_StrictModeRoot);

}

#if ENABLE\_PROFILE\_INFO

template <bool doProfile>

Var InterpreterStackFrame::ProfiledDivide(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

Var result = JavascriptMath::Divide(aLeft, aRight,scriptContext);

if (doProfile)

{

Js::FunctionBody\* body = this->m\_functionBody;

body->GetDynamicProfileInfo()->RecordDivideResultType(body, profileId, result);

}

return result;

}

template <bool doProfile>

Var InterpreterStackFrame::ProfileModulus(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

// If both arguments are TaggedInt, then try to do integer division

// This case is not handled by the lowerer.

if (doProfile)

{

Js::FunctionBody\* body = this->function->GetFunctionBody();

if(TaggedInt::IsPair(aLeft, aRight))

{

int nLeft = TaggedInt::ToInt32(aLeft);

int nRight = TaggedInt::ToInt32(aRight);

// nLeft is positive and nRight is +2^i

// Fast path for Power of 2 divisor

if (nLeft > 0 && ::Math::IsPow2(nRight))

{

body->GetDynamicProfileInfo()->RecordModulusOpType(body, profileId, /\*isModByPowerOf2\*/ true);

return TaggedInt::ToVarUnchecked(nLeft & (nRight - 1));

}

}

body->GetDynamicProfileInfo()->RecordModulusOpType(body, profileId, /\*isModByPowerOf2\*/ false);

}

return JavascriptMath::Modulus(aLeft, aRight,scriptContext);

}

template <bool doProfile>

Var InterpreterStackFrame::ProfiledSwitch(Var exp, ProfileId profileId)

{

if (doProfile)

{

Js::FunctionBody\* body = this->m\_functionBody;

body->GetDynamicProfileInfo()->RecordSwitchType(body, profileId, exp);

}

return exp;

}

#else

template <bool doProfile>

Var InterpreterStackFrame::ProfiledDivide(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

Assert(!doProfile);

return JavascriptMath::Divide(aLeft, aRight, scriptContext);

}

template <bool doProfile>

Var InterpreterStackFrame::ProfileModulus(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

Assert(!doProfile);

return JavascriptMath::Modulus(aLeft, aRight, scriptContext);

}

template <bool doProfile>

Var InterpreterStackFrame::ProfiledSwitch(Var exp, ProfileId profileId)

{

Assert(!doProfile);

return exp;

}

#endif

template <class T>

void InterpreterStackFrame::DoInitProperty(unaligned T\* playout, Var instance)

{

// Same fast path as in the backend.

InlineCache \*inlineCache = nullptr;

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache))

{

return;

}

DoInitProperty\_NoFastPath(playout, instance);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::DoInitProperty\_NoFastPath(unaligned T\* playout, Var instance)

{

JavascriptOperators::PatchInitValue<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

RecyclableObject::FromVar(instance),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMember(const unaligned T \* playout)

{

uint inlineCacheIndex = playout->inlineCacheIndex;

InlineCache \* inlineCache = this->GetInlineCache(inlineCacheIndex);

Var instance = GetReg(playout->Instance);

PropertyOperationFlags flags = PropertyOperation\_None;

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (!TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

JavascriptOperators::OP\_InitClassMember(instance, propertyId, GetReg(playout->Value));

}

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberGet(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberGet(

GetReg(playout->Instance),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberSet(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberSet(

GetReg(playout->Instance),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberSetComputedName(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberSetComputedName(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberGetComputedName(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberGetComputedName(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberComputedName(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberComputedName(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext());

}

template <class T>

void InterpreterStackFrame::DoInitLetFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags)

{

uint inlineCacheIndex = playout->inlineCacheIndex;

InlineCache \* inlineCache = this->GetInlineCache(inlineCacheIndex);

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (!TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

JavascriptOperators::OP\_InitLetProperty(instance, propertyId, GetReg(playout->Value));

}

}

template <class T>

void InterpreterStackFrame::DoInitConstFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags)

{

uint inlineCacheIndex = playout->inlineCacheIndex;

InlineCache \* inlineCache = this->GetInlineCache(inlineCacheIndex);

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (!TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

JavascriptOperators::OP\_InitConstProperty(instance, propertyId, GetReg(playout->Value));

}

}

template <class T>

void InterpreterStackFrame::OP\_InitProperty(unaligned T\* playout)

{

DoInitProperty(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_InitLocalProperty(unaligned T\* playout)

{

DoInitProperty(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_InitInnerFld(const unaligned T\* playout)

{

DoInitProperty(playout, InnerScopeFromIndex(playout->scopeIndex));

}

template <class T>

void InterpreterStackFrame::OP\_InitLetFld(const unaligned T \* playout)

{

DoInitLetFld(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_InitInnerLetFld(const unaligned T \* playout)

{

DoInitLetFld(playout, InnerScopeFromIndex(playout->scopeIndex));

}

template <class T>

void InterpreterStackFrame::OP\_InitLocalLetFld(const unaligned T \* playout)

{

DoInitLetFld(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_InitConstFld(const unaligned T \* playout)

{

DoInitConstFld(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_InitRootProperty(unaligned T\* playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

DoInitProperty(playout, this->GetRootObject());

}

template <class T>

void InterpreterStackFrame::OP\_InitRootLetFld(const unaligned T \* playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

DoInitLetFld(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_InitRootConstFld(const unaligned T \* playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

DoInitConstFld(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclLetProperty(unaligned T\* playout)

{

Var instance = InnerScopeFromIndex(playout->scopeIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitLetProperty(instance, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclLocalLetProperty(unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitLetProperty(this->localClosure, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

void InterpreterStackFrame::OP\_InitUndeclRootLetProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

PropertyId propertyId = this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex);

JavascriptOperators::OP\_InitUndeclRootLetProperty(instance, propertyId);

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclConstProperty(unaligned T\* playout)

{

Var instance = InnerScopeFromIndex(playout->scopeIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitConstProperty(instance, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclLocalConstProperty(unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitConstProperty(this->localClosure, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

void InterpreterStackFrame::OP\_InitUndeclRootConstProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

PropertyId propertyId = this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex);

JavascriptOperators::OP\_InitUndeclRootConstProperty(instance, propertyId);

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclConsoleLetProperty(unaligned T\* playout)

{

FrameDisplay\* pScope = (FrameDisplay\*)this->LdEnv();

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

PropertyId propertyId = m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex);

JavascriptOperators::OP\_InitLetProperty(pScope->GetItem(0), propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclConsoleConstProperty(unaligned T\* playout)

{

FrameDisplay\* pScope = (FrameDisplay\*)this->LdEnv();

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

PropertyId propertyId = m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex);

JavascriptOperators::OP\_InitConstProperty(pScope->GetItem(0), propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::ProfiledInitProperty(unaligned T\* playout, Var instance)

{

ProfilingHelpers::ProfiledInitFld(

RecyclableObject::FromVar(instance),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetReg(playout->Value),

GetFunctionBody());

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledInitProperty(unaligned T\* playout)

{

ProfiledInitProperty(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledInitLocalProperty(unaligned T\* playout)

{

ProfiledInitProperty(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledInitRootProperty(unaligned T\* playout)

{

ProfiledInitProperty(playout, this->GetRootObject());

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetElementI(const unaligned OpLayoutDynamicProfile<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

SetReg(

playout->Value,

ProfilingHelpers::ProfiledLdElem(

GetReg(playout->Instance),

GetReg(playout->Element),

m\_functionBody,

playout->profileId));

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_GetElementI(const unaligned T\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

// Only enable fast path if the javascript array is not cross site

Var element;

#if ENABLE\_PROFILE\_INFO

if (!TaggedNumber::Is(instance) && VirtualTableInfo<JavascriptArray>::HasVirtualTable(instance))

{

element =

ProfilingHelpers::ProfiledLdElem\_FastPath(

JavascriptArray::FromVar(instance),

GetReg(playout->Element),

GetScriptContext());

}

else

#endif

{

element = JavascriptOperators::OP\_GetElementI(instance, GetReg(playout->Element), GetScriptContext());

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, element);

}

template <typename T>

void InterpreterStackFrame::OP\_SetElementI(const unaligned T\* playout, PropertyOperationFlags flags)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

const Var varIndex = GetReg(playout->Element);

const Var value = GetReg(playout->Value);

#if ENABLE\_PROFILE\_INFO

// Only enable fast path if the javascript array is not cross site

if (!TaggedNumber::Is(instance) &&

VirtualTableInfo<JavascriptArray>::HasVirtualTable(instance) &&

!JavascriptOperators::SetElementMayHaveImplicitCalls(GetScriptContext()))

{

ProfilingHelpers::ProfiledStElem\_FastPath(

JavascriptArray::FromVar(instance),

varIndex,

value,

GetScriptContext(),

flags);

}

else

#endif

{

JavascriptOperators::OP\_SetElementI(instance, varIndex, value, GetScriptContext(), flags);

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#if ENABLE\_PROFILE\_INFO

template <typename T>

void InterpreterStackFrame::OP\_ProfiledSetElementI(

const unaligned OpLayoutDynamicProfile<T>\* playout,

PropertyOperationFlags flags)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

ProfilingHelpers::ProfiledStElem(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody,

playout->profileId,

flags);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_SetElementIStrict(const unaligned T\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

OP\_SetElementI(playout, PropertyOperation\_StrictMode);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#if ENABLE\_PROFILE\_INFO

template <typename T>

void InterpreterStackFrame::OP\_ProfiledSetElementIStrict(const unaligned OpLayoutDynamicProfile<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

OP\_ProfiledSetElementI(playout, PropertyOperation\_StrictMode);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#endif

template <class T>

void InterpreterStackFrame::OP\_LdArrayHeadSegment(const unaligned T\* playout)

{

JavascriptArray\* array = JavascriptArray::FromAnyArray(GetReg(playout->R1));

// The array is create by the built-in on the same script context

Assert(array->GetScriptContext() == GetScriptContext());

SetNonVarReg(playout->R0, array->GetHead());

}

template <class T>

void InterpreterStackFrame::OP\_SetArraySegmentItem\_CI4(const unaligned T\* playout)

{

SparseArraySegment<Var> \* segment = (SparseArraySegment<Var> \*)GetNonVarReg(playout->Instance);

uint32 index = playout->Element;

Var value = GetReg(playout->Value);

Assert(segment->left == 0);

Assert(index < segment->length);

segment->elements[index] = value;

}

template <class T>

void InterpreterStackFrame::OP\_NewScArray(const unaligned T \* playout)

{

JavascriptArray \*arr;

arr = scriptContext->GetLibrary()->CreateArrayLiteral(playout->C1);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#if ENABLE\_PROFILE\_INFO

template <bool Profiled, class T>

void InterpreterStackFrame::ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout)

{

if(!Profiled && !isAutoProfiling)

{

OP\_NewScArray(playout);

return;

}

SetReg(

playout->R0,

ProfilingHelpers::ProfiledNewScArray(

playout->C1,

m\_functionBody,

playout->profileId));

}

#else

template <bool Profiled, class T>

void InterpreterStackFrame::ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout)

{

Assert(!Profiled);

OP\_NewScArray(playout);

}

#endif

void InterpreterStackFrame::OP\_NewScIntArray(const unaligned OpLayoutAuxiliary \* playout)

{

#if ENABLE\_PROFILE\_INFO

if(isAutoProfiling)

{

OP\_ProfiledNewScIntArray(static\_cast<const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \*>(playout));

return;

}

#endif

const Js::AuxArray<int32> \*ints = Js::ByteCodeReader::ReadAuxArray<int32>(playout->Offset, this->GetFunctionBody());

JavascriptNativeIntArray \*arr = scriptContext->GetLibrary()->CreateNativeIntArrayLiteral(ints->count);

SparseArraySegment<int32> \* segment = (SparseArraySegment<int32>\*)arr->GetHead();

JavascriptOperators::AddIntsToArraySegment(segment, ints);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#if ENABLE\_PROFILE\_INFO

void InterpreterStackFrame::OP\_ProfiledNewScIntArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout)

{

const Js::AuxArray<int32> \*ints = Js::ByteCodeReader::ReadAuxArray<int32>(playout->Offset, this->GetFunctionBody());

Js::ProfileId profileId = playout->profileId;

FunctionBody \*functionBody = this->m\_functionBody;

ArrayCallSiteInfo \*arrayInfo = functionBody->GetDynamicProfileInfo()->GetArrayCallSiteInfo(functionBody, profileId);

Assert(arrayInfo);

JavascriptArray \*arr;

JavascriptLibrary \*lib = scriptContext->GetLibrary();

if (arrayInfo && arrayInfo->IsNativeIntArray())

{

#if ENABLE\_COPYONACCESS\_ARRAY

if (JavascriptLibrary::IsCopyOnAccessArrayCallSite(lib, arrayInfo, ints->count))

{

Assert(lib->cacheForCopyOnAccessArraySegments);

arr = scriptContext->GetLibrary()->CreateCopyOnAccessNativeIntArrayLiteral(arrayInfo, functionBody, ints);

}

else

#endif

{

arr = scriptContext->GetLibrary()->CreateNativeIntArrayLiteral(ints->count);

SparseArraySegment<int32> \*segment = (SparseArraySegment<int32>\*)arr->GetHead();

JavascriptOperators::AddIntsToArraySegment(segment, ints);

}

JavascriptNativeIntArray \*intArray = reinterpret\_cast<JavascriptNativeIntArray\*>(arr);

Recycler \*recycler = scriptContext->GetRecycler();

intArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

}

else if (arrayInfo && arrayInfo->IsNativeFloatArray())

{

arr = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(ints->count);

SparseArraySegment<double> \* segment = (SparseArraySegment<double>\*)arr->GetHead();

for (uint i = 0; i < ints->count; i++)

{

segment->elements[i] = (double)ints->elements[i];

}

JavascriptNativeFloatArray \*floatArray = reinterpret\_cast<JavascriptNativeFloatArray\*>(arr);

Recycler \*recycler = scriptContext->GetRecycler();

floatArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

}

else

{

arr = scriptContext->GetLibrary()->CreateArrayLiteral(ints->count);

SparseArraySegment<Var> \* segment = (SparseArraySegment<Var>\*)arr->GetHead();

for (uint i = 0; i < ints->count; i++)

{

segment->elements[i] = JavascriptNumber::ToVar(ints->elements[i], scriptContext);

}

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#endif

void InterpreterStackFrame::OP\_NewScFltArray(const unaligned OpLayoutAuxiliary \* playout )

{

#if ENABLE\_PROFILE\_INFO

if(isAutoProfiling)

{

OP\_ProfiledNewScFltArray(static\_cast<const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \*>(playout));

return;

}

#endif

const Js::AuxArray<double> \*doubles = Js::ByteCodeReader::ReadAuxArray<double>(playout->Offset, this->GetFunctionBody());

JavascriptNativeFloatArray \*arr = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(doubles->count);

SparseArraySegment<double> \* segment = (SparseArraySegment<double>\*)arr->GetHead();

JavascriptOperators::AddFloatsToArraySegment(segment, doubles);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#if ENABLE\_PROFILE\_INFO

void InterpreterStackFrame::OP\_ProfiledNewScFltArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout)

{

const Js::AuxArray<double> \*doubles = Js::ByteCodeReader::ReadAuxArray<double>(playout->Offset, this->GetFunctionBody());

Js::ProfileId profileId = playout->profileId;

FunctionBody \*functionBody = this->m\_functionBody;

ArrayCallSiteInfo \*arrayInfo = functionBody->GetDynamicProfileInfo()->GetArrayCallSiteInfo(functionBody, profileId);

Assert(arrayInfo);

JavascriptArray \*arr;

if (arrayInfo && arrayInfo->IsNativeFloatArray())

{

arrayInfo->SetIsNotNativeIntArray();

arr = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(doubles->count);

SparseArraySegment<double> \* segment = (SparseArraySegment<double>\*)arr->GetHead();

JavascriptOperators::AddFloatsToArraySegment(segment, doubles);

JavascriptNativeFloatArray \*floatArray = reinterpret\_cast<JavascriptNativeFloatArray\*>(arr);

Recycler \*recycler = scriptContext->GetRecycler();

floatArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

}

else

{

arr = scriptContext->GetLibrary()->CreateArrayLiteral(doubles->count);

SparseArraySegment<Var> \* segment = (SparseArraySegment<Var>\*)arr->GetHead();

for (uint i = 0; i < doubles->count; i++)

{

segment->elements[i] = JavascriptNumber::ToVarNoCheck(doubles->elements[i], scriptContext);

}

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#endif

void InterpreterStackFrame::OP\_SetArraySegmentVars(const unaligned OpLayoutAuxiliary \* playout)

{

const Js::VarArray \*vars = Js::ByteCodeReader::ReadAuxArray<Var>(playout->Offset, this->GetFunctionBody());

SparseArraySegment<Var> \* segment =(SparseArraySegment<Var> \*)GetNonVarReg(playout->R0);

JavascriptOperators::AddVarsToArraySegment(segment, vars);

}

template <class T>

void InterpreterStackFrame::OP\_SetArrayItemC\_CI4(const unaligned T\* playout)

{

JavascriptArray\* array = JavascriptArray::FromAnyArray(GetReg(playout->Instance));

uint32 index = playout->Element;

Var value = GetReg(playout->Value);

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(value);

#endif

// The array is create by the built-in on the same script context

Assert(array->GetScriptContext() == GetScriptContext());

TypeId typeId = array->GetTypeId();

if (typeId == TypeIds\_NativeIntArray)

{

JavascriptArray::OP\_SetNativeIntElementC(reinterpret\_cast<JavascriptNativeIntArray\*>(array), index, value, array->GetScriptContext());

}

else if (typeId == TypeIds\_NativeFloatArray)

{

JavascriptArray::OP\_SetNativeFloatElementC(reinterpret\_cast<JavascriptNativeFloatArray\*>(array), index, value, array->GetScriptContext());

}

else

{

array->SetArrayLiteralItem(index, value);

}

}

template <class T>

void InterpreterStackFrame::OP\_SetArrayItemI\_CI4(const unaligned T\* playout)

{

// Note that this code assumes that we only get here when we see an array literal,

// so we know that the instance is truly an array, and the index is a uint32.

// If/when we use this for cases like "a[0] = x", we'll at least have to check

// whether "a" is really an array.

JavascriptArray\* array = JavascriptArray::FromAnyArray(GetReg(playout->Instance));

// The array is create by the built-in on the same script context

Assert(array->GetScriptContext() == GetScriptContext());

uint32 index = playout->Element;

Var value = GetReg(playout->Value);

Assert(VirtualTableInfo<JavascriptArray>::HasVirtualTable(array));

SparseArraySegment<Var>\* lastUsedSeg = (SparseArraySegment<Var>\*)array->GetLastUsedSegment();

if (index < lastUsedSeg->left)

{

goto helper;

}

uint32 index2 = index - lastUsedSeg->left;

if (index2 < lastUsedSeg->size)

{

// Successful fastpath

array->DirectSetItemInLastUsedSegmentAt(index2, value);

return;

}

helper:

ScriptContext\* scriptContext = array->GetScriptContext();

JavascriptOperators::SetItem(array, array, index, value, scriptContext);

}

#if ENABLE\_PROFILE\_INFO

Var InterpreterStackFrame::OP\_ProfiledLdThis(Var thisVar, int moduleID, ScriptContext \*scriptContext)

{

FunctionBody \* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (JavascriptOperators::IsThisSelf(typeId))

{

Assert(typeId != TypeIds\_GlobalObject || ((Js::GlobalObject\*)thisVar)->ToThis() == thisVar);

Assert(typeId != TypeIds\_ModuleRoot || JavascriptOperators::GetThisFromModuleRoot(thisVar) == thisVar);

// Record the fact that we saw a trivial LdThis.

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Simple);

return thisVar;

}

thisVar = JavascriptOperators::OP\_GetThis(thisVar, moduleID, scriptContext);

// Record the fact that we saw a LdThis that had to map its source to something else, or at least

// forced us to call a helper, e.g., a FastDOM object with an unrecognized type ID.

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Mapped);

return thisVar;

}

Var InterpreterStackFrame::OP\_ProfiledStrictLdThis(Var thisVar, ScriptContext\* scriptContext)

{

FunctionBody \* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (typeId == TypeIds\_ActivationObject)

{

thisVar = scriptContext->GetLibrary()->GetUndefined();

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Mapped);

return thisVar;

}

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Simple);

return thisVar;

}

#endif

void InterpreterStackFrame::OP\_InitCachedFuncs(const unaligned OpLayoutAuxNoReg \* playout)

{

const FuncInfoArray \*info = Js::ByteCodeReader::ReadAuxArray<FuncInfoEntry>(playout->Offset, this->GetFunctionBody());

JavascriptOperators::OP\_InitCachedFuncs(this->localClosure, GetLocalFrameDisplay(), info, GetScriptContext());

}

Var InterpreterStackFrame::OP\_GetCachedFunc(Var instance, int32 index)

{

ActivationObjectEx \*obj = (ActivationObjectEx\*)ActivationObjectEx::FromVar(instance);

FuncCacheEntry \*entry = obj->GetFuncCacheEntry((uint)index);

return entry->func;

}

void InterpreterStackFrame::OP\_CommitScope(const unaligned OpLayoutAuxNoReg \* playout)

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

this->OP\_CommitScopeHelper(playout, propIds);

}

void InterpreterStackFrame::OP\_CommitScopeHelper(const unaligned OpLayoutAuxNoReg \*playout, const PropertyIdArray \*propIds)

{

ActivationObjectEx \*obj = (ActivationObjectEx\*)ActivationObjectEx::FromVar(/\*GetReg(playout->R0)\*/this->localClosure);

ScriptFunction \*func = obj->GetParentFunc();

Assert(obj->GetParentFunc() == func);

if (func->GetCachedScope() == obj)

{

PropertyId firstVarSlot = ActivationObjectEx::GetFirstVarSlot(propIds);

Var undef = scriptContext->GetLibrary()->GetUndefined();

for (uint i = firstVarSlot; i < propIds->count; i++)

{

obj->SetSlot(SetSlotArguments(propIds->elements[i], i, undef));

}

obj->SetCommit(true);

}

}

Var InterpreterStackFrame::OP\_NewScObjectSimple()

{

Var object = scriptContext->GetLibrary()->CreateObject(true);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(object));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

object = JavascriptProxy::AutoProxyWrapper(object);

}

#endif

return object;

}

void InterpreterStackFrame::OP\_NewScObjectLiteral(const unaligned OpLayoutAuxiliary \* playout )

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

Var newObj = JavascriptOperators::NewScObjectLiteral(GetScriptContext(), propIds,

this->GetFunctionBody()->GetObjectLiteralTypeRef(playout->C1));

SetReg(playout->R0, newObj);

}

void InterpreterStackFrame::OP\_NewScObjectLiteral\_LS(const unaligned OpLayoutAuxiliary \* playout, RegSlot& target)

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

target = playout->R0;

Var newObj = JavascriptOperators::NewScObjectLiteral(GetScriptContext(), propIds,

this->GetFunctionBody()->GetObjectLiteralTypeRef(playout->C1));

SetReg(playout->R0, newObj);

target = Js::Constants::NoRegister;

}

void InterpreterStackFrame::OP\_LdPropIds(const unaligned OpLayoutAuxiliary \* playout)

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

SetNonVarReg(playout->R0, (Var)propIds);

}

bool InterpreterStackFrame::IsCurrentLoopNativeAddr(void \* codeAddr) const

{

if (this->GetCurrentLoopNum() == LoopHeader::NoLoop)

{

return false;

}

// TODO: Do more verification?

return true;

}

#if ENABLE\_PROFILE\_INFO

void InterpreterStackFrame::OP\_RecordImplicitCall(uint loopNumber)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

Assert(loopNumber < this->m\_functionBody->GetLoopCount());

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

ThreadContext \* threadContext = scriptContext->GetThreadContext();

dynamicProfileInfo->RecordLoopImplicitCallFlags(functionBody, loopNumber, threadContext->GetImplicitCallFlags());

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopStart(const byte \* ip)

{

const uint32 C1 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

if(!profiled && !isAutoProfiling)

{

return ip;

}

ThreadContext \*const threadContext = GetScriptContext()->GetThreadContext();

threadContext->IncrementLoopDepth();

// Save the implicit call flags. The interpreter may switch to profiling mode during LoopBodyStart, so always do this.

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

this->savedLoopImplicitCallFlags[C1] = threadContext->GetImplicitCallFlags();

threadContext->SetImplicitCallFlags(ImplicitCall\_None);

this->currentLoopCounter = 0;

if(!profiled)

{

return ip;

}

LayoutSize localLayoutSize;

OpCode peekOp = m\_reader.PeekOp(ip, localLayoutSize);

Assert(peekOp != OpCode::LoopBodyStart);

if (peekOp == OpCode::ProfiledLoopBodyStart)

{

Assert(localLayoutSize == layoutSize);

ip += Js::OpCodeUtil::EncodedSize(peekOp, layoutSize);

// We are doing JIT loop body. Process the first ProfiledLoopBodyStart to avoid recording

// the implicit call before the first iteration

uint32 C2 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

Assert(C1 == C2);

(this->\*opProfiledLoopBodyStart)(C1, layoutSize, true /\* isFirstIteration \*/);

return m\_reader.GetIP();

}

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopEnd(const byte \* ip)

{

uint32 loopNumber = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

if(!profiled && !isAutoProfiling)

{

return ip;

}

this->CheckIfLoopIsHot(this->currentLoopCounter);

Js::FunctionBody \*fn = this->function->GetFunctionBody();

if (fn->HasDynamicProfileInfo())

{

fn->GetAnyDynamicProfileInfo()->SetLoopInterpreted(loopNumber);

// If the counter is 0, there is a high chance that some config disabled tracking that information. (ie: -off:jitloopbody)

// Assume it is valid for memop in this case.

if (this->currentLoopCounter >= (uint)CONFIG\_FLAG(MinMemOpCount) ||

(this->currentLoopCounter == 0 && !this->m\_functionBody->DoJITLoopBody())

)

{

// This flag becomes relevant only if the loop has been interpreted

fn->GetAnyDynamicProfileInfo()->SetMemOpMinReached(loopNumber);

}

}

this->currentLoopCounter = 0;

if (profiled)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

OP\_RecordImplicitCall(loopNumber);

if(switchProfileModeOnLoopEndNumber == loopNumber)

{

// Stop profiling since the jitted loop body would be exiting the loop

Assert(!switchProfileMode);

switchProfileMode = true;

switchProfileModeOnLoopEndNumber = 0u - 1;

}

}

// Restore the implicit call flags state and add with flags in the loop as well

ThreadContext \*const threadContext = GetScriptContext()->GetThreadContext();

threadContext->AddImplicitCallFlags(this->savedLoopImplicitCallFlags[loopNumber]);

threadContext->DecrementLoopDepth();

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopBodyStart(const byte \* ip)

{

uint32 C1 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

if(profiled || isAutoProfiling)

{

this->currentLoopCounter++;

}

if (profiled)

{

OP\_RecordImplicitCall(C1);

}

(this->\*(profiled ? opProfiledLoopBodyStart : opLoopBodyStart))(C1, layoutSize, false /\* isFirstIteration \*/);

return m\_reader.GetIP();

}

template<bool InterruptProbe, bool JITLoopBody>

void InterpreterStackFrame::ProfiledLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

if (InterruptProbe)

{

this->DoInterruptProbe();

}

if (!JITLoopBody || this->IsInCatchOrFinallyBlock())

{

// For functions having try-catch-finally, jit loop bodies for loops that are contained only in a try block,

// not even indirect containment in a Catch or Finally.

return;

}

LoopHeader const \* loopHeader = DoLoopBodyStart(loopNumber, layoutSize, false, isFirstIteration);

Assert(loopHeader == nullptr || this->m\_functionBody->GetLoopNumber(loopHeader) == loopNumber);

if (loopHeader != nullptr)

{

// We executed jitted loop body, no implicit call information available for this loop

uint currentOffset = m\_reader.GetCurrentOffset();

if (!loopHeader->Contains(currentOffset) || (m\_reader.PeekOp() == OpCode::ProfiledLoopEnd))

{

// Restore the outer loop's implicit call flags

scriptContext->GetThreadContext()->SetImplicitCallFlags(this->savedLoopImplicitCallFlags[loopNumber]);

}

else

{

// We bailout from the loop, just continue collect implicit call flags for this loop

}

}

}

#else

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopStart(const byte \* ip)

{

Assert(!profiled);

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopEnd(const byte \* ip)

{

Assert(!profiled);

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopBodyStart(const byte \* ip)

{

uint32 C1 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

Assert(!profiled);

(this->\*opLoopBodyStart)(C1, layoutSize, false /\* isFirstIteration \*/);

return m\_reader.GetIP();

}

#endif

template<bool InterruptProbe, bool JITLoopBody>

void InterpreterStackFrame::LoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration)

{

if (InterruptProbe)

{

this->DoInterruptProbe();

}

if (!JITLoopBody || this->IsInCatchOrFinallyBlock())

{

// For functions having try-catch-finally, jit loop bodies for loops that are contained only in a try block,

// not even indirect containment in a Catch or Finally.

return;

}

DoLoopBodyStart(loopNumber, layoutSize, true, isFirstIteration);

}

LoopHeader const \* InterpreterStackFrame::DoLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, const bool doProfileLoopCheck, const bool isFirstIteration)

{

#if ENABLE\_PROFILE\_INFO

class AutoRestoreLoopNumbers

{

private:

InterpreterStackFrame \*const interpreterStackFrame;

uint32 loopNumber;

bool doProfileLoopCheck;

public:

AutoRestoreLoopNumbers(InterpreterStackFrame \*const interpreterStackFrame, uint32 loopNumber, bool doProfileLoopCheck)

: interpreterStackFrame(interpreterStackFrame), loopNumber(loopNumber), doProfileLoopCheck(doProfileLoopCheck)

{

Assert(interpreterStackFrame->currentLoopNum == LoopHeader::NoLoop);

interpreterStackFrame->currentLoopNum = loopNumber;

interpreterStackFrame->m\_functionBody->SetRecentlyBailedOutOfJittedLoopBody(false);

}

~AutoRestoreLoopNumbers()

{

interpreterStackFrame->currentLoopNum = LoopHeader::NoLoop;

interpreterStackFrame->currentLoopCounter = 0;

Js::FunctionBody\* fn = interpreterStackFrame->m\_functionBody;

if (fn->RecentlyBailedOutOfJittedLoopBody())

{

if (doProfileLoopCheck && interpreterStackFrame->isAutoProfiling)

{

// Start profiling the loop after a bailout. Some bailouts require subsequent profile data collection such

// that the rejitted loop body would not bail out again for the same reason.

Assert(!interpreterStackFrame->switchProfileMode);

interpreterStackFrame->switchProfileMode = true;

Assert(interpreterStackFrame->switchProfileModeOnLoopEndNumber == 0u - 1);

interpreterStackFrame->switchProfileModeOnLoopEndNumber = loopNumber;

}

}

else

{

if (interpreterStackFrame->switchProfileModeOnLoopEndNumber == loopNumber)

{

// Stop profiling since the jitted loop body would be exiting the loop

Assert(!interpreterStackFrame->switchProfileMode);

interpreterStackFrame->switchProfileMode = true;

interpreterStackFrame->switchProfileModeOnLoopEndNumber = 0u - 1;

}

interpreterStackFrame->scriptContext->GetThreadContext()->DecrementLoopDepth();

}

}

};

#endif

Js::FunctionBody\* fn = this->m\_functionBody;

Assert(loopNumber < fn->GetLoopCount());

Assert(!this->IsInCatchOrFinallyBlock());

Js::LoopHeader \*loopHeader = fn->GetLoopHeader(loopNumber);

loopHeader->isInTry = (this->m\_flags & Js::InterpreterStackFrameFlags\_WithinTryBlock);

Js::LoopEntryPointInfo \* entryPointInfo = loopHeader->GetCurrentEntryPointInfo();

if (fn->ForceJITLoopBody() && loopHeader->interpretCount == 0 &&

(entryPointInfo != NULL && entryPointInfo->IsNotScheduled()))

{

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()))

{

scriptContext->GetThreadContext()->AddImplicitCallFlags(this->savedLoopImplicitCallFlags[loopNumber]);

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

GenerateLoopBody(scriptContext->GetNativeCodeGenerator(), fn, loopHeader, entryPointInfo, fn->GetLocalsCount(), this->m\_localSlots);

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

// If we have JITted the loop, call the JITted code

if (entryPointInfo != NULL && entryPointInfo->IsCodeGenDone())

{

#if DBG\_DUMP

if (PHASE\_TRACE1(Js::JITLoopBodyPhase) && CONFIG\_FLAG(Verbose))

{

fn->DumpFunctionId(true);

Output::Print(L": %-20s LoopBody Execute Loop: %2d\n", fn->GetDisplayName(), loopNumber);

Output::Flush();

}

loopHeader->nativeCount++;

#endif

#ifdef BGJIT\_STATS

entryPointInfo->MarkAsUsed();

#endif

entryPointInfo->EnsureIsReadyToCall();

RegSlot envReg = this->m\_functionBody->GetEnvReg();

if (envReg != Constants::NoRegister)

{

this->SetNonVarReg(envReg, this->LdEnv());

}

RegSlot localClosureReg = this->m\_functionBody->GetLocalClosureReg();

RegSlot localFrameDisplayReg = this->m\_functionBody->GetLocalFrameDisplayReg();

if (entryPointInfo->HasJittedStackClosure())

{

// The jitted code is expecting the closure registers to point to known stack locations where

// the closures can be found and possibly boxed.

// In a jitted loop body, those locations are the local closure fields on the interpreter instance.

if (localClosureReg != Constants::NoRegister)

{

this->SetNonVarReg(localClosureReg, &this->localClosure);

}

if (localFrameDisplayReg != Constants::NoRegister)

{

this->SetNonVarReg(localFrameDisplayReg, &this->localFrameDisplay);

}

}

else

{

// In non-stack-closure jitted code, the closure registers are expected to hold the addresses

// of the actual structures.

if (localClosureReg != Constants::NoRegister)

{

this->SetNonVarReg(localClosureReg, this->localClosure);

}

if (localFrameDisplayReg != Constants::NoRegister)

{

this->SetNonVarReg(localFrameDisplayReg, this->localFrameDisplay);

}

}

uint32 innerScopeCount = this->m\_functionBody->GetInnerScopeCount();

for (uint32 i = 0; i < innerScopeCount; i++)

{

// As with the function-level scope, transfer the inner scopes from the interpreter's side storage

// to their dedicated register slots.

SetNonVarReg(this->m\_functionBody->FirstInnerScopeReg() + i, InnerScopeFromIndex(i));

}

uint newOffset = 0;

if (fn->GetIsAsmJsFunction())

{

AutoRestoreLoopNumbers autoRestore(this, loopNumber, doProfileLoopCheck);

newOffset = this->CallAsmJsLoopBody((JavascriptMethod)entryPointInfo->address);

}

else

{

AutoRestoreLoopNumbers autoRestore(this, loopNumber, doProfileLoopCheck);

newOffset = this->CallLoopBody((JavascriptMethod)entryPointInfo->address);

}

if (envReg != Constants::NoRegister)

{

SetNonVarReg(envReg, nullptr);

}

if (localClosureReg != Constants::NoRegister)

{

SetNonVarReg(localClosureReg, nullptr);

}

if (localFrameDisplayReg != Constants::NoRegister)

{

SetNonVarReg(localFrameDisplayReg, nullptr);

}

for (uint32 i = 0; i < innerScopeCount; i++)

{

// Get the (possibly updated) scopes from their registers and put them back in side storage.

// (Getting the updated values may not be necessary, actually, but it can't hurt.)

// Then null out the registers.

RegSlot reg = this->m\_functionBody->FirstInnerScopeReg() + i;

SetInnerScopeFromIndex(i, GetNonVarReg(reg));

SetNonVarReg(reg, nullptr);

}

Assert(Js::OpCodeUtil::GetOpCodeLayout(OpCode::ProfiledLoopBodyStart) == Js::OpLayoutType::Unsigned1);

Assert(Js::OpCodeUtil::GetOpCodeLayout(OpCode::LoopBodyStart) == Js::OpLayoutType::Unsigned1);

Assert(Js::OpCodeUtil::EncodedSize(Js::OpCode::LoopBodyStart, layoutSize) == Js::OpCodeUtil::EncodedSize(Js::OpCode::ProfiledLoopBodyStart, layoutSize));

uint byteCodeSize = Js::OpCodeUtil::EncodedSize(Js::OpCode::LoopBodyStart, layoutSize);

if (layoutSize == SmallLayout)

{

byteCodeSize += sizeof(OpLayoutUnsigned1\_Small);

}

else if (layoutSize == MediumLayout)

{

byteCodeSize += sizeof(OpLayoutUnsigned1\_Medium);

}

else

{

byteCodeSize += sizeof(OpLayoutUnsigned1\_Large);

}

if (newOffset == loopHeader->startOffset || newOffset == m\_reader.GetCurrentOffset() - byteCodeSize)

{

// If we bail out back the start of the loop, or start of this LoopBodyStart just skip and interpret the loop

// instead of trying to start the loop body again

// Increment the interpret count of the loop

loopHeader->interpretCount++;

}

else

{

// we do not support this in asmjs, need to add support in IrBuilderAsmjs if we need this support for asmjs

if (!entryPointInfo->GetIsAsmJSFunction())

{

this->CheckIfLoopIsHot(loopHeader->profiledLoopCounter);

}

m\_reader.SetCurrentOffset(newOffset);

}

return loopHeader;

}

#endif

// Increment the interpret count of the loop

loopHeader->interpretCount += !isFirstIteration;

const uint loopInterpretCount = GetFunctionBody()->GetLoopInterpretCount(loopHeader);

if (loopHeader->interpretCount > loopInterpretCount)

{

if (this->scriptContext->GetConfig()->IsNoNative())

{

return nullptr;

}

if (!fn->DoJITLoopBody())

{

return nullptr;

}

#if ENABLE\_NATIVE\_CODEGEN

// If the job is not scheduled then we need to schedule it now.

// It is possible a job was scheduled earlier and we find ourselves looking at the same entry point

// again. For example, if the function with the loop was JITed and bailed out then as we finish

// the call in the interpreter we might encounter a loop for which we had scheduled a JIT job before

// the function was initially scheduled. In such cases, that old JIT job will complete. If it completes

// successfully then we can go ahead and use it. If it fails then it will eventually revert to the

// NotScheduled state. Since transitions from NotScheduled can only occur on the main thread,

// by checking the state we are safe from racing with the JIT thread when looking at the other fields

// of the entry point.

if (entryPointInfo != NULL && entryPointInfo->IsNotScheduled())

{

GenerateLoopBody(scriptContext->GetNativeCodeGenerator(), fn, loopHeader, entryPointInfo, fn->GetLocalsCount(), this->m\_localSlots);

}

#endif

}

#if ENABLE\_PROFILE\_INFO

else if(

doProfileLoopCheck &&

isAutoProfiling &&

loopHeader->interpretCount > fn->GetLoopProfileThreshold(loopInterpretCount))

{

// Start profiling the loop so that the jitted loop body will have some profile data to use

Assert(!switchProfileMode);

switchProfileMode = true;

Assert(switchProfileModeOnLoopEndNumber == 0u - 1);

switchProfileModeOnLoopEndNumber = loopNumber;

}

#endif

return nullptr;

}

void

InterpreterStackFrame::CheckIfLoopIsHot(uint profiledLoopCounter)

{

Js::FunctionBody \*fn = this->function->GetFunctionBody();

if (!fn->GetHasHotLoop() && profiledLoopCounter > (uint)CONFIG\_FLAG(JitLoopBodyHotLoopThreshold))

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if(PHASE\_TRACE(Js::JITLoopBodyPhase, fn))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(

L"Speculate Jit set for this function with loopbody: function: %s (%s)\n",

fn->GetDisplayName(),

fn->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

#endif

fn->SetHasHotLoop();

}

}

bool InterpreterStackFrame::CheckAndResetImplicitCall(DisableImplicitFlags prevDisableImplicitFlags, ImplicitCallFlags savedImplicitCallFlags)

{

ImplicitCallFlags curImplicitCallFlags = this->scriptContext->GetThreadContext()->GetImplicitCallFlags();

if (curImplicitCallFlags > ImplicitCall\_None)

{

//error implicit bit is set , reparse without asmjs

this->scriptContext->GetThreadContext()->SetDisableImplicitFlags(prevDisableImplicitFlags);

this->scriptContext->GetThreadContext()->SetImplicitCallFlags(savedImplicitCallFlags);

return true;

}

return false;

}

uint

InterpreterStackFrame::CallLoopBody(JavascriptMethod address)

{

#ifdef \_M\_IX86

void \*savedEsp = NULL;

\_\_asm

{

// Save ESP

mov savedEsp, esp

// 8-byte align frame to improve floating point perf of our JIT'd code.

and esp, -8

// Add an extra 4-bytes to the stack since we'll be pushing 3 arguments

push eax

}

#endif

#if defined(\_M\_ARM32\_OR\_ARM64)

// For ARM we need to make sure that pipeline is synchronized with memory/cache for newly jitted code.

// Note: this does not seem to affect perf, but if it was, we could add a boolean isCalled to EntryPointInfo

// and do ISB only for 1st time this entry point is called (potential working set regression though).

\_InstructionSynchronizationBarrier();

#endif

uint newOffset = ::Math::PointerCastToIntegral<uint>(address(function, CallInfo(CallFlags\_InternalFrame, 1), this));

#ifdef \_M\_IX86

\_asm

{

// Restore ESP

mov esp, savedEsp

}

#endif

return newOffset;

}

uint

InterpreterStackFrame::CallAsmJsLoopBody(JavascriptMethod address)

{

#ifdef \_M\_IX86

void \*savedEsp = NULL;

\_\_asm

{

// Save ESP

mov savedEsp, esp

// Add an extra 4-bytes to the stack since we'll be pushing 3 arguments

push eax

}

#endif

#if defined(\_M\_ARM32\_OR\_ARM64)

// For ARM we need to make sure that pipeline is synchronized with memory/cache for newly jitted code.

// Note: this does not seem to affect perf, but if it was, we could add a boolean isCalled to EntryPointInfo

// and do ISB only for 1st time this entry point is called (potential working set regression though).

\_InstructionSynchronizationBarrier();

#endif

uint newOffset = ::Math::PointerCastToIntegral<uint>(address(function, CallInfo(CallFlags\_InternalFrame, 1), this));

#ifdef \_M\_IX86

\_asm

{

// Restore ESP

mov esp, savedEsp

}

#endif

return newOffset;

}

template <class T>

void InterpreterStackFrame::OP\_NewScObjectNoCtorFull(const unaligned T\* playout)

{

Var function = GetReg(playout->R1);

Var newObj = JavascriptOperators::NewScObjectNoCtorFull(function, GetScriptContext());

SetReg(playout->R0, newObj);

}

///----------------------------------------------------------------------------

///

/// InterpreterStackFrame::OP\_NewScObject

///

/// OP\_NewScObject() allocates a new DynamicObject and initializes it with an

/// optional "constructor" function.

///

/// NOTE: The return register must be carefully chosen to ensure proper

/// behavior:

/// 1. OpCode::NewInstance should never specify "R0" as the register to

/// store the new instance, because it will get whacked from the

/// "constructor" function's return value:

///

/// var a1 = Date(); <-- a1 = string returned from Date() function

/// var a2 = new Date(); <-- a2 = instance return from NewInstance.

/// Date()'s return value is thrown away.

///

/// 2. If an exception is thrown during construction, the destination

/// variable / field should \_\_not\_\_ be modified. Therefore, the destination

/// register should always be a temporary and never a valid local variable.

/// After successfully returning from the constructor function, the new

/// instance is valid and may be stored in its final destination variable /

/// field.

///

/// OPCODE NewObject:

/// T1 = new DynamicObject(Function.Prototype)

/// OutArg[0] = T1

/// Call(Function, ArgCount)

/// Local[Return] = T1

///

/// - R0: Destination "local" register

/// - R1: Optional constructor JavascriptFunction instance or 'null'

///

///----------------------------------------------------------------------------

template <class T, bool Profiled, bool ICIndex>

void InterpreterStackFrame::OP\_NewScObject\_Impl(const unaligned T\* playout, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices)

{

if (ICIndex)

{

Assert(inlineCacheIndex != Js::Constants::NoInlineCacheIndex);

}

Var newVarInstance =

#if ENABLE\_PROFILE\_INFO

Profiled ?

ProfiledNewScObject\_Helper(

GetReg(playout->Function),

playout->ArgCount,

static\_cast<const unaligned OpLayoutDynamicProfile<T> \*>(playout)->profileId,

inlineCacheIndex,

spreadIndices) :

#endif

NewScObject\_Helper(GetReg(playout->Function), playout->ArgCount, spreadIndices);

SetReg((RegSlot)playout->Return, newVarInstance);

}

template <class T, bool Profiled>

void InterpreterStackFrame::OP\_NewScObjArray\_Impl(const unaligned T\* playout, const Js::AuxArray<uint32> \*spreadIndices)

{

// Always profile this operation when auto-profiling so that array type changes are tracked

#if ENABLE\_PROFILE\_INFO

if (!Profiled && !isAutoProfiling)

#else

Assert(!Profiled);

#endif

{

OP\_NewScObject\_Impl<T, Profiled, false>(playout, Js::Constants::NoInlineCacheIndex, spreadIndices);

return;

}

#if ENABLE\_PROFILE\_INFO

Arguments args(CallInfo(CallFlags\_New, playout->ArgCount), m\_outParams);

uint32 spreadSize = 0;

if (spreadIndices != nullptr)

{

spreadSize = JavascriptFunction::GetSpreadSize(args, spreadIndices, scriptContext);

// Allocate room on the stack for the spread args.

Arguments outArgs(CallInfo(CallFlags\_New, 0), nullptr);

outArgs.Info.Count = spreadSize;

const unsigned STACK\_ARGS\_ALLOCA\_THRESHOLD = 8; // Number of stack args we allow before using \_alloca

Var stackArgs[STACK\_ARGS\_ALLOCA\_THRESHOLD];

size\_t outArgsSize = 0;

if (outArgs.Info.Count > STACK\_ARGS\_ALLOCA\_THRESHOLD)

{

PROBE\_STACK(scriptContext, outArgs.Info.Count \* sizeof(Var) + Js::Constants::MinStackDefault); // args + function call

outArgsSize = outArgs.Info.Count \* sizeof(Var);

outArgs.Values = (Var\*)\_alloca(outArgsSize);

}

else

{

outArgs.Values = stackArgs;

outArgsSize = STACK\_ARGS\_ALLOCA\_THRESHOLD \* sizeof(Var);

ZeroMemory(outArgs.Values, outArgsSize); // We may not use all of the elements

}

JavascriptFunction::SpreadArgs(args, outArgs, spreadIndices, scriptContext);

SetReg(

(RegSlot)playout->Return,

ProfilingHelpers::ProfiledNewScObjArray(

GetReg(playout->Function),

outArgs,

function,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId2));

}

else

{

SetReg(

(RegSlot)playout->Return,

ProfilingHelpers::ProfiledNewScObjArray(

GetReg(playout->Function),

args,

function,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId2));

}

PopOut(playout->ArgCount);

#endif

}

void InterpreterStackFrame::OP\_NewScObject\_A\_Impl(const unaligned OpLayoutAuxiliary \* playout, RegSlot \*target)

{

const Js::VarArrayVarCount \* vars = Js::ByteCodeReader::ReadVarArrayVarCount(playout->Offset, this->GetFunctionBody());

int count = Js::TaggedInt::ToInt32(vars->count);

// Push the parameters to stack

for (int i=0;i<count; i++)

{

SetOut( (ArgSlot)(i+1), vars->elements[i]);

}

Var newVarInstance = NewScObject\_Helper(GetReg((RegSlot)playout->C1), (ArgSlot)count+1);

SetReg((RegSlot)playout->R0, newVarInstance);

}

Var InterpreterStackFrame::NewScObject\_Helper(Var target, ArgSlot ArgCount, const Js::AuxArray<uint32> \*spreadIndices)

{

Arguments args(CallInfo(CallFlags\_New, ArgCount), m\_outParams);

Var newVarInstance = JavascriptOperators::NewScObject(target, args, GetScriptContext(), spreadIndices);

PopOut(ArgCount);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newVarInstance));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newVarInstance = JavascriptProxy::AutoProxyWrapper(newVarInstance);

// this might come from a different scriptcontext.

newVarInstance = CrossSite::MarshalVar(GetScriptContext(), newVarInstance);

}

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

{

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().NewScriptObject( target, args, newVarInstance );

}

#endif

return newVarInstance;

}

#if ENABLE\_PROFILE\_INFO

Var InterpreterStackFrame::ProfiledNewScObject\_Helper(Var target, ArgSlot ArgCount, ProfileId profileId, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices)

{

Arguments args(CallInfo(CallFlags\_New, ArgCount), m\_outParams);

Var newVarInstance =

ProfilingHelpers::ProfiledNewScObject(

target,

args,

GetFunctionBody(),

profileId,

inlineCacheIndex,

spreadIndices);

PopOut(ArgCount);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newVarInstance));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newVarInstance = JavascriptProxy::AutoProxyWrapper(newVarInstance);

// this might come from a different scriptcontext.

newVarInstance = CrossSite::MarshalVar(GetScriptContext(), newVarInstance);

}

#endif

#ifdef TELEMETRY\_PROFILED

{

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().NewScriptObject( target, args, newVarInstance );

}

#endif

return newVarInstance;

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_LdElementUndefined(const unaligned OpLayoutT\_ElementU<T>\* playout)

{

if (this->m\_functionBody->IsEval())

{

JavascriptOperators::OP\_LoadUndefinedToElementDynamic(GetReg(playout->Instance),

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

}

else

{

JavascriptOperators::OP\_LoadUndefinedToElement(GetReg(playout->Instance),

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex));

}

}

template <typename T>

void InterpreterStackFrame::OP\_LdLocalElementUndefined(const unaligned OpLayoutT\_ElementRootU<T>\* playout)

{

if (this->m\_functionBody->IsEval())

{

JavascriptOperators::OP\_LoadUndefinedToElementDynamic(this->localClosure,

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

}

else

{

JavascriptOperators::OP\_LoadUndefinedToElement(this->localClosure,

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex));

}

}

template <typename T>

void InterpreterStackFrame::OP\_LdElementUndefinedScoped(const unaligned OpLayoutT\_ElementScopedU<T>\* playout)

{

// Implicit root object as default instance

JavascriptOperators::OP\_LoadUndefinedToElementScoped(GetEnvForEvalCode(),

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext());

}

void InterpreterStackFrame::OP\_ChkUndecl(Var aValue)

{

if (this->scriptContext->IsUndeclBlockVar(aValue))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

}

void InterpreterStackFrame::OP\_ChkNewCallFlag()

{

if (!(this->m\_callFlags & CallFlags\_New))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassConstructorCannotBeCalledWithoutNew);

}

}

void InterpreterStackFrame::OP\_EnsureNoRootProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

JavascriptOperators::OP\_EnsureNoRootProperty(instance, this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex));

}

void InterpreterStackFrame::OP\_EnsureNoRootRedeclProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

JavascriptOperators::OP\_EnsureNoRootRedeclProperty(instance, this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex));

}

void InterpreterStackFrame::OP\_ScopedEnsureNoRedeclProperty(Var aValue, uint propertyIdIndex, Var aValue2)

{

Js::PropertyId propertyId = this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex);

JavascriptOperators::OP\_ScopedEnsureNoRedeclProperty((FrameDisplay\*)aValue, propertyId, aValue2);

}

Var InterpreterStackFrame::OP\_InitUndecl()

{

return this->scriptContext->GetLibrary()->GetUndeclBlockVar();

}

void InterpreterStackFrame::OP\_InitUndeclSlot(Var aValue, int32 slot)

{

this->OP\_StSlot(aValue, slot, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

void InterpreterStackFrame::OP\_TryCatch(const unaligned OpLayoutBr\* playout)

{

Js::JavascriptExceptionObject\* exception = NULL;

try

{

this->nestedTryDepth++;

// mark the stackFrame as 'in try block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinTryBlock;

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

if (scriptContext->IsInDebugMode())

{

this->ProcessWithDebugging();

this->TrySetRetOffset();

}

else

{

this->Process();

this->TrySetRetOffset();

}

}

catch (Js::JavascriptExceptionObject \* exceptionObject)

{

// We are using C++ exception handling which does not unwind the stack in the catch block.

// For stack overflow and OOM exceptions, we cannot run user code here because the stack is not unwind.

exception = exceptionObject;

}

if (--this->nestedTryDepth == -1)

{

// unmark the stackFrame as 'in try block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinTryBlock;

}

// Now that the stack is unwound, let's run the catch block.

if (exception)

{

if (exception->IsGeneratorReturnException())

{

// Generator return scenario, so no need to go into the catch block and we must rethrow to propagate the exception to down level

throw exception;

}

exception = exception->CloneIfStaticExceptionObject(scriptContext);

// We've got a JS exception. Grab the exception object and assign it to the

// catch object's location, then call the handler (i.e., we consume the Catch op here).

Var catchObject = exception->GetThrownObject(scriptContext);

m\_reader.SetCurrentRelativeOffset((const byte \*)(playout + 1), playout->RelativeJumpOffset);

LayoutSize layoutSize;

OpCode catchOp = m\_reader.ReadOp(layoutSize);

#ifdef BYTECODE\_BRANCH\_ISLAND

if (catchOp == Js::OpCode::BrLong)

{

Assert(layoutSize == SmallLayout);

auto playoutBrLong = m\_reader.BrLong();

m\_reader.SetCurrentRelativeOffset((const byte \*)(playoutBrLong + 1), playoutBrLong->RelativeJumpOffset);

catchOp = m\_reader.ReadOp(layoutSize);

}

#endif

AssertMsg(catchOp == OpCode::Catch, "Catch op not found at catch offset");

RegSlot reg = layoutSize == SmallLayout ? m\_reader.Reg1\_Small()->R0 :

layoutSize == MediumLayout ? m\_reader.Reg1\_Medium()->R0 : m\_reader.Reg1\_Large()->R0;

SetReg(reg, catchObject);

ResetOut();

this->nestedCatchDepth++;

// mark the stackFrame as 'in catch block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

this->ProcessCatch();

if (--this->nestedCatchDepth == -1)

{

// unmark the stackFrame as 'in catch block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

}

}

void InterpreterStackFrame::ProcessCatch()

{

if (this->scriptContext->IsInDebugMode())

{

this->DebugProcess();

}

else

{

this->Process();

}

}

int InterpreterStackFrame::ProcessFinally()

{

this->nestedFinallyDepth++;

// mark the stackFrame as 'in finally block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinFinallyBlock;

int newOffset = 0;

if (scriptContext->IsInDebugMode())

{

newOffset = ::Math::PointerCastToIntegral<int>(this->DebugProcess());

}

else

{

newOffset = ::Math::PointerCastToIntegral<int>(this->Process());

}

if (--this->nestedFinallyDepth == -1)

{

// unmark the stackFrame as 'in finally block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinFinallyBlock;

}

return newOffset;

}

void InterpreterStackFrame::ProcessTryCatchBailout(EHBailoutData \* ehBailoutData, uint32 tryNestingDepth)

{

int catchOffset = ehBailoutData->catchOffset;

Js::JavascriptExceptionObject\* exception = NULL;

if (catchOffset != 0)

{

try

{

this->nestedTryDepth++;

// mark the stackFrame as 'in try block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinTryBlock;

if (tryNestingDepth != 0)

{

this->ProcessTryCatchBailout(ehBailoutData->child, --tryNestingDepth);

}

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

if (scriptContext->IsInDebugMode())

{

this->ProcessWithDebugging();

this->TrySetRetOffset();

}

else

{

this->Process();

this->TrySetRetOffset();

}

}

catch (Js::JavascriptExceptionObject \* exceptionObject)

{

// We are using C++ exception handling which does not unwind the stack in the catch block.

// For stack overflow and OOM exceptions, we cannot run user code here because the stack is not unwind.

exception = exceptionObject;

}

}

else

{

this->nestedCatchDepth++;

// mark the stackFrame as 'in catch block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

if (tryNestingDepth != 0)

{

this->ProcessTryCatchBailout(ehBailoutData->child, --tryNestingDepth);

}

this->ProcessCatch();

if (--this->nestedCatchDepth == -1)

{

// unmark the stackFrame as 'in catch block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

return;

}

if (--this->nestedTryDepth == -1)

{

// unmark the stackFrame as 'in try block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinTryBlock;

}

// Now that the stack is unwound, let's run the catch block.

if (exception)

{

if (exception->IsGeneratorReturnException())

{

// Generator return scenario, so no need to go into the catch block and we must rethrow to propagate the exception to down level

throw exception;

}

exception = exception->CloneIfStaticExceptionObject(scriptContext);

// We've got a JS exception. Grab the exception object and assign it to the

// catch object's location, then call the handler (i.e., we consume the Catch op here).

Var catchObject = exception->GetThrownObject(scriptContext);

m\_reader.SetCurrentOffset(catchOffset);

LayoutSize layoutSize;

OpCode catchOp = m\_reader.ReadOp(layoutSize);

#ifdef BYTECODE\_BRANCH\_ISLAND

if (catchOp == Js::OpCode::BrLong)

{

Assert(layoutSize == SmallLayout);

auto playoutBrLong = m\_reader.BrLong();

m\_reader.SetCurrentRelativeOffset((const byte \*)(playoutBrLong + 1), playoutBrLong->RelativeJumpOffset);

catchOp = m\_reader.ReadOp(layoutSize);

}

#endif

AssertMsg(catchOp == OpCode::Catch, "Catch op not found at catch offset");

RegSlot reg = layoutSize == SmallLayout ? m\_reader.Reg1\_Small()->R0 :

layoutSize == MediumLayout ? m\_reader.Reg1\_Medium()->R0 : m\_reader.Reg1\_Large()->R0;

SetReg(reg, catchObject);

ResetOut();

this->nestedCatchDepth++;

// mark the stackFrame as 'in catch block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

this->ProcessCatch();

if (--this->nestedCatchDepth == -1)

{

// unmark the stackFrame as 'in catch block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

}

return;

}

void InterpreterStackFrame::TrySetRetOffset()

{

Assert(this->m\_flags & Js::InterpreterStackFrameFlags\_WithinTryBlock);

// It may happen that a JITted loop body returned the offset of RET. If the loop body was

// called from a try, the interpreter "Process()" should also just return.

if (this->retOffset != 0)

{

m\_reader.SetCurrentOffset(this->retOffset);

}

}

bool InterpreterStackFrame::IsInCatchOrFinallyBlock()

{

return (this->m\_flags & Js::InterpreterStackFrameFlags\_WithinCatchBlock) ||

(this->m\_flags & Js::InterpreterStackFrameFlags\_WithinFinallyBlock);

}

void InterpreterStackFrame::OP\_ResumeCatch()

{

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

if (scriptContext->IsInDebugMode())

{

this->DebugProcess();

}

else

{

this->Process();

}

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

/// ---------------------------------------------------------------------------------------------------

/// The behavior we want is the following:

/// - If the control leaves the user's try without throwing, execute the finally and continue

/// after the end of the try.

/// - If the user code throws, catch this exception and then execute this finally while unwinding to

/// the handler (if any).

/// ---------------------------------------------------------------------------------------------------

void InterpreterStackFrame::ProcessTryFinally(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException, Js::RegSlot regOffset, bool hasYield)

{

Js::JavascriptExceptionObject\* pExceptionObject = nullptr;

bool skipFinallyBlock = false;

try

{

Js::Var result = nullptr;

this->nestedTryDepth++;

// mark the stackFrame as 'in try block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinTryBlock;

if (scriptContext->IsInDebugMode())

{

result = this->ProcessWithDebugging();

}

else

{

result = this->Process();

}

if (result == nullptr)

{

Assert(hasYield);

skipFinallyBlock = true;

}

}

catch (Js::JavascriptExceptionObject \* e)

{

pExceptionObject = e;

}

if (--this->nestedTryDepth == -1)

{

// unmark the stackFrame as 'in try block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinTryBlock;

}

if (skipFinallyBlock)

{

// A leave occurred due to a yield

return;

}

// Save the current IP so execution can continue there if the finally doesn't

// take control of the flow.

int newOffset = 0;

int currOffset = m\_reader.GetCurrentOffset();

if (hasYield)

{

// save the exception if there is one to a register in case we yield during the finally block

// and need to get that exception object back upon resume in OP\_ResumeFinally

SetNonVarReg(regException, pExceptionObject);

SetNonVarReg(regOffset, reinterpret\_cast<Js::Var>(currOffset));

}

if (pExceptionObject && !pExceptionObject->IsGeneratorReturnException())

{

// Clone static exception object early in case finally block overwrites it

pExceptionObject = pExceptionObject->CloneIfStaticExceptionObject(scriptContext);

}

if (pExceptionObject && scriptContext->IsInDebugMode() &&

pExceptionObject != scriptContext->GetThreadContext()->GetPendingSOErrorObject())

{

// Swallowing an exception that has triggered a finally is not implemented

// (This appears to be the same behavior as ie8)

pExceptionObject->SetDebuggerSkip(false);

}

// Call into the finally by setting the IP, consuming the Finally, and letting the interpreter recurse.

m\_reader.SetCurrentRelativeOffset(ip, jumpOffset);

ResetOut();

newOffset = this->ProcessFinally();

bool endOfFinallyBlock = newOffset == 0;

if (endOfFinallyBlock)

{

// Finally completed without taking over the flow. Resume where we left off before calling it.

m\_reader.SetCurrentOffset(currOffset);

}

else

{

// Finally seized the flow with a jump out of its scope. Resume at the jump target and

// force the runtime to return to this frame without executing the catch.

m\_reader.SetCurrentOffset(newOffset);

return;

}

if (pExceptionObject && (endOfFinallyBlock || !pExceptionObject->IsGeneratorReturnException()))

{

throw pExceptionObject;

}

}

void InterpreterStackFrame::OP\_TryFinally(const unaligned OpLayoutBr\* playout)

{

ProcessTryFinally((const byte\*)(playout + 1), playout->RelativeJumpOffset);

}

void InterpreterStackFrame::OP\_TryFinallyWithYield(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException, Js::RegSlot regOffset)

{

ProcessTryFinally(ip, jumpOffset, regException, regOffset, true);

}

void InterpreterStackFrame::OP\_ResumeFinally(const byte\* ip, Js::JumpOffset jumpOffset, RegSlot exceptionRegSlot, RegSlot offsetRegSlot)

{

this->m\_flags |= InterpreterStackFrameFlags\_WithinFinallyBlock;

int newOffset = 0;

if (scriptContext->IsInDebugMode())

{

newOffset = ::Math::PointerCastToIntegral<int>(this->DebugProcess());

}

else

{

newOffset = ::Math::PointerCastToIntegral<int>(this->Process());

}

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinFinallyBlock;

bool endOfFinallyBlock = newOffset == 0;

if (endOfFinallyBlock)

{

// Finally completed without taking over the flow. Resume where we left off before calling it.

int currOffset = ::Math::PointerCastToIntegral<int>(GetNonVarReg(offsetRegSlot));

m\_reader.SetCurrentOffset(currOffset);

}

else

{

// Finally seized the flow with a jump out of its scope. Resume at the jump target and

// force the runtime to return to this frame without executing the catch.

m\_reader.SetCurrentOffset(newOffset);

return;

}

Js::JavascriptExceptionObject\* exceptionObj = (Js::JavascriptExceptionObject\*)GetNonVarReg(exceptionRegSlot);

if (exceptionObj && (endOfFinallyBlock || !exceptionObj->IsGeneratorReturnException()))

{

throw exceptionObj;

}

}

template <typename T>

void InterpreterStackFrame::OP\_IsInst(const unaligned T\* playout)

{

Var instance = GetReg(playout->R1);

Var function = GetReg(playout->R2);

IsInstInlineCache \*inlineCache = this->GetIsInstInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

Var result = JavascriptOperators::OP\_IsInst(instance, function, scriptContext, inlineCache);

#ifdef ENABLE\_BASIC\_TELEMETRY

{

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().IsInstanceOf(instance, function, result);

}

#endif

SetReg(playout->R0, result);

}

template <typename T>

void InterpreterStackFrame::OP\_ApplyArgs(const unaligned OpLayoutT\_Reg5<T> \* playout)

{

// Always save and restore implicit call flags when calling out

// REVIEW: Can we avoid it if we don't collect dynamic profile info?

ThreadContext \* threadContext = scriptContext->GetThreadContext();

Js::ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

// Currently ApplyArgs is equivalent to CallFldVoid (where we don't use the return value)

Var v=GetNonVarReg(playout->R4);

JavascriptOperators::OP\_ApplyArgs(GetReg(playout->R1),GetReg(playout->R2),

(void\*\*)GetNonVarReg(playout->R3),\*((CallInfo\*)&v),GetScriptContext());

threadContext->SetImplicitCallFlags(savedImplicitCallFlags);

}

void InterpreterStackFrame::OP\_SpreadArrayLiteral(const unaligned OpLayoutReg2Aux \* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

Var instance = GetReg(playout->R1);

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

const Js::AuxArray<uint32> \*spreadIndices = m\_reader.ReadAuxArray<uint32>(playout->Offset, this->GetFunctionBody());

ScriptContext\* scriptContext = GetScriptContext();

Var result = JavascriptArray::SpreadArrayArgs(instance, spreadIndices, scriptContext);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->R0, result);

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext)

{

JavascriptOperators::CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext)

{

JavascriptOperators::CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplayNoParent<true>(argHead, scriptContext);

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay;

bool strict = this->m\_functionBody->GetIsStrictMode();

if (strict)

{

frameDisplay = JavascriptOperators::OP\_LdStrictFrameDisplay(argHead, argEnv, scriptContext);

}

else

{

frameDisplay = JavascriptOperators::OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

return frameDisplay;

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdFrameDisplaySetLocal(void \*argHead, void \*argEnv, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay = OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

this->SetLocalFrameDisplay(frameDisplay);

return frameDisplay;

}

FrameDisplay \*

InterpreterStackFrame::NewFrameDisplay(void \*argHead, void \*argEnv)

{

FrameDisplay \*frameDisplay;

bool strict = this->m\_functionBody->GetIsStrictMode();

if (!this->m\_functionBody->DoStackFrameDisplay() || !this->GetLocalFrameDisplay())

{

// Null local frame display probably indicates that we bailed out of an inlinee.

// Once we support stack closures in inlined functions, we can just assert that this value

// is never null if we should be allocating on the stack.

return this->OP\_LdFrameDisplaySetLocal(argHead, argEnv, this->GetScriptContext());

}

frameDisplay = this->GetLocalFrameDisplay();

Assert(frameDisplay != nullptr);

frameDisplay->SetTag(true);

frameDisplay->SetStrictMode(strict);

frameDisplay->SetLength(this->m\_functionBody->GetEnvDepth() + 1);

Assert(frameDisplay->GetLength() == ((FrameDisplay\*)argEnv)->GetLength() + 1);

for (uint i = 0; i < ((FrameDisplay\*)argEnv)->GetLength(); i++)

{

frameDisplay->SetItem(i + 1, ((FrameDisplay\*)argEnv)->GetItem(i));

}

frameDisplay->SetItem(0, argHead);

return frameDisplay;

}

template<bool innerFD>

FrameDisplay \*

InterpreterStackFrame::OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay;

bool strict = this->m\_functionBody->GetIsStrictMode();

Var argEnv = nullptr;

if (innerFD && this->m\_functionBody->GetLocalFrameDisplayReg() != Constants::NoRegister)

{

argEnv = this->GetLocalFrameDisplay();

}

if (argEnv == nullptr && this->m\_functionBody->GetEnvReg() != Constants::NoRegister)

{

argEnv = this->LdEnv();

}

if (argEnv == nullptr)

{

if (strict)

{

frameDisplay = JavascriptOperators::OP\_LdStrictFrameDisplayNoParent(argHead, scriptContext);

}

else

{

frameDisplay = JavascriptOperators::OP\_LdFrameDisplayNoParent(argHead, scriptContext);

}

}

else

{

if (strict)

{

frameDisplay = JavascriptOperators::OP\_LdStrictFrameDisplay(argHead, argEnv, scriptContext);

}

else

{

frameDisplay = JavascriptOperators::OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

}

return frameDisplay;

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdFuncExprFrameDisplaySetLocal(void \*argHead1, void \*argHead2, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay = OP\_LdFrameDisplayNoParent<false>(argHead2, scriptContext);

frameDisplay = OP\_LdFrameDisplay(argHead1, frameDisplay, scriptContext);

this->SetLocalFrameDisplay(frameDisplay);

return frameDisplay;

}

FrameDisplay\* InterpreterStackFrame::GetLocalFrameDisplay() const

{

return this->localFrameDisplay;

}

void InterpreterStackFrame::SetLocalFrameDisplay(FrameDisplay\* frameDisplay)

{

this->localFrameDisplay = frameDisplay;

}

Var InterpreterStackFrame::GetLocalClosure() const

{

return this->localClosure;

}

void InterpreterStackFrame::SetLocalClosure(Var closure)

{

this->localClosure = closure;

}

void

InterpreterStackFrame::OP\_NewInnerScopeSlots(uint innerScopeIndex, uint count, int scopeIndex, ScriptContext \*scriptContext, FunctionBody \*functionBody)

{

Var \* slotArray;

slotArray =

JavascriptOperators::OP\_NewScopeSlotsWithoutPropIds(count, scopeIndex, scriptContext, functionBody);

this->SetInnerScopeFromIndex(innerScopeIndex, slotArray);

}

template <typename T>

void InterpreterStackFrame::OP\_CloneInnerScopeSlots(const unaligned OpLayoutT\_Unsigned1<T> \*playout)

{

uint innerScopeIndex = playout->C1;

Var \* slotArray;

slotArray = (Var\*)this->InnerScopeFromIndex(innerScopeIndex);

slotArray = JavascriptOperators::OP\_CloneScopeSlots(slotArray, scriptContext);

this->SetInnerScopeFromIndex(innerScopeIndex, slotArray);

}

template <typename T>

void InterpreterStackFrame::OP\_CloneBlockScope(const unaligned OpLayoutT\_Unsigned1<T> \*playout)

{

uint innerScopeIndex = playout->C1;

Var scope = this->InnerScopeFromIndex(innerScopeIndex);

BlockActivationObject\* blockScope = BlockActivationObject::FromVar(scope);

scope = JavascriptOperators::OP\_CloneBlockScope(blockScope, scriptContext);

this->SetInnerScopeFromIndex(innerScopeIndex, scope);

}

Var \*

InterpreterStackFrame::NewScopeSlots(unsigned int size, ScriptContext \*scriptContext, Var scope)

{

Var \* slotArray;

slotArray = JavascriptOperators::OP\_NewScopeSlots(size, scriptContext, scope);

this->SetLocalClosure(slotArray);

return slotArray;

}

Var \*

InterpreterStackFrame::NewScopeSlots()

{

Var \* slotArray;

FunctionBody \* functionBody = this->m\_functionBody;

uint scopeSlotCount = functionBody->scopeSlotArraySize;

Assert(scopeSlotCount != 0);

if (!functionBody->DoStackScopeSlots())

{

return this->NewScopeSlots(

scopeSlotCount + ScopeSlots::FirstSlotIndex, this->GetScriptContext(), (Var)functionBody);

}

slotArray = (Var\*)this->GetLocalClosure();

Assert(slotArray != nullptr);

ScopeSlots scopeSlots(slotArray);

scopeSlots.SetCount(scopeSlotCount);

scopeSlots.SetScopeMetadata((Var)functionBody);

Var undef = functionBody->GetScriptContext()->GetLibrary()->GetUndefined();

for (unsigned int i = 0; i < scopeSlotCount; i++)

{

scopeSlots.Set(i, undef);

}

return slotArray;

}

Var

InterpreterStackFrame::NewScopeObject()

{

Var scopeObject;

if (m\_functionBody->HasCachedScopePropIds())

{

const Js::PropertyIdArray \*propIds =

Js::ByteCodeReader::ReadPropertyIdArray(0, this->GetFunctionBody(), ActivationObjectEx::ExtraSlotCount());

Var funcExpr = this->GetFunctionExpression();

PropertyId objectId = ActivationObjectEx::GetLiteralObjectRef(propIds);

scopeObject = JavascriptOperators::OP\_InitCachedScope(funcExpr, propIds,

this->GetFunctionBody()->GetObjectLiteralTypeRef(objectId),

propIds->hasNonSimpleParams, GetScriptContext());

}

else

{

scopeObject = JavascriptOperators::OP\_NewScopeObject(GetScriptContext());

}

this->SetLocalClosure(scopeObject);

return scopeObject;

}

FrameDisplay \*

InterpreterStackFrame::GetFrameDisplayForNestedFunc() const

{

if (this->localFrameDisplay == nullptr)

{

return (FrameDisplay\*)LdEnv();

}

return this->localFrameDisplay;

}

template <class T>

void InterpreterStackFrame::OP\_NewStackScFunc(const unaligned T \* playout)

{

uint funcIndex = playout->SlotIndex;

FrameDisplay \*frameDisplay = this->GetFrameDisplayForNestedFunc();

SetRegAllowStackVarEnableOnly(playout->Value,

StackScriptFunction::OP\_NewStackScFunc(frameDisplay,

reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(funcIndex)),

this->GetStackNestedFunction(funcIndex)));

}

template <class T>

void InterpreterStackFrame::OP\_NewInnerStackScFunc(const unaligned T \* playout)

{

uint funcIndex = playout->SlotIndex;

FrameDisplay \*frameDisplay = (FrameDisplay\*)GetNonVarReg(playout->Instance);

SetRegAllowStackVarEnableOnly(playout->Value,

StackScriptFunction::OP\_NewStackScFunc(frameDisplay,

reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(funcIndex)),

this->GetStackNestedFunction(funcIndex)));

}

template <class T>

void InterpreterStackFrame::OP\_DeleteFld(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteLocalFld(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteProperty(this->localClosure, m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

SetReg(playout->Instance, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteRootFld(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteRootProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteFldStrict(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext(), PropertyOperation\_StrictMode);

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteRootFldStrict(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteRootProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext(), PropertyOperation\_StrictMode);

SetReg(playout->Value, result);

}

template <typename T>

void InterpreterStackFrame::OP\_ScopedDeleteFld(const unaligned OpLayoutT\_ElementScopedC<T> \* playout)

{

// Implicit root object as default instance

Var result = JavascriptOperators::OP\_DeletePropertyScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext());

SetReg(playout->Value, result);

}

template <typename T>

void InterpreterStackFrame::OP\_ScopedDeleteFldStrict(const unaligned OpLayoutT\_ElementScopedC<T> \* playout)

{

// Implicit root object as default instance

Var result = JavascriptOperators::OP\_DeletePropertyScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext(), PropertyOperation\_StrictMode);

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_ScopedLdInst(const unaligned T \* playout)

{

Var thisVar;

Var rootObject = GetFunctionBody()->GetRootObject();

Var result = JavascriptOperators::OP\_GetInstanceScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), rootObject, &thisVar, GetScriptContext());

SetReg(playout->Value, result);

SetReg(playout->Value2, thisVar);

}

template <typename T>

void InterpreterStackFrame::OP\_ScopedInitFunc(const unaligned OpLayoutT\_ElementScopedC<T> \* playout)

{

JavascriptOperators::OP\_InitFuncScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(playout->Value), GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_ClearAttributes(const unaligned T \* playout)

{

JavascriptOperators::OP\_ClearAttributes(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex));

}

template <class T>

void InterpreterStackFrame::OP\_InitGetFld(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitGetter(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitSetFld(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitSetter(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitSetElemI(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitElemSetter(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext()

);

}

template <class T>

void InterpreterStackFrame::OP\_InitGetElemI(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitElemGetter(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext()

);

}

template <class T>

void InterpreterStackFrame::OP\_InitComputedProperty(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitComputedProperty(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext()

);

}

template <class T>

void InterpreterStackFrame::OP\_InitProto(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitProto(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(playout->Value));

}

void InterpreterStackFrame::DoInterruptProbe()

{

PROBE\_STACK(scriptContext, 0);

}

void InterpreterStackFrame::InitializeStackFunctions(StackScriptFunction \* scriptFunctions)

{

this->stackNestedFunctions = scriptFunctions;

FunctionBody \* functionBody = this->m\_functionBody;

uint nestedCount = functionBody->GetNestedCount();

for (uint i = 0; i < nestedCount; i++)

{

StackScriptFunction \* stackScriptFunction = scriptFunctions + i;

FunctionProxy\* nestedProxy = functionBody->GetNestedFunc(i);

ScriptFunctionType\* type = nestedProxy->EnsureDeferredPrototypeType();

new (stackScriptFunction)StackScriptFunction(nestedProxy, type);

}

}

StackScriptFunction \* InterpreterStackFrame::GetStackNestedFunction(uint index)

{

Assert(index < this->m\_functionBody->GetNestedCount());

// Re-check if we have disable stack nested function

if (this->m\_functionBody->DoStackNestedFunc())

{

return this->stackNestedFunctions + index;

}

return nullptr;

}

void InterpreterStackFrame::SetExecutingStackFunction(ScriptFunction \* scriptFunction)

{

Assert(ThreadContext::IsOnStack(this->function));

Assert(this->m\_functionBody == scriptFunction->GetFunctionBody());

this->function = scriptFunction;

}

DWORD\_PTR InterpreterStackFrame::GetStackAddress() const

{

return m\_stackAddress;

}

void\* InterpreterStackFrame::GetAddressOfReturnAddress() const

{

return this->addressOfReturnAddress;

}

template <class T>

const byte \* InterpreterStackFrame::OP\_Br(const unaligned T \* playout)

{

return m\_reader.SetCurrentRelativeOffset((const byte \*)(playout + 1), playout->RelativeJumpOffset);

}

template <class T>

void InterpreterStackFrame::OP\_InitClass(const unaligned OpLayoutT\_Class<T> \* playout)

{

JavascriptOperators::OP\_InitClass(GetReg(playout->Constructor), playout->Extends != Js::Constants::NoRegister ? GetReg(playout->Extends) : NULL, GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_EmitTmpRegCount(const unaligned OpLayoutT\_Unsigned1<T> \* playout)

{

this->scriptContext->GetDebugContext()->GetProbeContainer()->SetCurrentTmpRegCount(playout->C1);

}

Var InterpreterStackFrame::OP\_LdSuper(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_LdSuper(function, scriptContext);

}

Var InterpreterStackFrame::OP\_LdSuperCtor(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_LdSuperCtor(function, scriptContext);

}

Var InterpreterStackFrame::OP\_ScopedLdSuper(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_ScopedLdSuper(function, scriptContext);

}

Var InterpreterStackFrame::OP\_ScopedLdSuperCtor(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_ScopedLdSuperCtor(function, scriptContext);

}

void InterpreterStackFrame::ValidateRegValue(Var value, bool allowStackVar, bool allowStackVarOnDisabledStackNestedFunc) const

{

#if DBG

if (value != nullptr && !TaggedNumber::Is(value))

{

if (!allowStackVar || !this->m\_functionBody->DoStackNestedFunc())

{

Assert(!ThreadContext::IsOnStack(value)

|| (allowStackVar && allowStackVarOnDisabledStackNestedFunc && StackScriptFunction::IsBoxed(value)));

}

Assert(!CrossSite::NeedMarshalVar(value, GetScriptContext()));

}

#endif

}

template <typename RegSlotType>

Var InterpreterStackFrame::GetReg(RegSlotType localRegisterID) const

{

Var value = m\_localSlots[localRegisterID];

ValidateRegValue(value);

return value;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetReg(RegSlotType localRegisterID, Var value)

{

Assert(localRegisterID == 0 || localRegisterID >= m\_functionBody->GetConstantCount());

ValidateRegValue(value);

m\_localSlots[localRegisterID] = value;

}

template <typename T>

T InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (T)m\_localIntSlots[localRegisterID];

}

// specialized version for doubles

template <>

double InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (double)m\_localDoubleSlots[localRegisterID];

}

template <>

float InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (float)m\_localFloatSlots[localRegisterID];

}

template <typename T>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, T bValue)

{

m\_localIntSlots[localRegisterID] = (int)bValue;

}

template <>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, float bValue)

{

m\_localFloatSlots[localRegisterID] = (float)bValue;

}

template <>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, double bValue)

{

m\_localDoubleSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

int InterpreterStackFrame::GetRegRawInt(RegSlotType localRegisterID) const

{

return m\_localIntSlots[localRegisterID];

}

template <typename RegSlotType>

double InterpreterStackFrame::GetRegRawDouble(RegSlotType localRegisterID) const

{

return m\_localDoubleSlots[localRegisterID];

}

template <typename RegSlotType>

float InterpreterStackFrame::GetRegRawFloat(RegSlotType localRegisterID) const

{

return m\_localFloatSlots[localRegisterID];

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawInt(RegSlotType localRegisterID, int bValue)

{

m\_localIntSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawDouble(RegSlotType localRegisterID, double bValue)

{

m\_localDoubleSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawFloat(RegSlotType localRegisterID, float bValue)

{

m\_localFloatSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

Var InterpreterStackFrame::GetRegAllowStackVar(RegSlotType localRegisterID) const

{

Var value = m\_localSlots[localRegisterID];

ValidateRegValue(value, true);

return value;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegAllowStackVar(RegSlotType localRegisterID, Var value)

{

Assert(localRegisterID == 0 || localRegisterID >= m\_functionBody->GetConstantCount());

ValidateRegValue(value, true);

m\_localSlots[localRegisterID] = value;

}

template <typename RegSlotType>

Var InterpreterStackFrame::GetRegAllowStackVarEnableOnly(RegSlotType localRegisterID) const

{

Var value = m\_localSlots[localRegisterID];

ValidateRegValue(value, true, false);

return value;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegAllowStackVarEnableOnly(RegSlotType localRegisterID, Var value)

{

Assert(localRegisterID == 0 || localRegisterID >= m\_functionBody->GetConstantCount());

ValidateRegValue(value, true, false);

m\_localSlots[localRegisterID] = value;

}

template <>

AsmJsSIMDValue InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (AsmJsSIMDValue)m\_localSimdSlots[localRegisterID];

}

template<>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, AsmJsSIMDValue bValue)

{

m\_localSimdSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

AsmJsSIMDValue InterpreterStackFrame::GetRegRawSimd(RegSlotType localRegisterID) const

{

return m\_localSimdSlots[localRegisterID];

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawSimd(RegSlotType localRegisterID, AsmJsSIMDValue bValue)

{

m\_localSimdSlots[localRegisterID] = bValue;

}

template <class T>

void InterpreterStackFrame::OP\_SimdLdArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot dstReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value;

value = SIMDLdData(data, dataWidth);

SetRegRawSimd(dstReg, value);

}

template <class T>

void InterpreterStackFrame::OP\_SimdLdArrConstIndex(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = playout->SlotIndex;

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot dstReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value;

value = SIMDLdData(data, dataWidth);

SetRegRawSimd(dstReg, value);

}

template <class T>

void InterpreterStackFrame::OP\_SimdStArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot srcReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value = GetRegRawSimd(srcReg);

SIMDStData(data, value, dataWidth);

}

template <class T>

void InterpreterStackFrame::OP\_SimdStArrConstIndex(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = playout->SlotIndex;

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot srcReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value = GetRegRawSimd(srcReg);

SIMDStData(data, value, dataWidth);

}

Var InterpreterStackFrame::GetNonVarReg(RegSlot localRegisterID) const

{

return m\_localSlots[localRegisterID];

}

void InterpreterStackFrame::SetNonVarReg(RegSlot localRegisterID, Var aValue)

{

m\_localSlots[localRegisterID] = aValue;

}

Var InterpreterStackFrame::GetRootObject() const

{

Var rootObject = GetReg(Js::FunctionBody::RootObjectRegSlot);

Assert(rootObject == this->GetFunctionBody()->LoadRootObject());

return rootObject;

}

Var InterpreterStackFrame::OP\_ArgIn0()

{

return m\_inParams[0];

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::OP\_ProfiledArgOut\_A(const unaligned T \* playout)

{

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

Assert(playout->Reg > FunctionBody::FirstRegSlot && playout->Reg < functionBody->GetConstantCount());

Var value = GetReg(playout->Reg);

if (value != nullptr && TaggedInt::Is(value))

{

dynamicProfileInfo->RecordConstParameterAtCallSite(playout->profileId, playout->Arg);

}

SetOut(playout->Arg, GetReg(playout->Reg));

}

#endif

template <class T>

void InterpreterStackFrame::OP\_ArgOut\_A(const unaligned T \* playout)

{

SetOut(playout->Arg, GetReg(playout->Reg));

}

#if DBG

template <class T>

void InterpreterStackFrame::OP\_ArgOut\_ANonVar(const unaligned T \* playout)

{

SetOut(playout->Arg, GetNonVarReg(playout->Reg));

}

#endif

template <class T>

void InterpreterStackFrame::OP\_ArgOut\_Env(const unaligned T \* playout)

{

Var argEnv;

if (this->m\_functionBody->GetLocalFrameDisplayReg() != Constants::NoRegister)

{

argEnv = this->GetLocalFrameDisplay();

}

else

{

argEnv = this->LdEnv();

}

SetOut(playout->Arg, argEnv);

}

BOOL InterpreterStackFrame::OP\_BrFalse\_A(Var aValue, ScriptContext\* scriptContext)

{

return !JavascriptConversion::ToBoolean(aValue, scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrTrue\_A(Var aValue, ScriptContext\* scriptContext)

{

return JavascriptConversion::ToBoolean(aValue, scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrNotNull\_A(Var aValue)

{

return aValue != NULL;

}

BOOL InterpreterStackFrame::OP\_BrUndecl\_A(Var aValue)

{

return this->scriptContext->GetLibrary()->IsUndeclBlockVar(aValue);

}

BOOL InterpreterStackFrame::OP\_BrNotUndecl\_A(Var aValue)

{

return !this->scriptContext->GetLibrary()->IsUndeclBlockVar(aValue);

}

BOOL InterpreterStackFrame::OP\_BrOnHasProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext)

{

return JavascriptOperators::OP\_HasProperty(argInstance,

this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex), scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrOnNoProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext)

{

return !JavascriptOperators::OP\_HasProperty(argInstance,

this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex), scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrOnNoEnvProperty(Var envInstance, int32 slotIndex, uint propertyIdIndex, ScriptContext\* scriptContext)

{

Var instance = OP\_LdFrameDisplaySlot(envInstance, slotIndex);

return !JavascriptOperators::OP\_HasProperty(instance,

this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex), scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrOnClassConstructor(Var aValue)

{

return JavascriptOperators::IsClassConstructor(aValue);

}

template<class T>

void InterpreterStackFrame::OP\_LdLen(const unaligned T \* const playout)

{

Assert(playout);

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

const auto instance = GetReg(playout->R1);

Var length = JavascriptOperators::OP\_GetLength(instance, GetScriptContext());

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->R0, length);

}

#if ENABLE\_PROFILE\_INFO

template<class T>

void InterpreterStackFrame::OP\_ProfiledLdLen(const unaligned OpLayoutDynamicProfile<T> \*const playout)

{

Assert(playout);

const auto functionBody = m\_functionBody;

const auto profileData = functionBody->GetDynamicProfileInfo();

const auto instance = GetReg(playout->R1);

LdElemInfo ldElemInfo;

ldElemInfo.arrayType = ValueType::Uninitialized.Merge(instance);

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

Var length = JavascriptOperators::OP\_GetLength(instance, GetScriptContext());

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

ldElemInfo.elemType = ldElemInfo.elemType.Merge(length);

profileData->RecordElementLoad(functionBody, playout->profileId, ldElemInfo);

SetReg(playout->R0, length);

}

#endif

Var InterpreterStackFrame::GetFunctionExpression()

{

// Make sure we get the boxed function object if is there, (or the function itself)

return StackScriptFunction::GetCurrentFunctionObject(this->function->GetRealFunctionObject());

}

template <class T>

void InterpreterStackFrame::OP\_LdFunctionExpression(const unaligned T \* playout)

{

SetRegAllowStackVar(playout->R0, this->GetFunctionExpression());

}

template <class T>

void InterpreterStackFrame::OP\_StFunctionExpression(const unaligned T \* playout)

{

OP\_StFunctionExpression(GetReg(playout->Instance), GetReg(playout->Value), playout->PropertyIdIndex);

}

template <class T>

void InterpreterStackFrame::OP\_StLocalFunctionExpression(const unaligned T \* playout)

{

OP\_StFunctionExpression(this->localClosure, GetReg(playout->Instance), playout->PropertyIdIndex);

}

void InterpreterStackFrame::OP\_StFunctionExpression(Var instance, Var value, PropertyIdIndexType index)

{

JavascriptOperators::OP\_StFunctionExpression(instance,

this->m\_functionBody->GetReferencedPropertyId(index), value);

}

template <class T>

void InterpreterStackFrame::OP\_LdNewTarget(const unaligned T\* playout)

{

if (this->m\_callFlags & CallFlags\_NewTarget)

{

SetRegAllowStackVar(playout->R0, (Js::RecyclableObject\*)this->m\_inParams[this->m\_inSlotsCount]);

}

else if (this->m\_callFlags & CallFlags\_New)

{

SetRegAllowStackVar(playout->R0, this->GetFunctionExpression());

}

else

{

SetReg(playout->R0, this->GetScriptContext()->GetLibrary()->GetUndefined());

}

}

Var InterpreterStackFrame::OP\_Ld\_A(Var aValue)

{

return aValue;

}

Var InterpreterStackFrame::LdEnv() const

{

return this->function->GetEnvironment();

}

void InterpreterStackFrame::SetEnv(FrameDisplay \*frameDisplay)

{

this->function->SetEnvironment(frameDisplay);

}

Var InterpreterStackFrame::OP\_LdLocalObj()

{

if (!VirtualTableInfo<ActivationObject>::HasVirtualTable(this->localClosure) &&

!VirtualTableInfo<ActivationObjectEx>::HasVirtualTable(this->localClosure))

{

Js::Throw::FatalInternalError();

}

return this->localClosure;

}

#ifndef TEMP\_DISABLE\_ASMJS

template <typename T2>

void InterpreterStackFrame::OP\_StArr(uint32 index, RegSlot value)

{

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

if (index < (arr->GetByteLength()))

{

BYTE\* buffer = arr->GetBuffer();

\*(T2\*)(buffer + index) = (T2)GetRegRaw<T2>(value);

}

}

#endif

template<> inline double InterpreterStackFrame::GetArrayViewOverflowVal()

{

return \*(double\*)&NumberConstants::k\_Nan;

}

template<> inline float InterpreterStackFrame::GetArrayViewOverflowVal()

{

return (float)\*(double\*)&NumberConstants::k\_Nan;

}

template<typename T> T InterpreterStackFrame::GetArrayViewOverflowVal()

{

return 0;

}

template <class T>

void InterpreterStackFrame::OP\_LdArrFunc(const unaligned T\* playout)

{

Var\* arr = (Var\*)GetNonVarReg(playout->Instance);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex);

m\_localSlots[playout->Value] = arr[index];

}

#ifndef TEMP\_DISABLE\_ASMJS

template <typename T2>

void InterpreterStackFrame::OP\_LdArr(uint32 index, RegSlot value)

{

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

T2 val = index < (arr->GetByteLength()) ? \*(T2\*)(buffer + index) : GetArrayViewOverflowVal<T2>();

SetRegRaw<T2>(value, val);

}

#endif

template <class T, typename T2>

void InterpreterStackFrame::OP\_StSlotPrimitive(const unaligned T\* playout)

{

T2\* buffer = (T2\*)GetNonVarReg(playout->Instance);

buffer[playout->SlotIndex] = GetRegRaw<T2>(playout->Value);

}

template <class T>

Var InterpreterStackFrame::OP\_LdAsmJsSlot(Var instance, const unaligned T\* playout)

{

return ((Var\*)instance)[playout->SlotIndex];

}

template <class T, typename T2>

void InterpreterStackFrame::OP\_LdSlotPrimitive(const unaligned T\* playout)

{

T2\* buffer = (T2\*)GetNonVarReg(playout->Instance);

SetRegRaw<T2>(playout->Value, buffer[playout->SlotIndex]);

}

template <class T>

void InterpreterStackFrame::OP\_LdArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

(this->\*LdArrFunc[playout->ViewType])(index, playout->Value);

}

template <class T>

void InterpreterStackFrame::OP\_LdArrConstIndex(const unaligned T\* playout)

{

const uint32 index = playout->SlotIndex;

Assert(playout->ViewType < 8);

(this->\*LdArrFunc[playout->ViewType])(index, playout->Value);

}

template <class T>

void InterpreterStackFrame::OP\_StArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

(this->\*StArrFunc[playout->ViewType])(index, playout->Value);

}

template <class T>

void InterpreterStackFrame::OP\_StArrConstIndex(const unaligned T\* playout)

{

const uint32 index = playout->SlotIndex;

Assert(playout->ViewType < 8);

(this->\*StArrFunc[playout->ViewType])(index, playout->Value);

}

Var InterpreterStackFrame::OP\_LdSlot(Var instance, int32 slotIndex)

{

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if ((uintptr\_t)((Var\*)instance)[ScopeSlots::EncodedSlotCountSlotIndex] <= (uintptr\_t)(slotIndex - ScopeSlots::FirstSlotIndex))

{

Js::Throw::FatalInternalError();

}

}

return ((Var\*)(instance))[slotIndex];

}

template <class T>

Var InterpreterStackFrame::OP\_LdSlot(Var instance, const unaligned T\* playout)

{

return OP\_LdSlot(instance, playout->SlotIndex);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdSlot(instance, playout->SlotIndex);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

template <class T>

Var InterpreterStackFrame::OP\_LdInnerSlot(Var slotArray, const unaligned T\* playout)

{

return OP\_LdSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdInnerSlot(Var slotArray, const unaligned T\* playout)

{

Var value = OP\_LdInnerSlot(slotArray, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

template <class T>

Var InterpreterStackFrame::OP\_LdInnerObjSlot(Var slotArray, const unaligned T\* playout)

{

return OP\_LdObjSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdInnerObjSlot(Var slotArray, const unaligned T\* playout)

{

Var value = OP\_LdInnerObjSlot(slotArray, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

Var InterpreterStackFrame::OP\_LdFrameDisplaySlot(Var instance, int32 slotIndex)

{

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if (((FrameDisplay\*)instance)->GetLength() < slotIndex - Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Var))

{

Js::Throw::FatalInternalError();

}

}

return ((Var\*)instance)[slotIndex];

}

template <class T>

Var InterpreterStackFrame::OP\_LdEnvObj(Var instance, const unaligned T\* playout)

{

return OP\_LdFrameDisplaySlot(instance, playout->SlotIndex);

}

template <class T>

Var InterpreterStackFrame::OP\_LdEnvSlot(Var instance, const unaligned T\* playout)

{

Var slotArray = OP\_LdFrameDisplaySlot(instance, playout->SlotIndex1);

return OP\_LdSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdEnvSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdEnvSlot(instance, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

Var InterpreterStackFrame::OP\_LdObjSlot(Var instance, int32 slotIndex)

{

Var \*slotArray = \*(Var\*\*)((char\*)instance + DynamicObject::GetOffsetOfAuxSlots());

return slotArray[slotIndex];

}

template <class T>

Var InterpreterStackFrame::OP\_LdObjSlot(Var instance, const unaligned T\* playout)

{

return OP\_LdObjSlot(instance, playout->SlotIndex);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdObjSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdObjSlot(instance, playout->SlotIndex);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

template <class T>

Var InterpreterStackFrame::OP\_LdEnvObjSlot(Var instance, const unaligned T\* playout)

{

Var slotArray = OP\_LdFrameDisplaySlot(instance, playout->SlotIndex1);

return OP\_LdObjSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdEnvObjSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdEnvObjSlot(instance, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

void InterpreterStackFrame::OP\_StSlot(Var instance, int32 slotIndex, Var value)

{

// We emit OpCode::StSlot in the bytecode only for scope slot arrays, which are not recyclable objects.

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if ((uintptr\_t)((Var\*)instance)[ScopeSlots::EncodedSlotCountSlotIndex] <= (uintptr\_t)(slotIndex - ScopeSlots::FirstSlotIndex))

{

Js::Throw::FatalInternalError();

}

}

((Var\*)(instance))[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StEnvSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

Var slotArray = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StSlot(slotArray, slotIndex2, value);

}

void InterpreterStackFrame::OP\_StSlotChkUndecl(Var instance, int32 slotIndex, Var value)

{

// We emit OpCode::StSlot in the bytecode only for scope slot arrays, which are not recyclable objects.

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if ((uintptr\_t)((Var\*)instance)[ScopeSlots::EncodedSlotCountSlotIndex] <= (uintptr\_t)(slotIndex - ScopeSlots::FirstSlotIndex))

{

Js::Throw::FatalInternalError();

}

}

OP\_ChkUndecl(((Var\*)instance)[slotIndex]);

((Var\*)(instance))[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StEnvSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

Var slotArray = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StSlotChkUndecl(slotArray, slotIndex2, value);

}

void InterpreterStackFrame::OP\_StObjSlot(Var instance, int32 slotIndex, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var \*slotArray = \*(Var\*\*)((char\*)instance + DynamicObject::GetOffsetOfAuxSlots());

slotArray[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StObjSlotChkUndecl(Var instance, int32 slotIndex, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var \*slotArray = \*(Var\*\*)((char\*)instance + DynamicObject::GetOffsetOfAuxSlots());

OP\_ChkUndecl(slotArray[slotIndex]);

slotArray[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StEnvObjSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var envInstance = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StObjSlot(envInstance, slotIndex2, value);

}

void InterpreterStackFrame::OP\_StEnvObjSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var envInstance = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StObjSlotChkUndecl(envInstance, slotIndex2, value);

}

Var InterpreterStackFrame::OP\_LdStackArgPtr(void)

{

// Return the address of the first param after "this".

return m\_inParams + 1;

}

// Called for the debug purpose, to create the arguments object explicitly even though script has not declared it.

Var InterpreterStackFrame::CreateHeapArguments(ScriptContext\* scriptContext)

{

return JavascriptOperators::LoadHeapArguments(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, &this->m\_inParams[1], scriptContext->GetLibrary()->GetNull(), (PropertyId\*)scriptContext->GetLibrary()->GetNull(), scriptContext, false);

}

template <bool letArgs>

Var InterpreterStackFrame::LdHeapArgumentsImpl(Var argsArray, ScriptContext\* scriptContext)

{

Var frameObj;

if (m\_functionBody->HasScopeObject() && argsArray != scriptContext->GetLibrary()->GetNull())

{

frameObj = this->localClosure;

Assert(frameObj);

}

else

{

frameObj = scriptContext->GetLibrary()->GetNull();

}

Var args = JavascriptOperators::LoadHeapArguments(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, &this->m\_inParams[1], frameObj, (PropertyId\*)argsArray, scriptContext, letArgs);

this->m\_arguments = args;

return args;

}

Var InterpreterStackFrame::OP\_LdHeapArguments(Var argsArray, ScriptContext\* scriptContext)

{

return LdHeapArgumentsImpl<false>(argsArray, scriptContext);

}

Var InterpreterStackFrame::OP\_LdLetHeapArguments(Var argsArray, ScriptContext\* scriptContext)

{

return LdHeapArgumentsImpl<true>(argsArray, scriptContext);

}

Var InterpreterStackFrame::OP\_LdHeapArgsCached(ScriptContext\* scriptContext)

{

uint32 formalsCount = this->m\_functionBody->GetInParamsCount() - 1;

Var args = JavascriptOperators::LoadHeapArgsCached(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, formalsCount, &this->m\_inParams[1], this->localClosure, scriptContext, false);

this->m\_arguments = args;

return args;

}

Var InterpreterStackFrame::OP\_LdLetHeapArgsCached(ScriptContext\* scriptContext)

{

uint32 formalsCount = this->m\_functionBody->GetInParamsCount() - 1;

Var args = JavascriptOperators::LoadHeapArgsCached(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, formalsCount, &this->m\_inParams[1], this->localClosure, scriptContext, true);

this->m\_arguments = args;

return args;

}

Var InterpreterStackFrame::OP\_LdArgumentsFromFrame()

{

return this->m\_arguments;

}

void\* InterpreterStackFrame::OP\_LdArgCnt()

{

return (void\*)m\_inSlotsCount;

}

Var InterpreterStackFrame::OP\_ResumeYield(Var yieldDataVar, RegSlot yieldStarIterator)

{

ResumeYieldData\* yieldData = static\_cast<ResumeYieldData\*>(yieldDataVar);

RecyclableObject\* iterator = yieldStarIterator != Constants::NoRegister ? RecyclableObject::FromVar(GetNonVarReg(yieldStarIterator)) : nullptr;

return JavascriptOperators::OP\_ResumeYield(yieldData, iterator);

}

void\* InterpreterStackFrame::operator new(size\_t byteSize, void\* previousAllocation)

{

//

// Placement 'new' is used by InterpreterStackFrame to initialize the C++ object on the RcInterpreter's

// program stack:

// - Unlike most other allocations, the previously allocated memory will \_\_not\_\_ be

// zero-initialized, as we do not want the overhead of zero-initializing the frame when

// calling functions.

//

// NOTE: If we wanted to add C# semantics of all locals are automatically zero-initialized,

// need to determine the most efficient mechanism for this.

//

return previousAllocation;

}

void \_\_cdecl InterpreterStackFrame::operator delete(void \* allocationToFree, void \* previousAllocation)

{

AssertMsg(allocationToFree == previousAllocation, "Memory locations should match");

AssertMsg(false, "This function should never actually be called");

}

} // namespace Js

// Make sure the macro and the layout for the op is consistent

#define DEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define DEF3(x, op, ...) DEF2(x, op)

#define EXDEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define EXDEF3(x, op, ...) EXDEF2(x, op)

#define DEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define DEF3\_WMS(x, op, ...) DEF2\_WMS(x, op)

#define EXDEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define EXDEF3\_WMS(x, op, ...) EXDEF2\_WMS(x, op)

#include "InterpreterHandler.inl"

// Make sure the macro and the layout for the op is consistent

#define DEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define DEF3(x, op, ...) DEF2(x, op)

#define DEF4(x, op, ...) DEF2(x, op)

#define EXDEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define EXDEF3(x, op, ...) EXDEF2(x, op)

#define EXDEF4(x, op, ...) EXDEF2(x, op)

#define DEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define DEF3\_WMS(x, op, ...) DEF2\_WMS(x, op)

#define DEF4\_WMS(x, op, ...) DEF2\_WMS(x, op)

#define EXDEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define EXDEF3\_WMS(x, op, ...) EXDEF2\_WMS(x, op)

#define EXDEF4\_WMS(x, op, ...) EXDEF2\_WMS(x, op)

#include "InterpreterHandlerAsmJs.inl"