Xinaos

A new compiler and runtime framework for detecting and implementing parallelism in C# programs

matt.tait@gmail.com

Final Year Project

June 2009

Abstract

The recent rise of parallel architectures in the mainstream has led to a proliferation of badly-written, difficult to debug and maintain programs written by programmers who introduce parallelism into their applications in an ad-hoc manner, without fully understanding the implications of parallelism on the program as a whole.

In this project I reject wholesale the current pessimistic paradigm that parallelism cannot be introduced into a program by a compiler, and not only show that in principle such a compiler can be built, but introduce with this project a small compiler which does just this.

Project Archive

This report and the full source code for all of the project is available at

http://www.doc.ic.ac.uk/~mt806/finalyearproject/

The project is split into two major sections – the *Xinaos* compiler which transforms C# code into XIL binary files for the runtime, including the tokeniser, lexxer, optimizing syntax tree builder, assembly generator, verifier, optimizer, parallel detection engine and binary output; and the *Xinaos* runtime which is responsible for loading the XIL binary files into memory and executing them, including maintaining the garbage-collected heap, just-in-time compiler and method thunks required for allowing calls from the runtime into the operating system.

The *Xinaos* compiler, optimizer and verifier are roughly 20,000 lines of C# spread over 226 files. The *Xinaos* runtime is written in C++ with major sections written in x86 assembler as part of the exception manager and Just-in-time compiler.

The runtime is approximately 7,000 lines of C++ code over 89 files, which includes the automatic garbage collector, just-in-time compiler and other runtime program code.

Acknowledgements

I would like to thank Professor Sophia Drossopoulou for supervising this project.

I would also like to thank Professor Paul Kelly – his enthusiasm and energy for compilers and processors were an inspiration to me for this project, particularly during my brief time as part of the Software Performance Optimizations research group.

.

There's been a lot of research into [automatic parallelization] and discovering there is parallelism and actually using it. It has been especially done in the context of languages such as Haskell without side-effects where you know you can do any computation in parallel and it's safe to do so. But I'd also say that most of this research [in languages such as VB and C#] hasn't worked out as well as you may hope, and it seems like a really hard problem, and I would be surprised to see it ever working in practice.

Daan Leijen
Microsoft Research
Senior Researcher and developer for Microsoft's Task Parallel Library (MsPFX)

1. Introduction

Parallel-computing has been available as part of scientific and large-scale business computing for many decades, although it is only with the recent introduction of dual-core, quad-core and the near release of 80-core processors [1] to the mainstream consumer market that parallelism has become so important for non-specialist programmers to consider and implement as part of mainstream program development.

It is with this in mind that it has become the norm to teach students and programmers to introduce parallelism into their programs - either by manual control of threads, locks and mutexes or by using third-party libraries or software to spread the tasks over a number of processors to perform the task in parallel – either by dynamic runtime allocation such as is the case with the Microsoft Parallel Runtime Library for .NET [2], or statically as is the case with OpenMP for languages such as C and C++ [3].

Whilst parallelism is certainly necessary for the general speed-up of computing which Moore's law can no longer provide and on which computing has relied for its quick growth in the late 20th and early 21st century, I believe that it is fundamentally the wrong approach to introduce this parallelism manually and explicitly, since misapplication of parallelism can introduce errors which are hard to debug or even detect, and which reduce the legibility and thus self-documentation of code. Indeed, in some cases a misuse of parallelism can cause the entire application to slow down.

I believe that for the vast majority of programs, parallelism is essentially an implementation detail – an optimization if you will, rather than something to be explicitly coded for, and that explicitly coding for parallelism hides the intent of the programmer which may make other, perhaps more significant optimizations impossible.

It is for this reason that I reject the mainstream opinion of asking programmers to explicitly mark parallelism opportunities in their program; instead asking programmers to indicate properties in their programs which the compiler can choose to use to introduce coarse-grained parallelism, and propose a radical alternative to the mainstream mechanism for performing parallelism.

In this project I introduce a new compiler and intermediate language which can detect parallelism in C# code. By way of testing the project I also introduce a fully garbage collected, just-in-time compiled runtime which loads and runs binaries written in the intermediate language. The task of scheduling and work-balancing code in the background is performed by a new parallel library also included with this project.

The intermediate language (XIL), whose full listing is given in Appendix F is a variation of the MSIL and Java-byte code intermediate languages, with some subtle changes, including introducing for-loops whose iteration space is upper-bounded before the first iteration of the loop, and which can be marked by the runtime as parallelizable.

Like MSIL and Java-byte code the XIL intermediate language is type-safe stack-abstraction of a machine, although unlike MSIL the XIL intermediate language does not directly allow pointer-arithmetic or direct loading of variable addresses.

These changes, although seemingly minor, allow us to detect parallelism when the program is expressed in XIL. This is in essence the main difference between XIL and the MSIL and Java-byte code intermediate languages. Whereas Java byte code and MSIL are an abstraction *up* from general machine-code, and are therefore designed to be concise, efficient and map easily onto multiple machine architectures, XIL is a step *down* from high-level languages, and is designed with optimizations and expressing user intention in mind. It is for this reason that it became necessary to create a new runtime and compiler, rather than reusing an existing one.

Writing any new compiler, Just-in-time compiler, garbage collector or parallel library is in itself a significant task, and writing them together posed a great deal of challenges. It should therefore be no surprise to the reader that I developed and tested each of these independent from each other in order to detect potential errors.

It should also be no surprise to the reader that a heavy emphasis was placed on proving and testing all parts of the project to exhaustion. Inside the garbage collector, for example, over a quarter of the instructions executed on debug builds are simply guaranteeing correctness of each stage.

For real applications, the cost of checking can become punishingly expensive, and as such these checks and balances can be turned off for entire components of the runtime or compiler via one of the configuration options – thus leaving the conditions in place as formal comments for any future changes or alterations that may be made.

In this document I discuss some of the more important and exotic changes to the conventional compilation/runtime environment, in particular with regards to compilation and optimization, and detail some of the challenges and design decisions taken with the libraries and runtime itself.

In Chapters 5 and 6 I detail the optimizations taken by the front-end compiler, ranging from the simple (such as constant-folding) to the reasonably complex (such as converting heap objects to stack-objects, detecting parallelism and inlining methods)

In Chapter 7 I detail the design and design choices of the garbage collector. Even though the *Xinaos* garbage collector is (currently) merely a single-generational and non-incremental garbage collector, there are still some improvements over the status quo, including a deterministic ordering on finalization objects following a common ownership pattern which generalizes and makes obsolete the two-tiered finalization queue in the .NET framework [4]. Appendix B shows a sample heap compaction taken by the garbage collector, which should demonstrate to any reader brave enough to check that the garbage collector is correct.

In Chapter 9 I detail the design and design choices of the parallel library, which is responsible for load-balancing tasks across multiple cores and executing loops in parallel. The parallel library is written in pure C#, and is actually used explicitly by the front-end compiler, resulting in a ~20% speedup of program compilation.

.Appendix F and G are a reference for the binary format of the XIL instruction set and XIL program file respectively.

Motivation

Consider the code for a loop performing some action over all elements in some collection

```
foreach(var element in collection){
  action(element);
}
```

A natural question from the point-of-view of writing compiler optimizations is under which circumstances the above loop can be made parallel.

There are two considerations that affect the parallelizability of the loop. The first is to do with direct side-effects, such as writing to the console, creating a window, writing to a file or destructively reading from a socket. If *action* causes, or has the potential to cause such side-effects then the loop is not parallelizable.

The second consideration has to do with whether one iteration could potentially interact with a future iteration of the loop in such a way that re-ordering the order of iterations of the loop would change the behavior of the function, for example if one iteration writes to a variable which is read and used by a later iteration, then these iterations cannot be swapped.

Direct dependencies

Direct dependencies, such as writing to the console, sending a page to a printer or playing a noise are relatively simple to detect in managed languages where pointers and interrupts have been outlawed. In such frameworks, side-effects can only ever occur from interacting with the outside world via external calls (termed a PInvoke [5] in C# parlance). Calling a PInvoke is therefore a side-effect, and any method which calls such a function is itself a method which could cause side-effects. By forming the transitive closure of this “causing side-effects” property of a method, we can detect all methods which can cause a side-effect quickly and efficiently at compile-time.

One thing to note here is that not all calls out of managed code are necessarily direct side-effects – reading the title of a window is safely parallelizable, for instance. In this case we would require the programmer to mark the PInvoke responsible for reading the title of a window as being a non-side-effecting external call, and thus that calls to the PInvoke may overlap. Such markings would need to be made on a case-by-case basis, since a PInvoke is simply a named method signature into a DLL.

The responsibility of ensuring that PInvokes which *do* cause side-effects or which may object to being called concurrently on multiple threads lies with the programmer; however as the Microsoft implementation of C# has shown, a large and feature-rich base-class library which hides away the implementation of the PInvokes allows most programmers to avoid the need to call PInvokes for the vast majority of their code. Moreover, such a marking on PInvoke signatures is an efficiency marking on the method, and thus failing to mark a side-effect free PInvoke signature as such does not impair the correctness of the program; it merely prevents certain parallel optimizations from being performed.

Although certainly not mainstream, there is some research which shows that even such side-effects can in principle be parallelized by introducing a notion of anti-side-effects [6]. Although the *Xinaos* compiler does not attempt to perform anti-side-effects (and indeed most standard operating systems are not conducive to such an approach), the compiler may de-fuse loops into a parallel segment and a non-parallel segment; typically moving the computation of intermediary results and arguments into the parallel segment before the non-parallel side-effects are called in order, although this leads to some problems if the intermediary results can cause exceptions. This feature is discussed in more detail in the Optimizations chapter.

Memory dependencies

The other important deciding factor on whether the loop is parallelizable depends on the order in which reads and writes to memory occur within the loop – specifically if the reads and writes access the same position. There are generally three forms of hazard:

True-dependencies: read after write

If an iteration writes a value which is read by a later iteration, then the earlier, writing iteration must occur before the later reading one in order for the later iteration to definitely obtain the correct value.

Anti-dependencies: write after read.

If an iteration reads a value which is overwritten by a later iteration, then if the order of iterations is interchanged then the earlier iteration may obtain the value from the later iteration rather than the value that was there before.

Output-dependence: write after write.

If an iteration writes to a value and a later iteration also writes to the same location, then the later write must take precedence. Note that there may be some circumstances which allow optimizations to completely avoid writing the output for the earlier iteration, which would alleviate this kind of output-dependence.

The problem of detecting memory dependencies has been well documented [7], although it is interesting to note that the majority of the research into how to circumvent such problems comes from the hardware rather than the software industry. Modern processors take code which looks like it should be run sequentially, however they tend to perform a great deal of instruction reordering in order to maximize the parallelism at the instruction level so as to gain an effective improvement of processor performance.

Arguably the most important such hardware innovation towards the inter-instruction dependency problem is given in the *Tomasulo* processor design [8] which precedes modern day register-renaming algorithms as a mechanism of allowing pending operations to have their operands forwarded to them directly, rather than going through a strictly named register. The design solves the true-dependency problem for processors which make use of instruction-level parallelism.

The *Xinaos* compiler does not manage to perform optimizations of this kind over macro code-structures such as loops, but it does perform such optimizations on linear code (see linear code optimizations

Chapter 4) – avoiding the read altogether in the read-after-write dependency scenario. This has the added advantage that if the location is not read from at a later stage before a subsequent write, the write can additionally be dropped. The *XIL* intermediary language has been designed with optimizations in mind, and this form of optimization can be straightforwardly applied when the program is translated into XIL.

The other problems – how to avoid write after read and write after write has a different solution in hardware, with modern processors tending to use load and store queues to protect in-flight instructions from writing out of order back to the system and detecting if a write after read dependency violation has taken place whereupon a rollback would occur which keeps the semantic of the instructions being performed "in order".

There has been some recent research which shows that a similar scheme could be used in software to help increase the potential parallelism in loops by deferring all writes to memory over a loop and performing the writes in order at the end of the loop – thus enabling the loop itself to run in parallel [9]. Although an exact equivalent in the *Xinaos* compiler does not exist – largely because the cost of deferring writes is expensive – the *Xinaos* compiler can de-fuse loops, pushing the side-effecting part of the loop into a small synchronous loop, and the generation of intermediary results for the side-effecting functions (e.g. values to be written in console writes) into a loop which can potentially be performed in parallel.

Disjoint collections

Consider the parallelizability of the foreach block in the following code:

```
void disposeAll(IEnumerable<ComplexNumber> collection){
  foreach(ComplexNumber d in collection)
    d.RealPart += d.ComputeNorm();
}
```

The operation performed in the loop clearly has no direct side-effects. The question is can it possibly have memory side-effects? The answer is that it depends on whether a ComplexNumber could possibly appear in the collection twice.

Because the C# language does not natively support a *disjoint* keyword, the *Xinaos* compiler relies upon an aggressive inlining policy to bring loop operations closer to the loop instantiation, and makes judicious use of automatically detected properties of types (such as immutability) and methods (such as how parameters may "leak" out of scope of a method before it returns) to detect and implement parallelism over loops.

Non-disjoint collection parallelism

In some cases where memory disjointness cannot be guaranteed, the operation is such that it does not matter whether or not the loop can be performed in parallel. Take the following code, for example:

```
int a = 10;
int b = 4;
int c = 1;
for(int i = 0; i < 10 ; i++){
  a = a;
  b = b + 1234;
  c = Math.Pow(c, 14);
  d = 2*d*d - 1;
}
```

In this case the variable ***a*** is read and written by every loop iteration. Consequently the loop suffers from read after write, write after write and write after read memory hazards. It is perhaps surprising then that the iterations of the loop can be interchanged. The reason for this comes from the interesting observation about the commutability of the operations applied to ***a***, ***b, c*** and ***d***. It should be pointed out that this does not necessarily make the loop parallelizable – merely that the loop's iteration space can be arbitrarily reordered.

A partial classification of permutable rational functions is given by Ritt [10] and Julia using algebraic number theory, and in particular shows that if i_1 and i_2 are polynomials of the same field then they commute iff they are both *trivial (*such as ***a*** above*), straightforward* (such as ***b*** above), both iterates of the same polynomial of the element to which they write (such as ***c*** above), or the general class of Tchebycheff polynomials (such as ***d*** above) [11] [12].

Non-loop based parallelism

While we are mostly concerned with loop-based parallelism when it comes to detecting and implementing parallelism, there is another major class of parallelism that can be achieved by predictive background computation of intermediary results. Consider the code for naively computing the n^{th} element of the Fibonacci sequence:

```
int fib(int n){
  if(n < 2) return n;
  return fib(n-1) + fib(n-2);
}
```

Notice that the addition operation does not attempt to compute `fib(n-2)` until `fib(n-1)` has been entirely computed. We may notice, however, that when we reach the addition operation, it is already clear `fib(n-2)` will be needed at some stage, even if `fib(n-1)` has not yet been computed. It should also be clear that the two operations do not mutually depend on any reference-type object or static variable and cannot cause a side-effect (other than perhaps throwing an exception), and thus are completely independent.

We can now begin to see that we could somehow mark the `fib(n-2)` operation as an operation that will be needed later, and which should (if possible) be issued in the background. Using the `Future<T>` class we can write the code in the following way:

```
int pfib(int n){
  if(n < 2) return n;
  Future<int> fib_n2 = new Future<int>( () => pfib(n-2) );
  return pfib(n-1) + fib_n2.Value;
}
```

The **Future<T>** class takes as its argument a delegate (closure) which yields a value of **T** and takes no parameters, and yields the result of the method when the **.Value** property on the **Future** is called. What makes the **Future** class interesting is that it can compute the result by calling the method at any point between when the **Future** is created and when **Value** is called. If **Value** is called and the **Future** has not already computed the result, the **Future** executes the method synchronously.

In practice this means that when calling pfib above with some large parameter, the first few iterations are performed entirely on different cores (the mechanism by which this occurs is discussed in the parallel library chapter at the end of this document) and the lower-down iterations are performed synchronously. This leads to a near-optimal load distribution on the cores and a significant speed-up of the program as a whole

The *Xinaos* Framework

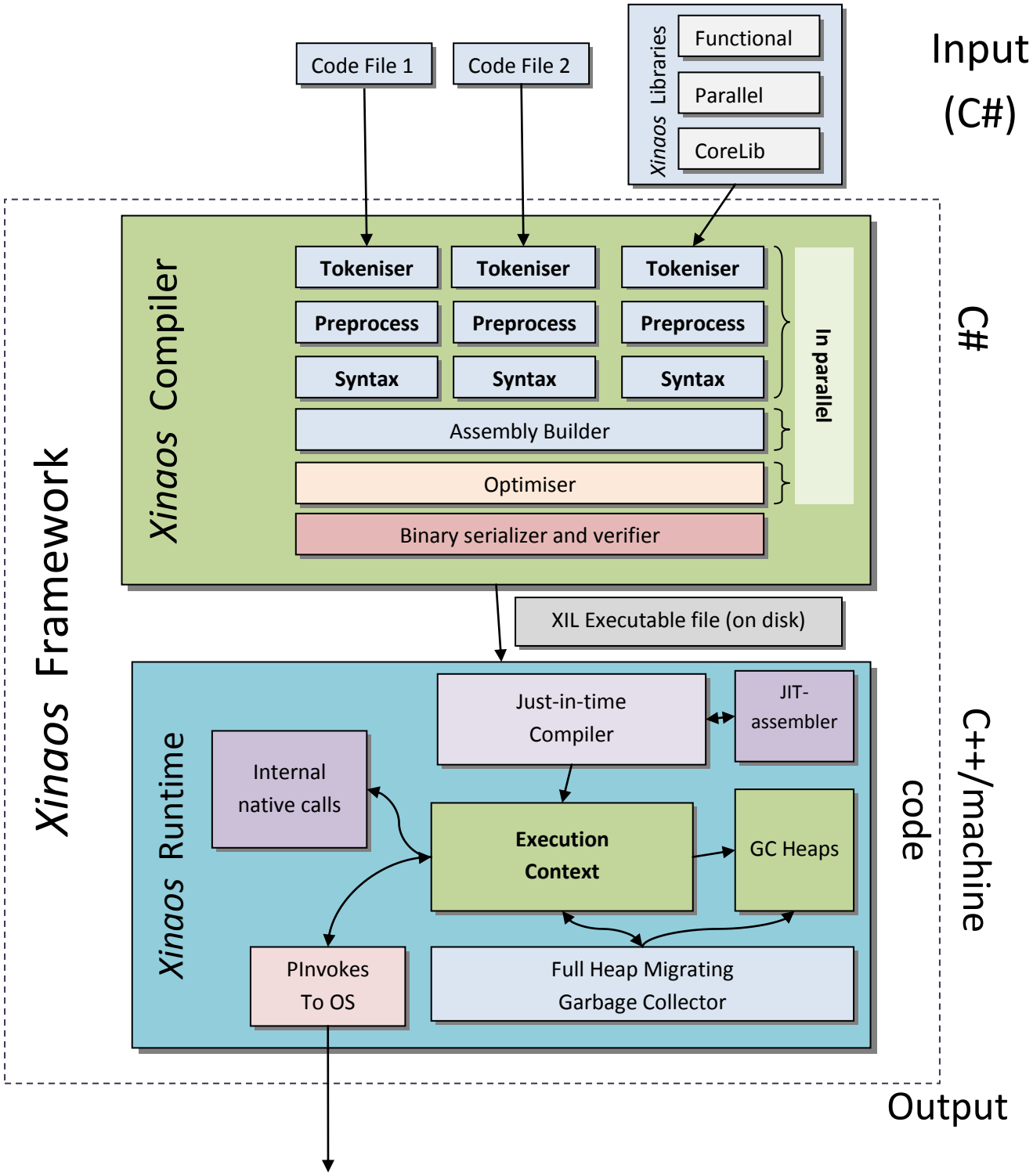


Figure 1: Layout of components in *Xinaos* framework

The *Xinaos* Intermediate language

The *Xinaos* framework is the collection of programs and libraries that make up the compiler, runtime and supporting libraries, displayed diagrammatically in Figure 1, and is the environment in which XIL programs are constructed and executed.

In order to properly understand the motivation and eventual construction of the various components in the *Xinaos* framework, it is necessary for us to first discuss the motivation, design and working of the XIL programs which are the *raison d'être* of the framework itself.

Firstly let us discuss what it is that we want XIL to *achieve*; that is, what we want the *XIL* intermediate language to do that could not be provided by an equivalent alternative solution such as Java-byte code or MSIL.

XIL should be concise and intuitive

The first goal is a protection against XIL becoming like the x86 instruction set [1], where efficiency has been the driving force behind opcode additions. x86 instructions have variable length opcodes and there is a large degree of redundancy allowing clever programmers or optimizers to create very fast programs. While laudable in its aim, the x86 instruction set is now huge and much of it goes unused. The principle that simple instructions are the most commonly used has come back to haunt x86 processor developers, who have to expend large amounts of real-estate (and thus clock-cycle time) supporting complex instructions that nobody uses.

Although *XIL* is entirely software based, the same principle applies – having more complex instructions leads to a larger and slower just-in-time compiler.

The intuitive word in the first goal is also important. In some machine architectures a branch-delay slot holds one instruction executed *before* the jump reaches its destination. The rationale here being that branches are expensive, and the processor would otherwise remain idle, so the programmer might as well get a free instruction. Although this looks like a great idea, the ideal number of branch delay slots is equal to the length of the processor instruction pipeline, but because of backwards compatibility issues, the architecture cannot change the number of delay slots from one generation to another and thus newer generations must contain additional hardware to ensure the same behavior is followed, despite it no longer being relevant.

XIL should be easy to optimize

The second goal is perhaps the most important. Whereas Java-byte code is designed to be the *least-abstracted common abstraction* of a machine (i.e. it is the closest to a machine architecture that it can be without tying it to any actual machine architecture) and MSIL is designed to be a unified target language for the many languages that Microsoft supported at the time of MSIL's conception; XIL's main focus is on it being optimizable. By having this as the forefront goal of the language design, we can compile to XIL and perform all of our important optimizations in one stage.

In a break from the norm, the *Xinaos* compiler performs *all* of its non-machine specific optimizations on the XIL intermediate language, which means that XIL must be capable of expressing high-level concepts

such as immutability, loops and structures which are typically abstracted away in intermediate and final representations of code in the compiler.

This is different to compilers such as GCC [2]and Microsoft's Phoenix compiler [2], for instance which use multiple intermediate languages to optimize code at different stages through the compiler pipeline.

XIL should be complete and easily mapped to common machine instructions

This final goal is simply a way of insisting that the language be practicable. The XIL language must be efficiently translatable into machine code in order for it to reasonably compete with existing implementations when it comes to runtime performance.

How *XIL* differs from Java-byte code and MSIL

It is important to not always re-invent the wheel; and an obvious question arises: Is *XIL* implementable with MSIL or Java-byte code? The simple answer is no; the long answer is "it depends". Java-byte code does not allow pointer-arithmetic, and while admirable in doing so; pointer arithmetic is the bane of optimization writer's lives; it means that Java byte code is not suitable for expressing the full range of programs such as those found in C# using unsafe code. Perhaps more importantly, Java does not support either in the language or the byte-code a mechanism of dealing with delegate closures – an absolute necessity when it comes to breaking up functions for parallel blocks.

MSIL suffers from an altogether different problem. MSIL is well constructed and despite supporting multiple redundant operands (MSIL has 15 different instructions for loading a numeric constant to the stack.) The biggest problem is gaining access to the source code that would be required if new parallel passes are to be introduced. Although Microsoft's new *Phoenix* compiler does allow external developers to add optimization passes to the compiler, by the time the compiler yields control to the optimization pass, the semantic information which would be needed in order to reasonably detect parallelism inside the method has been destroyed beyond all hope.

XIL is heavily influenced by the MSIL intermediate language; and many of its instructions are directly analogous, albeit simplified. *XIL* is *not* however a subset of the MSIL intermediate language; some new additions to the XIL instruction set such as for-loop instructions (which can be marked as parallel by the optimizer) and the deliberate lack of being able to load fields and locals as pointers for pointer arithmetic (pointers are allowed in *XIL* but must always be aligned on an object on the heap or be exclusively off the heap. In practice this means that pointer arithmetic is not performed in *XIL* but is performed by the JIT to gain the associated performance boost).

These alterations allow *XIL* to search for and detect parallelism where Java-byte code and MSIL cannot.

Semantic working of *XIL*

Unlike "real" machine languages such as MIPS and the x86 instruction set (but similar to MSIL and the JVM), XIL is a stack-based virtual machine with no registers. All instructions take their arguments from the evaluation-stack and the results of evaluation are placed back onto the evaluation stack once the operation returns. The expression stack is strongly typed, which is to say all values on the expression stack have a definite type, however any value of any value can be placed on the stack.

Some operations require certain types on the stack to work – for example AddF requires two floating point values and StField requires that the value on the top of the stack is (or extends from) the value of the field being stored to.

Consider the code

```
void main(float b, int c){
  float a = b + (5.0f * (float)c);
}
```

This yields the *XIL* code:

```
Instruction         Value on the stack          Types on the stack

                    -                           -
LdParam b           b                           float
LdFloat 0.5f        b, 0.5f                     float, float
LdParam c           b, 0.5f, c                  float, float, int
IntToFloat          b, 0.5f, (float)c           float, float, float
MulF                b, 0.5f * (float)c          float, float
AddF                b + (0.5f * (float)c)       float
StLocal a           -                           -
```

Note that the IntToFloat instruction is not optional, even though the explicit cast of c from int to float is optional in C#. The reason here is that while an integer can be coerced to a floating point operation in C#, there are no implicit conversions in XIL. For this reason the integer must be explicitly converted to an integer. Note also that because MulF expects two floating point values as arguments, the omission of IntToFloat would lead to a type violation which would halt the program at runtime with an InvalidProgramException which would be raised by the *XIL* verification unit when the method was called for the first time.

A full listing of the *XIL* instruction set, including types and semantic operation can be found in Appendix F at the end of this document.

The *Xinaos* compiler

The *Xinaos* compiler is the code responsible for translating source files written in C# and writing the resulting *XIL* to a file following the *XIL* file-format laid out in Appendix F.

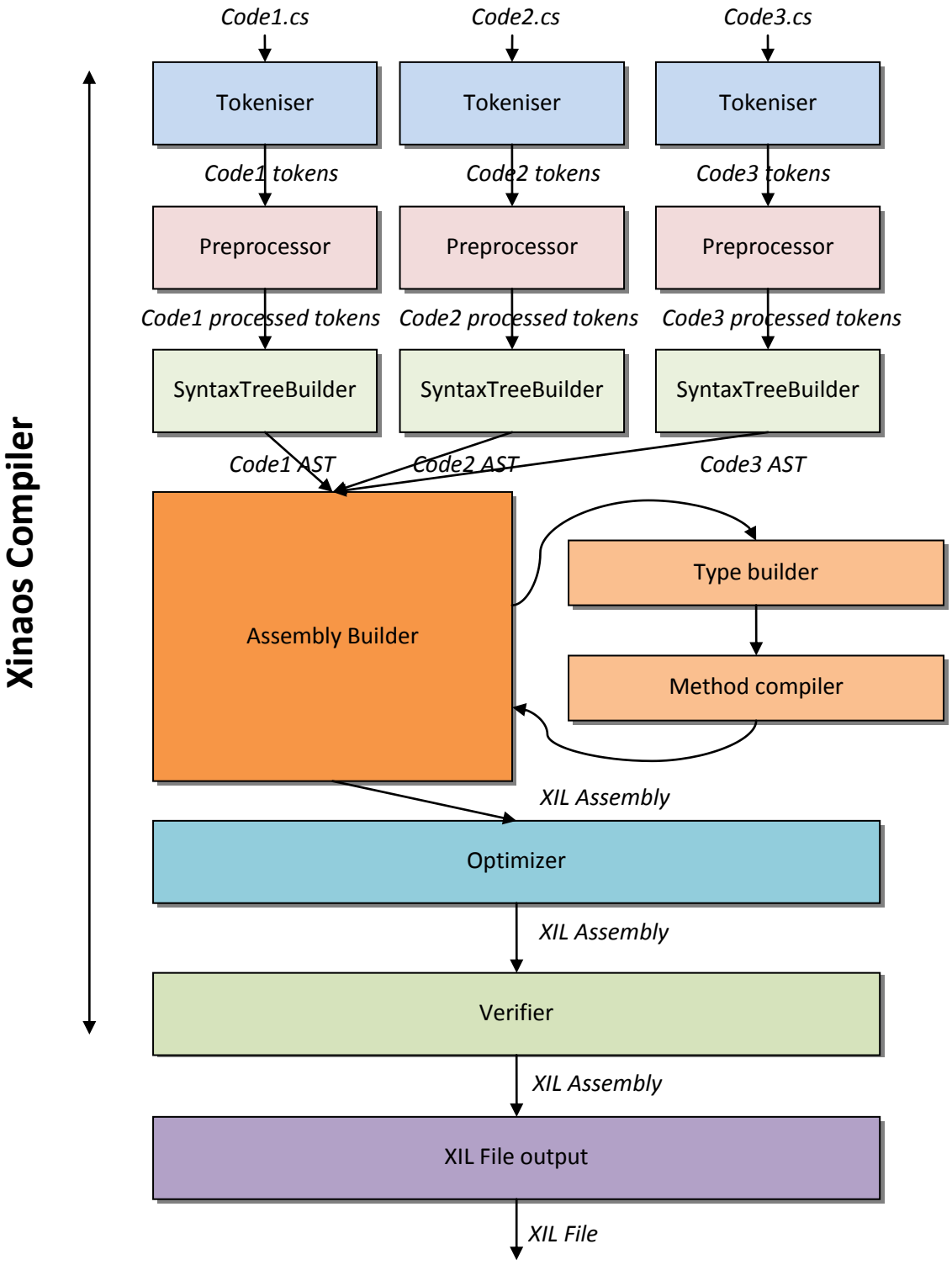


Figure 2: Layout and interactions of the components in the *Xinaos* compiler

Tokeniser

The tokeniser in the *Xinaos* framework converts a sequence of characters from an input file into an ordered sequence of terminals which are held in the file, for instance if the characters input into the stream were

```
value = 1.0;
```

The corresponding output of tokens would be

```
Identifier: 'value'
Equals
Numeric: 1.0 (floating point)
Semicolon
```

The tokeniser supports two tokenising operations thus:

Next

Gets the next token in the stream and moves the pointer in the tokeniser forward

Peek

Gets the next token in the stream, but does not progress the pointer in the tokeniser (i.e. Peek does not have side-effects)

The tokeniser is built around an internal radix-tree which is used to quickly detect and keywords in a stream in $O(n)$ time. Ideally the tokeniser would be generated directly from a finite-state machine description of the language which could be flattened to an efficient but verbose tokeniser by using a compiler-compiler. For simplicity, however, the *Xinaos* compiler makes do with an internal radix tree and a few custom coded methods for complex terminals.

In the event that the radix tree does not recognise a terminal, a series of additional tests are performed to check if the terminal is an integer, floating-point value, character literal, string literal or identifier. The output type of the tokeniser is a Token, which has as its primary field an enum TokenType which identifies the type of token as well as the position in the file that the token was encountered and optional other data (such as the value of a string literal if encountered).

Preprocessor

Unlike conventional preprocessors, the *Xinaos* preprocessor does not exist as a separate program to the compiler itself, nor does it operate fully on the file prior to the tokeniser beginning work on it. Instead the tokeniser feeds tokens into the preprocessor, which outputs its own tokens which are the preprocessed form of the input token stream.

The preprocessor has two main tasks:

1. The preprocessor swallows line comments, XML comments[1] and multiline comments
2. The preprocessor should interpret and remove #define, #undef statements and should conditionally remove tokens between #if, #ifdef, #else, #elif and #endif blocks.

Unlike the Microsoft C# compiler, the preprocessor does *not* currently remove function calls which are marked with the System.Diagnostics.Conditional attribute whose value is not defined, although in future the Conditional attribute will be detected and will be replaced with Conditional(True) or Conditional(False) depending on the definition of the literal on which the condition is made. The call would then be removed much later by the optimizer.

Rationale: This enforces constraints on conditional method calls too – including that conditional method calls (even ones not taken) must obey *requires* contracts and syntax rules. Note that code between conditional blocks not taken do not need to obey syntax rules or calling contracts (since their tokens are hidden by the preprocessor) although they must still obey token rules:

```
#define DEFINED
#undef UNDEFINED

#ifdef UNDEFINED
  a + b = 4;          // valid, since the tokens are ignored by the preprocessor
  /*
#endif
  */
  Console.Write("Still inside the #ifdef, since the #endif was commented");
#elif DEFINED
   a + b = 4          // syntax violation, since a + b is not an LValue.
   Console.Write("This will be printed");
#else
   // these are still violations, even though the tokens appear in a non-taken block.
   // This is because the error is a token-error, not a syntax error.

   "hello world       // syntax violation - string literal not closed.
   '0xFFFFFF'         // syntax violation - char literal too long
   0xFFFFFFFFFF       // syntax violation - int literal too long
   a` = b;            // syntax violation - ` is not a valid character

  / + * = ! & += a*b  // valid, since these tokens are swallowed
#endif
```

[1] XML comments are a particularly nice way of conveying descriptive information to the programmer via an IntelliSense™-style IDE. Since the Xinaos framework does not have such an IDE, and given the additional complexity of maintaining XML comments through the program, XML comments are treated as normal comments and are currently stripped by the preprocessor.

Syntax Tree builder

The syntax tree builder is split between two classes – the Syntax Tree builder and a Method Syntax Tree builder. The Syntax tree builder is a hand-written LL1-recursive decent parser and the method syntax tree builder is a hand-written look-ahead parser using a modified Djikstra's shunting yard algorithm to detect operator precedence. The reason for this design choice is simply one to do with speed of development. I intend to replace the syntax tree builder at some stage in the future, possibly with an LR or LALR tree builder, presumably with the help of a compiler-compiler such as yacc.

The syntax tree builder does not attempt to interpret any of the method code – including type information. Instead the output of the method syntax compiler is an IExpressionLayout object, and the output of the syntax tree builder is a SourceFileLayout. The intermediary leaves and nodes in the syntax tree all derive from elements inside the Xinaos.Compilers. Layout namespace, and are strictly concerned with the layout of elements in code.

Rationale

In contrast to C and C++, C# (and most other managed languages) does not care about the order in which files are processed – two source code files may legitimately depend on types declared in the other. It is for this reason that methods cannot be parsed until type information is available for them – in particular with regards to resolving the identifiers and dotted identifiers and their resulting types.

In the *Xinaos* framework all files are processed into syntax tree structures in parallel, and are then resolved by the AssemblyBuilder which compiles the method into XIL code.

A further benefit of this strategy is that the front-end can easily be made incremental – there is no reason to reparse a file into a new syntax tree if the file has not been changed – even if the identifiers in the file would be resolved to different things due to other source files changing i.e. "MyType.Value" is stored as operator(Dot, identifier(MyType), identifier(Value)), and thus can resolve to radically different things depending on how MyType and Value are resolved with respect to other files in the compilation process.

EndRationale

There are four forms of layout object:

- Layout of expressions is governed by the Xinaos.Compilers. Layout.Expressions namespace, and all derive from the IExpressionLayout interface in that namespace.
- Layout of statements is governed by the Xinaos.Compilers.Layout.Statements namespace, and all derive from the IStatementLayout interface in that namespace.
- Layout to do with type definitions, type references and interfaces are governed by the Xinaos.Compilers.Layout.TypeSystem namespace.
- Layout of methods, properties, event and indexer headers are governed by the Xinaos.Compilers.Layout.MethodSystem namespace.

Assembly Builder

Unlike languages such as C and C++ the order of file compilation[2] is not important. One consequence of this is that methods may rely on types that have not yet been fully defined when the compiler reaches their statement. As a result, compilation of the assembly takes part in phases.

Phase 1: Syntax generation

This stage encompasses tokenization, preprocessing and syntax tree generation, and is responsible for converting all source files to syntax trees describing the files. Because each file can be processed independently this stage is performed in parallel using the parallel library described in Chapter 9. Furthermore, this stage caches the results on a per-file basis, and can thus avoid parsing files which do not change between compilations, leading to a significant overall speedup.

Phase 2: Namespace and Type signature generation

In the second phase a block of namespaces is generated corresponding to the namespaces in the various files. At this stage types are added to the namespaces corresponding to the types in the layouts. These types are not populated with methods, fields or inheritance data at this stage, but are parsed for generic parameters and class type (i.e. struct/interface/class variant)

Phase 3: Ensure that the predefined types have been defined

C# defines a collection of types and interfaces which must be defined in order for some operations and statements to work. These are looked for in this order (all classes are in the namespace System unless otherwise defined)

1. Object must be defined as a class.
2. Void, Byte, Int16, Int32, Int64, Single, Double, Char, Boolean, SByte, UInt16, UInt32, UInt64, IntPtr, UIntPtr must be defined as struct types.
3. String, Delegate, MulticastDelegate, Array, Exception, Type, ValueType[3], Enum, Attribute, ParamArrayAttribute, RuntimeFieldHandle, RuntimeTypeHandle, System.Runtime.InteropServices.OutAttribute and System.Reflection.DefaultMemberAttribute must be defined as class types.
4. IDisposable, System.Collections.IEnumerable, System.Collections.IEnumerator must be defined as interfaces.

Phase 4: Normalize types for inheritance

At this stage all types have been defined, so all types are enumerated and derive either from System.Object or the type listed on their type layout. Types can additionally inherit from multiple interfaces.

Phase 5: Add Field stubs

All types have been defined, so we can add fields normalized to the correct type into each method. If fields have a value assigned to them in the code, we cannot yet parse it, but will do this later.

---

[2] With regard to files against headers – in fairness this is partly due to the more complex nature of the preprocessors in these languages.

[3] System.ValueType is not itself a value type. It is the basis of any struct type, and must be itself a class type to allow boxing of values and the unification of the type system around System.Object.

Phase 6: Add Method stubs

At this stage we add method stubs to each method. This includes the parameter list, name and return type of each method, but not the method data itself (since method calls cannot be resolved until this phase has been finished)

Phase 7: Compile Methods (and field initial values) to XIL

At this stage we can compile each method to its XIL equivalent. No optimizations are performed at this stage, so the output may be inefficient, and in particular loops are not preserved into the XIL code[4]. Method compilation is handled by the MethodCompiler class which resolves the method to XIL code.

This stage is rather involved due to the size of the C# language, but largely not interesting, and includes type coercion, type verification, identifier resolution and resolving expressions and statements to XIL code.

Parallelism in the Assembly builder

Because of the structure of the *Xinaos* compiler, although each phase must happen sequentially, most of the operations within any given phase can be done in an arbitrary order. For this reason, the high-order loop within each phase is performed as a parallel iteration[5].

---

[4] *Rationale:* The XIL "for" instruction is reserved for loops which have a known upper bound on the number of iterations that the for loop will take when the for loop is entered. For this reason loops of the form "for(int i=A;i<N;i+=P)" are in the form expected by the for loop, whereas loops of the form "for(;true;)" would not be. In general since C# does not put this condition on its for loops, we must assume the worst and use ordinary jumps and gotos to implement the loop, and if possible optimize these to "proper" loops when we reach the optimizer.

[5] For debugging this can be turned off by using the NO_PARALLEL build flag on the Xinaos.Compilers project.

The *Xinaos* Optimization engine

Although the *XIL* output of the frontend is correct, in that it can be immediately written and run by the runtime, the output is not necessarily the most efficient way of describing the program.

The task of the *Xinaos* optimization engine is to perform transformations on *XIL* code that convert the inputted *XIL* code into "better" XIL code which does the same thing. The concept of "better" here depends on the context in which the program is being compiled. Typically "better" means faster, but it can just as well mean "with less memory footprint", "taking up less space on disk" or even "running with less power" (for example on embedded devices).

The *Xinaos* optimization engine has two optimization settings, which allow the compiler to switch between "optimize for speed" and "optimize for space". The compiler also has a third option for debugging target code, in which the compiler performs a brief series of simple optimizations to the code, but does not attempt to perform more expensive loop optimizations or method-inlining. The purpose of this setting is for programmers who are developing target code and need a fast compiler turn-around time in order to test their newest alterations to the code. This is juxtaposed to release compilation, when spending several minutes making methods 10-20% faster would be a seen as a generally good idea.

There are three major types of optimization in the *Xinaos* compiler. *Linear* optimizations are optimizations that can be performed within a basic-block, and including constant folding and removing redundant loads. *Control flow* optimizations are optimizations performed over an entire method, and mainly involve reducing the number of jumps in a program, removing unreachable code and detecting "natural" for loops. *Whole program* optimizations are optimizations which analyze the context of a method within the program as a whole, and include method inlining, removing and coalescing redundant exception handling blocks and detecting parallelism over for loops.

When the *Xinaos* is compiling debug versions of code, the optimizer automatically stops optimizing after a certain number of iterations to avoid taking too long, however when compiling for release code, the optimizer continues to optimize until no further optimizations can take place.

A new (2009) paper discussion on saturation search for maximal optimization [1] is of particular interest to myself, however the late release of the paper with regards to the production cycle of the *Xinaos* compiler is such that there was not time to incorporate such a system into the *Xinaos* compiler before writing this document. Instead, an older approach effectively converting the program into a mathematical description of program behavior and mathematically simplifying the program before converting the program back to *XIL* allows some powerful optimizations to take place [2].

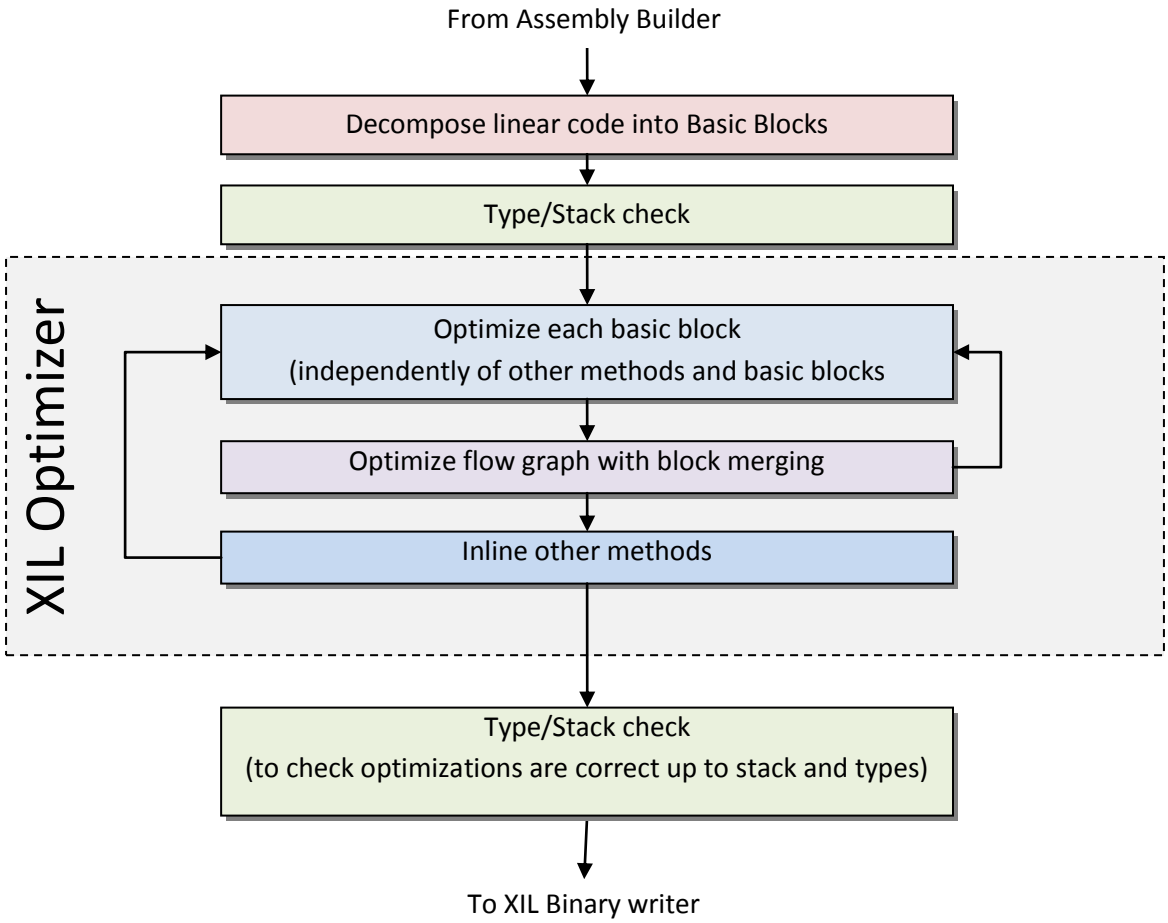


Figure 3: XIL Optimizer layout

Linear optimizations

Linear optimizations are optimizations that can be performed on code between control flow statements, and are typically small, fast and easy to check. The *Xinaos* compiler currently has 12 linear (peephole) optimizers:

Binary Constant Folding

The “binary constant folding” pass reduces binary operations where both arguments are known at compile time. Although it is fairly unusual for such operations to exist directly in the source-code, it is reasonably common for such optimizations to become available as a result of other optimizations.

	Before	**After**
General form	*Load x* *Load* y *BinaryOperation B*	*Load Q* Where Q = B(x,y)
XIL Example	LdFloat 4.5 LdFloat 5.0 Add LdNum 4 LdNum 6 Eq	LdFloat 9.5 LdFalse
C# equivalent example	(4.5f + 5.0f) (4 == 6)	(9.5f) (false)

Unary Constant Folding

The “unary constant folding” pass reduces unary operations where the argument is known at compile time.

	Before	After
General form	*Load* x *UnaryOperation U*	*Load Q* Where Q = U(x)
XIL Example	LdNum 4 IntToFloat LdNum 1 Negate LdTrue Not	LdFloat 4.0 LdNum (-1) LdTrue
C# equivalent example	float f = 4 -(1) !(true)	float f = 4.0f (-1) (false)

Constant conditional jump

The “constant conditional jump” optimization is responsible for reducing conditional jumps where the condition is constant at compile time, and is of particular importance when eliminating dead code.

	Before	After
General form	*LdTrue / LdFalse* *JmpIfTrue/JmpIfFalse lbl*	*Jmp lbl* *or* *Nop*
XIL Example	LdTrue JmpIfTrue lbl LdTrue JmpIfFalse lbl	Jmp lbl Nop
C# equivalent example	if(true) { code(); } if(false) { code(); } _label:	code(); goto _label code(); _label: (which is eliminated when detecting dead code)

Note that this often eliminates the first conditional in a *for* loop:

```
for(int i=0; i<10 ;i++){
  code(i);
}
```

becomes (in conjunction with “First Store”, “First Load” and “binary constant folding” optimizations)

```
int i = 0;
do {
  code(i);
  i++;
}while( i < 10 );
```

i.e. saving the conditional check on entry to the for loop.

First Store

The “first store” optimization is only valid on the entry block to the method, and promotes the first assignment to a variable into an entry in the method’s shadow-stack, thus obtaining the assignment “for free” during runtime.

The optimization is only valid on an assignment to a variable which has not been assigned or loaded at any previous instruction in the method.

	Before	**After**
General form	*[Type **local**]* *LdType **x*** *StLocal **local***	*[Type **local** = **x**]* *Nop*
XIL Example	[Boolean b] LdTrue StLocal [Float f = 0.4f] LdFloat 23.0 StLocal f	[Boolean b = true] Nop [Float f = 23.0] Nop
C# equivalent example	*bool b;* *b = true* *float f;* *// preload f with 0.4f* *f = 23.0*	*bool b;* *// preload b with TRUE* *float f;* *// preload f with 23.0*

First Load

The “first load” is the partner of “first store” and takes account of the fact that until the first assignment into a local variable, the value of that local variable can be found in the shadow-stack.

The optimization is only valid on an load to a variable which has not been assigned or loaded at any previous instruction in the method, and thus is only valid on the first basic block of a method.

	Before	After
General form	*[Type **local = value**]* *LdLocal **local***	*[Type **local** = **value**]* *Load **value***
XIL Example	[Boolean b = true] LdLocal b [Float f = 0.4f] LdLocal f [Object obj = null[6]] LdLocal obj	[Boolean b = true] LdTrue [Float f = 0.4f] LdFloat 0.4f [Object obj = null] LdNull
C# equivalent example	*bool b;* *// preload b with TRUE* (b) *float f;* *// preload f with 0.4* *(f)* *object obj;* *(obj)*	*bool b;* *// preload b with TRUE* *(true)* *float f;* *// preload f with 0.4* *(0.4f)* *object obj;* *(null)*

[6] Since System.Object is not a primitive value type (such as int, float or bool), no value can be promoted to its shadow stack; and thus the System.Object local variable must have the default value (null) stored in the local value slot on the shadow-stack.

Load Dup

The virtual “Dup” instruction duplicates the value on the stack, thus avoiding the recomputation of an intermediate result. While this is great for the runtime, it makes constant folding etc much harder to achieve. This optimization converts simple loads followed by a DUP into two simple loads, so that they can be potentially optimized. This optimization is reversed (to make the program faster) at the end of the compilation process.

	Before	**After**
General form	*Load* **x** *Dup*	*Load* ***x*** *Load* ***x***
XIL Example	LdNum 1 Dup Add LdLocal *a* Dup LdNum 5 Add Add	LdNum 1 LdNum 1 Add LdLocal *a* LdLocal *a* LdNum 5 Add Add
ANSI C equivalent example	*register a = 1* (a + a) *register* t = a; (t + (t + 5))	(1 + 1) (a + a + 5)

Load Pop

The virtual “pop” instruction is used to remove the top element from the evaluation stack. If the preceding expression does not have any side-effects (setting into fields, calling methods etc) then the expression can be reduced or eliminated.

“Load pop” also affects operations whose results are ignored; for example if the result of an Add operation is ignored then the Add need not be performed. Because the operands to Add may, however, cause side-effects, we cannot necessarily eliminate the operands to the operation. We can however ignore the result of them. By repeated application of this optimization the operands to an ignored operation may also be eliminated.

	Before	After
General form	*Load **x*** *Pop* *[…]* *unchecked UnaryOperation U* *Pop* *[…]* *Unchecked BinaryOperation B* *Pop* *[…]* *LdArrayIndex (ArrayType)* *Pop*	*Nop* *[…]* *Pop* *[…]* *Pop* *Pop* *[…]* *Pop* *Pop*
XIL Example	LdFloat 4.0f Pop LdNum 1 LdNum 4 Add Pop NoArgs CallStatic arr NoArgs CallStatic index LdArrayIndex Pop	Nop LdNum 1 LdNum 4 Pop Pop NoArgs CallStatic arr NoArgs CallStatic index Pop Pop
C# equivalent example	*ignore_result(4.0f)* *ignore_result(1 + 4)* *ignore_result(arr() [index()])*	// do nothing ignore_result(1) ignore_result(4) ignore_result(index()); ignore_result(arr());

New Object to Alloc and initialize

The NewObject instruction allocates and initializes an object all at the same time and is more efficient than allocating an object and initializing it separately. Despite this, constructors are usually simple methods and good candidates for inlining; so when compiling for speed on non-debug builds this optimization converts NewObject instructions which refer to inlinable constructors into an AllocObject followed by a static call that will later be inlined.

	Before	**After**
General form	*NewObject T::.ctor*	*AllocObject T* *CallInstance T::ctor*
XIL Example	LdNum 1 LdNum 2 Comma NewObject MyClass::ctor Pop	LdNum 1 LdNum 2 Comma AllocObject MyClass CallInstance MyClass::ctor Pop
C++ equivalent example	*new MyClass(1,2)*	(gcnew<MyClass>) -> ctor(1,2);

Retargetted local

When a (non volatile) local is assigned from the result of a simple load (e.g. a constant, parameter or local), the next load from this local variable can be changed to load from the constant, parameter or local that was just assigned into the variable slot.

This optimization is of huge significance in countering the inefficiencies and large number of locals introduced by inlining methods, and is further enhanced when combined with the FinalStore optimization.

	Before	**After**
General form	*Load (local,param,const)* ***x*** *StLocal* ***y*** … *LdLocal* ***y*** *Load (local,param,const)* ***x*** *StParam* ***y*** … *LdParam* ***y***	*Load (local,param,const)* ***x*** *StLocal* ***y*** … *Load (local,param,const)* ***x*** *Load (local,param,const)* ***x*** *StParam* ***y*** … *Load (local,param,const)* ***x***
XIL Example	LdNum 1 StLocal **x** NoArgs CallStatic foo Pop LdLocal **x** LdNull StParam **p** … LdParam **p**	LdNum 1 StLocal **x** NoArgs CallStatic foo Pop LdNum 1 LdNull StParam **p** … LdNull
C++ equivalent example	x = 1; foo(); (x) p = null; … f(p)	x = 1; foo(); (1) p = null; … f(null)

Store Load

Store load is an important optimization which avoids un-necessary loads to memory. There are two forms: Store Load and Load Store.

	Before	After
General form	*Load **x*** *Store **x***	*Nop*
	*Store **y*** *Load **y***	*Dup* *Store **y***
XIL Example	LdLocal ***Q*** StLocal ***Q***	Nop
	LdNum 1 StLocal ***M*** LdLocal ***M***	LdNum 1 Dup StLocal ***M***
ANSI C equivalent example	Q = Q;	// do nothing
	M = 1; (M)	*register* t = 1; M = t; (t)

Flatten Transient objects

Of the many criticisms that managed languages often face – garbage collection's long pauses are towards the top of the list. While garbage collection can be sped up in a number of ways, perhaps the easiest is to simply not allocate too many objects on the heap.

"Flatten Transient Objects" is a pass which detects whether a local variable can "escape" from the method scope, either by being thrown, being stored to another location (including other locals, since they can be loaded and escape from method scope too) or by being passed to another function (which could store the object to a static field, thus allowing it to escape the function scope).

If "Flatten Transient Objects" determines that an object is transient and tied to a method scope, the object is "flattened" into a series of locals that represent the fields of the object, and all field load and stores to that object are retargeted into loads and stores of these locals.

"Flatten Transient loads" not only reduces the number of objects allocated on the heap, but also allows the newly flattened object to take advantage of the shadow-stack; and since the object has been flattened, any fields of that object which are not used become "dead locals" which can be removed from the method signature, thus potentially reducing the total memory footprint of the program as a whole.

	Before	**After**
General form	*AllocObject* ***T*** *StLocal* ***x*** … *LdLocal* ***x*** *LdField* ***F*** … *LdLocal* ***x*** *StField* ***Q*** *where F : FType is a field of T* *and Q : QType is a field of T*	*[**F** ffield, **Q** qfield, …]* *Nop* … *LdLocal* ***ffield*** *StLocal* ***qfield***
XIL Example	AllocObject Complex StLocal *cplx* LdNum 1 LdLocal *cplx* StField Complex::Real LdLocal *cplx* LdField Complex::Im RetVal	[int cplx_real, int cplx_im] Nop LdNum 1 StField *cplx_real* LdLocal *cplx_im* RetVal
C++ equivalent example	Complex* cplx = new Complex(); … cplx->real = 1; return cplx->im;	Complex cplx = Complex(); … cplx.real = 1; return cplx.im;

Generating *XIL* basic blocks

Basic blocks are series of linear code which fulfill two axioms:

- The expression stack must be empty at the start (and end) of any basic-block.
- The basic block can only be executed as a unit – control cannot enter a basic-block except through the top of the basic block[7].

Let us consider by way of example the following *XIL* code:

```
public static Main(void) returns void
Int32 i = 5;
  Jmp $endfor

  Label $for
  NoArgs
  LdLocal 0
  Call System.Int32::ToString();
  Call System.Console::WriteLine(string!)
  Pop
  LdLocal 0
  LdNum 1
  Add
  Dup
  StLocal 0
  LdNum -1
  Lt
  JmpIfTrue $for

  Label $endfor
  Ret
```

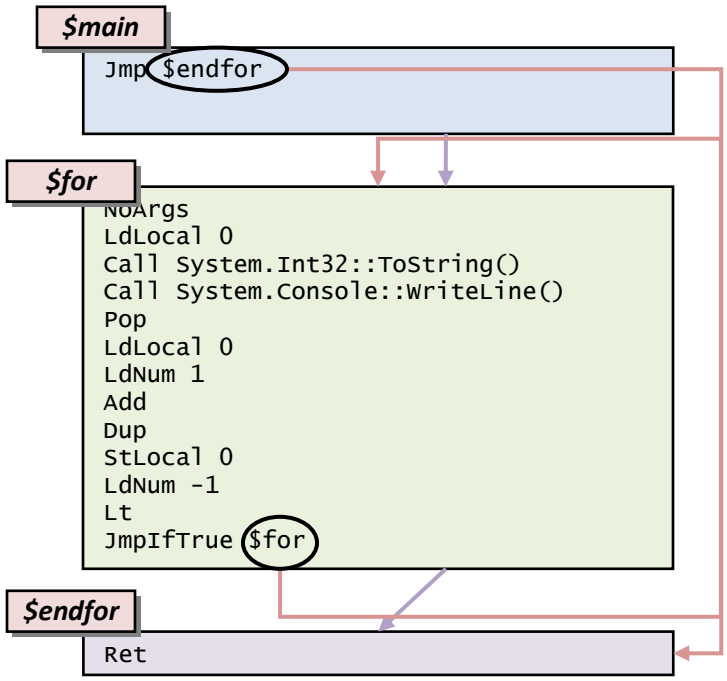


Figure 4: Basic block layout of Figure 3

[7] This is a relaxation of the definition given in the dragon book [1], which poses the additional constraint that basic-blocks should only naturally branch at the end of a basic-block.

Building the basic-block diagram from a linear description of the program in XIL is reasonably simple; assume at first (for simplicity) that we are converting a method with no special blocks into a control flow diagram such as that shown in Figure 4. In XIL, control can change at many points in the method, but can only be transferred to known points (labels) and can start at the beginning of a method. For this reason we take the method and break it into blocks which lie between labels.

It is usual at this stage to replace instruction branches (either conditional or unconditional) to point to basic-blocks rather than labels, since significant modification to the arrangement of basic blocks and the instructions within them means that a significant overhead of constantly retargeting these instructions would be cumbersome.

In Figure 4 we see the method from the previous page has been broken into three sections – the code from the start of the method to the first label, the code between the first label and the second label, and the code between the final label and the end of the method.

In addition to control flow being deviated via conditional and unconditional jumps, control flow occurs naturally between adjacent basic-blocks, for instance at the end of the second block if the conditional is not taken, control flows naturally to the following block. In the *Xinaos* compiler this is termed a "fall through" basic block transition. These are shown in purple in Figure 4 as opposed to "active basic block transitions" which are shown in red. The distinction is important since active block transitions must occur as the result of a branch, whereas fall-through transitions do not necessarily involve branches, and can therefore be taken without the overhead of actually performing a jump when the method is reassembled into linear code.

Terminal instructions

In the XIL instruction set there are five "terminal" instructions which unconditionally and permanently migrate control-flow out of a basic-block. These are Throw, ReThrow, Jmp, Ret and RetVal.

The various call instructions as well as instructions that cause returnable interrupts such as page faults do not need to be considered, since these return control flow to the basic-block on completion at the point immediately after the executing instruction, and for this reason the compiler may behave as though the instruction has not performed any branching behavior whatsoever.

There are two important consequences of these terminal instructions migrating code out of basic-blocks. The first is that all instructions in a basic-block following one of these terminal instructions are unreachable and are thus dead code. This is a natural consequence of the second basic-block axiom which states that control cannot enter part-way through a basic-block, and thus code following such terminal instructions is unreachable and is discarded. The second consequence is that if the basic-block contains a terminal instruction, then the end of the basic-block is unreachable and thus the fall-through of the block can never occur.

In Figure 4 we can see that the first basic-block finishes with an unconditional branch, and thus the first purple fall-through arrow is unreachable, and thus is eliminated. Additionally the final block contains a terminal instruction and thus the final block has no fall-through.

In the *Xinaos* compiler, if a basic block ends with an unconditional jump, the jump is discarded and the jump's target becomes the fall through target of the basic block. This is performed for simplicity of the control flow graph. Naively, such a translation from an active block transition (via a jump) to a fall through transition is a semantic violation; since while it is possible for two jumps to target the same basic block, it is not possible for two basic blocks to fall through to the same basic block without a jump instruction. In the next section we shall see that when we compile for speed this is not necessarily the case.

If this optimization is performed on Figure 4, the unconditional jump in the first basic block is disposed and the first block is connected to the final block via a purple arrow. The result of this can be seen in Figure 5.

Detecting unreachable basic-blocks

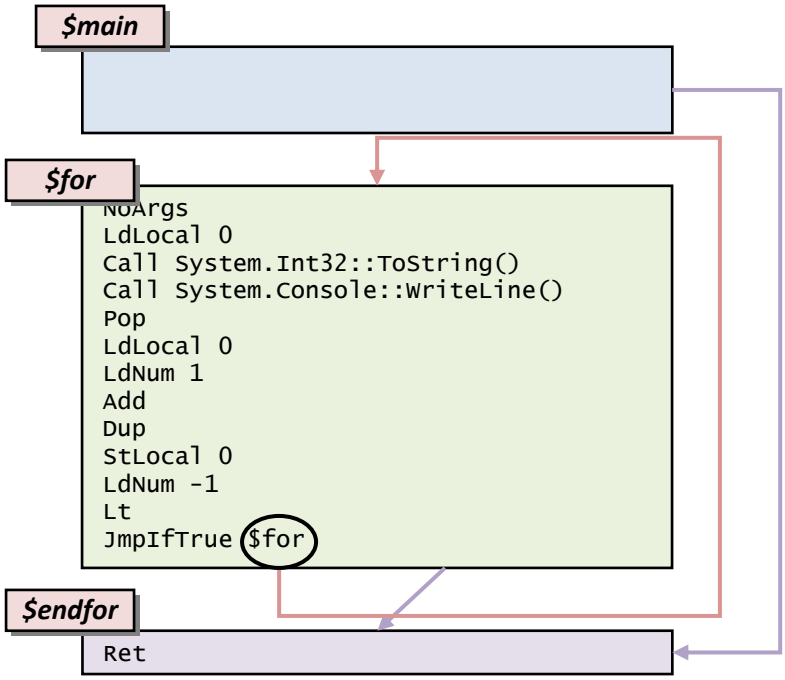


Figure 5: Branch optimized version of Figure 4

A basic-block is considered "dead" if it cannot be reached by any permutation of control flow originating at the start of the method, or a try or finally block. In control-flow graph terms, this means a block is dead if (and only if) it is not possible to get to the block by following arrows through the graph starting at an initial basic-block.

In Figure 5 the second basic block is dead, because it cannot be reached starting at the first method. The third block is not dead, since it can be reached by following the purple arrow connecting the first block to the third block.

Dead code-blocks can always be safely removed from the method, since the code that they contain is unreachable.

Merging code basic-blocks

Two blocks A and B can be safely merged together if

- The natural fall-through of A is B (or A finishes with an unconditional jump to B)
1. No reachable basic-block in the entire method jumps (conditionally or unconditionally) or falls-through to B

In Figure 5 we can merge the *$main* and *$endfor* basic blocks, since *$main* falls-through to *$endfor* and no other reachable instruction points or falls-through to *$endfor*.

The merged code block is simply the concatenation of the instructions which form A and the instructions which form B.

Proof of correctness of merging blocks

Let A and B be basic blocks; A not the entry point to the method, and Let A_s be all instructions pointing to A (either by fall-through or as an instruction) and B_s be the set of instructions pointing to B. Let A_D be the basic-block which is the fall-through of A and B_D be the basic-block which is the fall through of B. Let $a_i \in A$ be the instructions in A and $b_i \in B$ be the instructions in B and let $\#P$ be the size of some set P. We then have that if $\#A = \#B, \forall i : a_i = b_i$ and $A_D = B_D$ then all instructions A_S can be retargeted to B without affecting the operation of the method.

Proof is inductive.

Consider some point $s \in A_s$

Let ${E^A}_i$ be the state of the execution stack after the i^{th} instruction through A, and let ${Q^A}_i$ be the system state after the i^{th} instruction through A, which is the union of the state of the locals, object state space, parameters and static-variables in the program.

Consider first the effect of the expression stack at each point through A. An effect of the determinism of the XIL instruction set[8] is that $\forall a_i \in A, \forall b_i \in B, (a_i = b_i) \wedge \left(Q_i^A = Q_i^B\right) \wedge \left(E_i^B = E_i^A\right) \rightarrow E_{i+1}^B = E_{i+1}^A$.

Consider also the effect of the system state as the method progresses. An effect of the determinism of the XIL instruction set is that $\forall a_i \in A, \forall b_i \in B, (a_i = b_i) \wedge (E_i^B = E_i^A) \wedge \left(Q_i^A = Q_i^B\right) \rightarrow Q_{i+1}^A = Q_{i+1}^B$.

Since by assumption $\forall i, a_i = b_i$ and $Q_0^A = Q_0^B$ and $E_0^A = E_0^B = \emptyset$ (by axiom of basic-blocks) we inductively have that $\forall i, Q_i^A = Q_i^B$ and $\forall i\ E_i^A = E_i^B$, in particular $Q_{\#A}^A = Q_{\#B}^B$ i.e. that the system state having run through the set of instructions in A is the same as if those instructions had originated from B.

The only thing left is the ensure that at the end of the basic-block the next instruction to be executed is the same, and this is the case since by construction $A_D = B_D$, and thus the proof is completed because of the arbitrariness of s.

The major corollary to this is that A is dead-code. After the translation of A_S to point to B, $A_s^{'} = \emptyset$ by definition, and since A is by assumption not the entry-point to the method, it is unreachable by the entry-point to the method, and thus is dead-code.

[8] The only point where this is not strictly adhered to is when using unsafe code or direct-side-effects (PInvokes), both of which could potentially use (or abuse) features such as instruction pointer position, return address in memory, layout of the method on disk etc. Except perhaps for reflection and debug-interchange, this should never be relied upon, and even with these two examples the optimization remains valid so long as care is taken in interpreting the layout of the method, e.g. when “edit-and-continuing” code.

Semi-merging blocks of code

The main result of the previous proof is that it is possible to merge some blocks by duplication.

Suppose we have n blocks $A_1 \ldots A_n$, all of which point to some third block C which definitely terminates or falls through to some block C_D. Then if C does not use itself as a target for any conditional, unconditional or fall-through instruction, then C can be merged into all of the A_i blocks by copying the instructions of C to the end of the instructions in each A_i and retargeting the fall-through of the various A_i to point to C_D if appropriate.

Note that this is not affected by other instructions branching to C via conditional jumps, since C is not in general removed by this operation, although if no further instructions in the method point to C then C becomes dead-code and can be safely eliminated.

If $\#A_i = 1$ we have the special case of ordinary block merging.

It should be noticed that when there is only one physical manifestation of C on disk and in memory, only one basic-block can fall-through naturally to C without an unconditional jump. It is therefore the case that performing semi-merging is an actual speedup of the program by eliminating the unconditional jump that would otherwise occur.

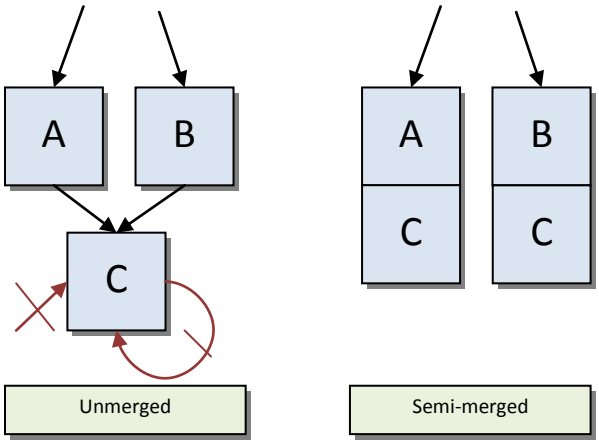


Figure 6: Semi-merging code

It should also be noted that merging code blocks yields a longer basic-block and thus potential for further simple optimizations on the basic-block. Moreover, semi-merging allows an even greater flexibility for such optimizations since many potential sites for semi-merging arise as a result of a conditional branch which rejoins at the end of the if statement from which it was derived.

```
int foo(int i){
  int j;
  if(i == 0){
    j = 4;
  } else {
    j = 0;
  }
  return i * j;
}
```

Figure 7: Example code before merging

```
int foo(int i){
  int j;
  if(i == 0){
    j = 4;
    return i * j;
  }else{
    j = 0;
    return i * 1;
  }
}
```

Figure 8: code after merging the final basic block (return i*j) into the two previous blocks

```
int foo(int i){
  if(i == 0){
    return 0;
  }else{
    return 0;
  }
}
```

Figure 9: After merging, the basic blocks can be re-optimised.

```
int foo(int i){
  return 0;
}
```

Figure 10: In this case, the two basic blocks are equivalent, since they contain the same code and fall through to the same basic block. In this case the conditional is redundant and the method can be further reduced.

Suppose for instance that we have a method which queries some variable i to see if has a value equal to zero. In the two basic blocks which derive from the if and else block respectively, the conditional itself provides a bound on i which can be used when further optimizing these basic blocks. Normally speaking any code which lies *after* the if statement cannot rely on information yielded in this way, since code which lies after the conditional block is taken regardless of the result of the conditional itself.

By performing a semi-merge operation on the block following the if statement into the taken and non-taken branches, we not only remove the unconditional branch which must otherwise occur at the end of the taken branch, we also propagate the bound placed on i by the conditional which can be used for optimizations even in code which lies after the if statement itself, for example if *i* is multiplied by some other value after the conditional block, then the multiplication can be optimized away (since $0i = 0$) in all cases where the code follows the conditional branch.

Special blocks

In XIL there are two forms of special blocks. The first is try..catch..finally blocks, and the second is the definitely-terminating for-loop instruction pair.

For try..catch..finally blocks we treat the blocks as a set of basic-blocks. The catch block is always reachable from the try block. Both the try and catch blocks fall through to the finally block, which falls through to the instruction which logically follows the try..catch..finally block.

In particular it should be noted that the catch block is reachable from the try block only if the exception type caught by the catch block is an exception which can be thrown by the code inside the try block; for example a catch block catching IOExceptions is not reachable from its try block if the try block contains no throw IOException instruction and no calls to methods which can throw an IOException[9].

If the catch block is unreachable and the finally block is empty, the try block can be considered to be an ordinary basic block.

Finally blocks are interesting – they can be executed in three different contexts. Firstly they can be executed when the try block finishes naturally. Secondly they can be executed when the catch block finishes naturally. Thirdly they can be executed if an error occurs in the try block which is *not* caught by the catch block – in which case the finally block is executed and then the exception propagates up the call stack until an appropriate catch statement is found to handle the exception.

For this reason we can treat the finally block as two different blocks. The first is an ordinary basic-block which both the catch and the try block fall naturally to. The second is a block which is called only when winding up the stack (the "unwinding-finally block") when the try block faults and the catch block does not catch the exception (or when the catch block catches the exception but subsequently faults). In this case we can treat the first block as we would treat any other basic-block, and can block-merge it with its neighbors. With the second form we have a further potential optimization. If there are no other exception blocks in the method, then when the unwinding-finally block finishes, all parameter sites and local variables are trashed as we return from the method. A natural consequence of this is that we can lose some additional stores and loads, for instance when clearing a local pointer to zero after releasing the native handle of some object.

Note that this is not generally the case that try blocks can be merged or semi-merged, since extending the try block upwards may accidentally catch exceptions thrown at the end of the previous block, and extending the block downwards may accidentally catch exceptions thrown by the finally block or code after the try block has ended.

---

[9] For most exceptions this is reasonably easy, however for the System.Exception class and all exceptions which can be thrown by various non-throw related instructions this is more complex, e.g. a DivideByZeroException could be thrown by any divide operation whose second parameter is not guaranteed against being zero, and thus a catch(Exception) or catch(DivideByZeroException) block is reachable from any try block containing such a divide instruction or a call to a method which could contain such a divide instruction.

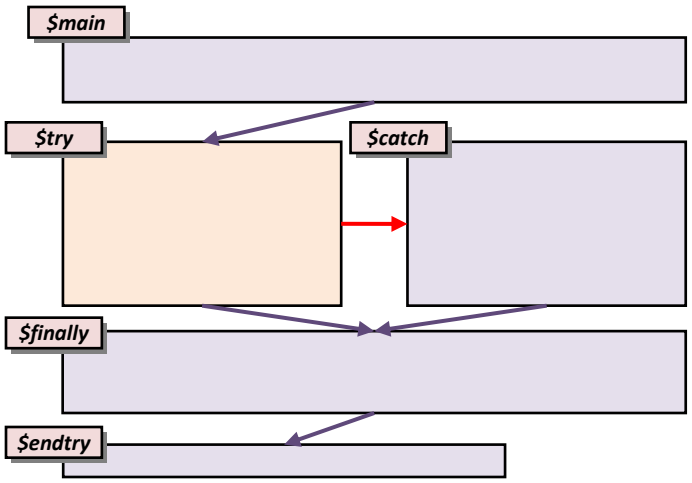


Figure 11: Layout of unoptimised exception blocks in a typical method

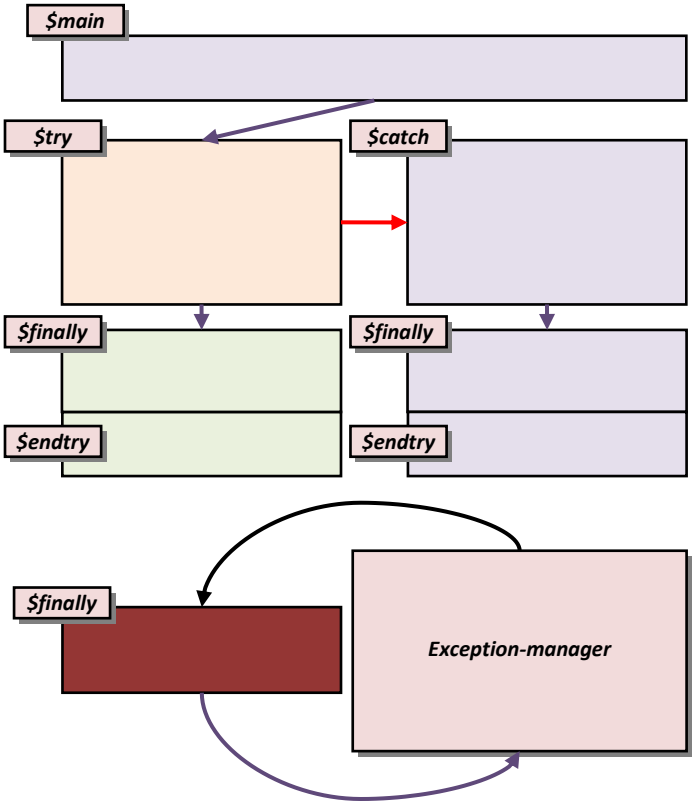


Figure 12: Speed-optimized alternative layout for Figure 11

Note that in Figure 12, the left-hand path is the most common control flow through the method (the right hand path by definition occurs only in exceptional circumstances). Although we cannot *extend* the try block into the finally and end-try blocks that follow it, we can still propagate *knowledge* from the end of the try block to the subsequent blocks – i.e. we know if the left-hand finally block is entered that the try completed successfully. This allows us to perform optimizations on the left hand, most used path.

Notice that there are three finally blocks in the method which correspond to the three ways a finally block can be reached. Firstly the finally block may be called because the try occurred successfully. This corresponds to the finally block on the left which may utilize the knowledge that the try occurred successfully and be merged with the code after the try..catch..finally block. Another finally block per catch block exists which corresponds to the finally block executed after the catch block occurred. This can utilize knowledge and be merged with the catch block and be merged with blocks after the try..catch..finally block, but cannot use knowledge from the try block itself (although knowledge at the start of the try..catch..finally block which is independent of the try block can still be used).

A further single finally block occurs in this diagram that corresponds to the finally blocks executed by the runtime as the exception manager winds up the call-stack when an exception is thrown from inside the try block that is not caught by any of the attached catch-blocks.

It is important to note that for the majority of programs which use the try..catch..finally construct, the most common course of action is to progress through the try block with no exceptions. Happily this is the most optimizable path through the method. The paths which correspond to finally blocks being run after catch blocks are both more optimizable and more commonly executed than finally blocks which are simply used to wind up the call stack, and so we can see that the splitting up of the finally blocks in this way allows for an increase of speed in the program as a whole.

Detecting "natural" for-loops

A "natural" for loop is a loop which

1. is taken at least once, and
2. whose total number of iterations is upper bounded before the loop enters.

It is not necessary that this upper bound be known at compile time, although it must be fixed before the loop is entered for the first time.

The first condition is often relaxed by putting the natural for loop inside a conditional block, and the name then derives from the fact that code analogous to the following can be converted easily to a natural for loop:

```
// a "natural" for loop which iterates forwards from A to B
for(int i=A;i<B;i++){
  code(i);
}

// a "natural" for loop which iterates backwards from Q to P
for(int j=Q;j>P;j--){
  code(j);
}
```

In a method made up of basic blocks $b_i \in B$, a set $(s_o \dots s_n \in S) \subset B$ are a "natural for loop" if

1. $\forall b_i \in B, \forall s_j \in S: b_i R s_j \rightarrow b_i \in S$
2. s_n conditionally loops to s_0 based on some variable v which is an integer and which is incremented unconditionally exactly once at the end of s_n
3. No other instruction stores a value to v during the execution of S.

i.e. no basic block outside of the natural for loop points into the natural loop and S conditionally loops on the loop variable v, which is incremented exactly once at the end of the loop.

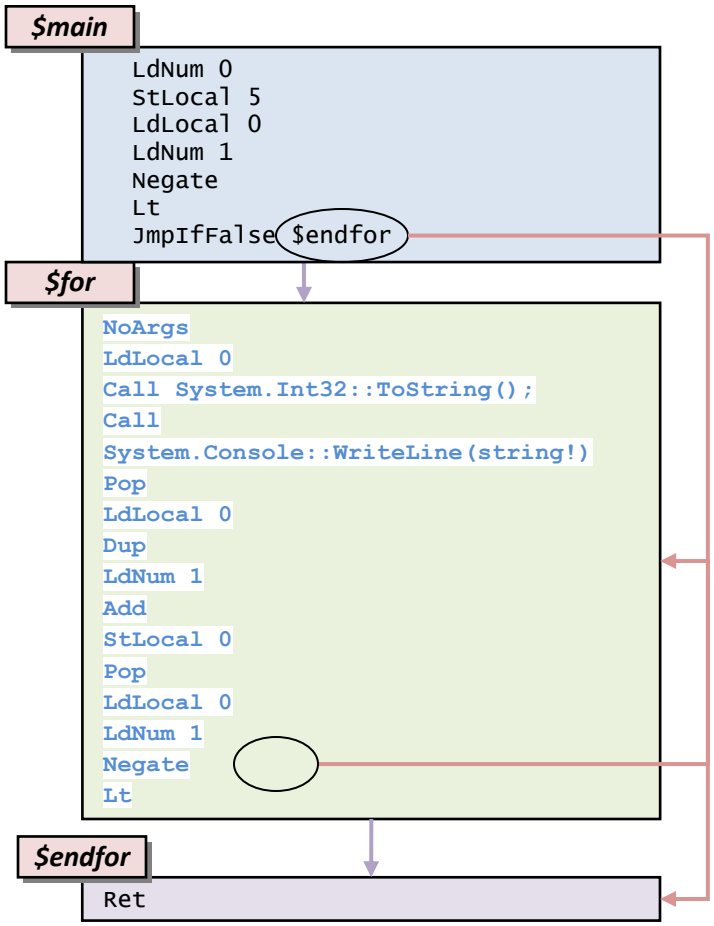
Detecting such a for loop is reasonably straightforward:

1. Detect candidates for S by looking for loops in the control flow graph, which are one or more basic blocks which form a cycle.
2. Filter such candidates by removing cycles which can be entered from a basic block not in that cycle.
3. Look at the end of the cycle and scan backwards to see if the condition is based on a condition from a variable (which must derive from int or unsigned int), and scan the final basic block in the cycle to see if that variable is stored exactly once from an increment of itself. Additionally check that such a variable is not stored at any other point in the cycle.

Once discovered, the natural for loop can be constructed. Consider the loop code:

```
public static void Main(){
  for(int i = 5; i <= (-1); i++){
    Console.WriteLine(i.ToString());
  }
}
```

which translates to the following XIL code



An inspection of the above code finds only one loop – that centering on the middle basic block. The central basic block jumps to itself conditional on local variable 0 (called i in Figure 2) which is incremented once during the basic block and not set at any other point.

For this reason the block becomes a candidate for for-loop conversion with local variable 0 as the loop variable. The central basic block is converted to the following code:

```
  LdLocal 0       ; where to loop from
  LdNum 1         ; }
  Negate          ; } where to loop to
  For (Local 0)
_forloop          ; we need a label for the Loop instruction to point to
  NoArgs
  LdLocal 0
```

```
  Call System.Int32::ToString();
  Call System.Console::WriteLine(string!)
  Pop
  Loop _forloop
```

The “for” instruction takes two expressions a and b from the expression stack which describe the iteration bounds. The first expression a is the initial value of the for loop variable and the second expression b is the final value. The for loop instruction takes a direct argument which is the variable to be treated as the loop variable (in this case local 0) as well as a flags byte which describes whether the for loop can be performed in parallel and whether the for loop should go forwards or backwards.

The loop instruction takes a single direct argument which is the label which points to just after the for loop. Depending on the flags byte of the “for” instruction, local 0 is either incremented or decremented, and if the value is equal to the final value of the “for”, then the loop terminates and control continues past the loop variable. If not, control moves to the instruction immediately after the for loop instruction.

Some important remarks about loops:

1. Multiple for loops can be embedded within each other, however each for loop must use a distinct local variable as its loop variable.
2. While any given for loop can have only one “for” instruction, it can have any number of “loop” instructions. Additional loop instructions may, for instance, correspond to “continue” statements in the loop.
3. While the for-loop is upper bounded in terms of iterations, it is possible for control to exit the for loop early by moving control flow beyond the last loop command of the loop, throwing an exception or returning a value. Note that doing so means that the loop cannot be performed in parallel.

Inlining methods

Abstraction is a great idea for programmers. It reduces the amount of code that a programmer needs to write and maintain, it helps the programmer by forcing him to think of problems in broad, generic terms and it aids code reuse by giving the programmer a body of general methods and types that can be reused as necessary by the programmer at a later stage.

The downside is that efficiency comes from context. Sure the call site might know that the first parameter of a method call to *foo* is positive, but does that stop *foo* from checking? Of course not. The programmer would have written his method to check because he wrote *foo* to be generic.

Inlining methods is the process by which these two extremes meet. Despite all of the bad connotations that "copy and paste code" has, cloning a method and replacing a call site with a carefully reconstructed method that avoids the call has the benefit that the code written by the programmer for general use can be optimized and tailored to the call site.

Consider the following call:

```
int foo(int a){
  int b;
  if(a == 2) {
    b = (a + 4);
    return b;
  } else {
    return a;
  }
}

int Main(){
  int i = foo(2);
  return i;
}
```

Figure 13: Before inlining

```
int Main(){
  // locals of the “Main” method
  int i;
  // parameters to “foo”
  int foo_param_a;
  // locals of “foo”
  int foo_local_b;
  // result of “foo”
  int foo_result;

  // int i = foo(2);
  foo_param_a = 2;
_foo:
    if(foo_param_a == 2){
      foo_local_b = foo_param_a + 4;
      foo_result = foo_local_b;
      goto _endfoo;
    }else{
      foo_result = foo_param_a;
      goto _endfoo;
    }
_endfoo:
    i = foo_result;
    return i;
}
```

Figure 14: After inlining

There are several important observations to make about the inlining process:

1. Inlining adds *lots* of local variables to a method

Because parameters to the target method, the “this” keyword and the target methods’ local variables need to be randomly accessible by the target method once inlined, a substantial number of new local variable are needed. Happily this number usually be dramatically reduced when the resulting inlined method is optimized, in particular using “retargeted locals” and “final store”

2. Inlining a method dramatically increases the complexity of control flow within the method

Inlining a method with B basic blocks into a method with C basic blocks yields an inlined method with $B + C + 1$ basic blocks. The additional one comes from the need to split the block that contained the original call. Typically many of these blocks can be merged or semi-merged to give longer basic blocks and fewer control statements.

3. Lots of new control flow and linear optimizations become available after inlining the method

Because methods are not created with prior knowledge or context of their parameters, inlining can often yield a substantial number of new optimizations which simply apply the context to the method to yield a more efficient solution for the method.

4. Objects which could not be flattened because they were sent as parameters to other methods may become candidates for flattening after the method has been inlined.

Inlining policy within the *Xinaos* compiler

Inlining methods does yield substantial gains to the overall speed of the program, however re-optimizing the methods once the inlining has taken place can and does lead to a significant time overhead when compiling. Furthermore, inlining methods can lead to multiple copies of a method being distributed around the program, leading to a large executable file.

In principle inlining can occur on any method that does not directly recurse to itself – a method which indirectly recurses to itself can inline called methods until the target for inlining calls itself directly, however such a policy can lead to huge and unwieldy methods, as well as taking up large amounts of compile-time.

For this reason, inlining is disabled entirely on debug builds and when “optimize for space” is enabled. When “optimize for speed” is enabled, inlining is performed on simple methods which contain only a few basic blocks.

Detecting parallelism over loops

Definition

The *exposed writability* of a variable site a is the union of all fields (which are strongly typed, named and attached to parent types) on a which are potentially accessed and written to.

The *exposed readability* of a variable site b is the union of all fields (which are strongly typed, named and attached to parent types) on b which are potentially accessed for reading.

The *collision* of two variables a and b is the union of fields (which are strongly typed, named and attached to parent types) which may hold a reference to the same value at that point in the code.

a and b are *disjoint* if their collision is the empty set.

Notes

Let a be type A and b be type B.

1. If f is a field of type T and T is a value type, then $\forall a, b: f \notin collision(a, b)$
2. if f is a field defined on type T, then $\neg extends(A, T) \rightarrow f \notin collision(a, b)$ and $\neg extends(B, T) \rightarrow f \notin collision(a, b)$
3. $collision(a, b) \ni \bigcup \begin{bmatrix} write(a) \wedge read(b) \\ write(a) \wedge write(b) \\ read(a) \wedge write(b) \end{bmatrix}$

Fundamental Theorem of parallelizability

A block of code $M(p_1, \ldots, p_n)$ is parallelizable iff:

1. M is not a method which causes side-effects, and does not call a method which could potentially cause side-effects
2. $staticVariableWrites(M) = \emptyset$
3. $\forall i, j: collision(p_i, p_j) \neq \emptyset \rightarrow i = j$

Proof

1. (Necessary condition)

The set of methods which cause side-effects is a subset of the methods which potentially cause side-effects, so without loss of generality, consider parallelizing a block M which causes side-effects. Given that paralleling a block of code potentially affects the order in which the code occurs, the result may be a reordering of side-effects in the block. Since a change of order of side-effects is potentially visible to an outside observer, the parallelism ceases to be a correct optimization.

2. (Necessary condition)

Consider parallelizing a method which writes to a static field *S*. Suppose it writes to the field twice with two distinct values. If the order of writes is reversed, the value of the static field is different when the block is reversed. Since parallelizing the block potentially changes the order of execution within the

block, and the static variable is visible to an outside observer, the parallelism is not a correct optimization.

3. (Necessary condition)

Suppose by way of contrast we have that the $collision(a, b) \neq \emptyset$. Then $\exists f: f \in collision(a, b)$. Suppose the method writes to field f which is reachable from a (see Notes 2). Then by Notes 2 we also have that a field f is readable from *b*, which means that the change to a may have affected b. Suppose the iteration order is now reversed. By reading *f* from *b* before writing *f* through *a*, we have potentially changed the value read from *b*, thus potentially changing the value of that iteration. If the value of that iteration is critical to the result of the block *M* then the result is an incorrect optimization.

1, 2, 3 (Sufficient conditions)

Suppose for contradiction we have a method *M* whose parallelization is visible to an outside observer and yet obeys the axioms of parallelization above.

Then for M to contact the outside world either we must:

a) Inform the outside world,
however this contradicts the axiom 1, that the method has no direct side-effects

b) Change the order of iteration to force a WAW, WAR or RAW hazard.
however this requires that one iteration writes to a value that is later read or written by an earlier iteration. This cannot be the case, since we cannot write to static variables (axiom 2) and no assignment to any field, local or parameter reachable from M can affect any subsequent read from any field, local or parameter from M, other than through that same variable site (axiom 3)

Parallelism and Arrays

For loops were invented to do operations over arrays, and so it is hardly surprising that loops often occur alongside arrays which take either parameters or results for the array. Array reads and writes can be treated just like any other variable site read and write – an array is safely written to if each iteration writes to a distinct part of the array. In practice this means trying to show at compile time that the array accesses in some way follow the bounds of the array.

Currently in the *Xinaos* compiler, the only way to ensure that an array is uniquely stored to/read from is if the array indexer is either the same as the loop variable, or some fixed offset from it.

Detecting properties of methods in the *Xinaos* compiler

There are a number of properties of methods which are automatically detected within the *Xinaos* compiler which are used by the parallelism detection engine. Mostly these properties describe what a method *can* do, rather than what a method *will* do, and as such are more accurate when a greater number of optimizations are applied (due to finding and deleting dead code which may contain code that corresponds to a potential effect of the method).

For these purposes, the "this" parameter can be considered to be a special parameter to the method which is always a non-null reference type.

Direct side-effects

- A PInvoke signature is a direct side-effect unless it is specifically marked otherwise.
- Any method which contains a reachable call to a method which may cause a direct side-effect may cause a direct side-effect
- Any non-resolved[10] reachable virtual call (either through an interface, delegate or abstract implementation) may cause a direct side-effect unless marked otherwise.

Static variable reads

- A method which includes a reachable LdSField *Type::f* instruction potentially reads the Type::f static variable.
- A method which reachably calls a method which potentially reads some static variable *v* also potentially reads that static variable.

Static variable writes

- A method which includes a reachable StSField *Type::f* instruction potentially writes to the Type::f static variable.
- A method which reachably calls a method which potentially writes to some static variable *v* also potentially writes to that static variable.

Variable exposure

- If LdLocal *i* is followed by a Throw, RetVal or is stored into a field or static field of an object, then local i is exposed method.
- If LdParam *i* is followed by a Throw, RetVal or is stored into a field or static field of an object, then param i is exposed by the method.
- If LdParam *x* or LdLocal *y* is immediately stored into *param Q* or local *R* then *x* or *y* is exposed if *Q* or *R* are exposed.

Readable fields

- If LdLocal i is followed by an LdField Type::f, then Type::f is a readable field of i.
- If LdParam p is followed by anLdField Type::f, then Type::f is a readable field of p.
- If LdLocal i is followed by an StLocal x, then all fields readable by x are readable by i.
- If LdLocal i is followed by an StParam x then all fields readable by x are readable by i.

---

[10] A resolved virtual call is a call which is technically virtual, but which can be resolved to a fixed type. For example, in new List<int>()).GetEnumerator(), the GetEnumerator() method is a virtual method call on type IEnumerable, however since we know that it is List<int> that is doing the call, we can resolve to an instance (non –virtual) call of List<int>::GetEnumerator() rather than the virtual IEnumerable<int>::GetEnumerator

- If LdParam p is followed by an StLocal x then all fields readable by x are readable by p.
- If LdParam p is followed by an StParam x then all fields readable by x are readable by p.
- if LdParam or LdLocal *p* is followed by a call instruction, then the all the fields readable by the parameter site that corresponds to *p* in the target method are readable by *p*.

Writable fields

- If LdLocal i is followed by an LdField Type::f, then Type::f is a writable field of i.
- If LdParam p is followed by anLdField Type::f, then Type::f is a writable field of p.
- If LdLocal i is followed by an StLocal x, then all fields writable by x are writable by i.
- If LdLocal i is followed by an StParam x then all fields writable by x are writable by i.
- If LdParam p is followed by an StLocal x then all fields writable by x are writable by p.
- If LdParam p is followed by an StParam x then all fields writable by x are writable by p.
- if LdParam or LdLocal *p* is followed by a call instruction, then the all the fields writable by the parameter site that corresponds to *p* in the target method are writable by *p*.

Constructors

- Objects passed to constructors (either via a NewObject instruction, or via an immediate call to a constructor after AllocObject) do not need to consider mutations to the "this" parameter during the course of the constructor, as AllocObject and NewObject guarantee to return a new object whose memory does not overlap with any existing object. This is an important optimization, as it allows objects which are effectively immutable to still be initialized without penalty in terms of parallelization.

Consider the code

```
float Dot(Vector a, Vector b){
  float x,y,z;
  p = a.x * b.x;
  q = a.y * b.y;
  r = a.z * b.z;
  return p + q + r;
}
```

The read sites of a and b are "Vector::x", "Vector::y" and "Vector::z", whereas the read site of p is just "Vector::x".

The collision of a and b is zero, since neither have any write sites. Notice that the const parameter modifier keyword in C++ corresponds to an assertion that the related parameter has no field write sites.

Because Dot performs no writes to static variables, has no collision in any of its parameters and performs no direct side-effects, the following loop is parallelizable, regardless of whether bigListOfVectors is disjoint.

```
float[] lotsOfDots(Vector[] bigListOfVectors){
  float[] results = new float[bigListOfVectors.Length - 1];

  for(int i=0;i<bigListOfVectors.Length - 1;i++){
    results[i] = Dot(bigListOfVectors[i], bigListOfVectors[i+1]);
  }
```

```
  return results;
}
```

Consider now the following code:

```
void DoLotsOfThings(ComplexNumber[] cmplx) {
  for(int i=0;i<cmplx.Length;i++){
    cmplx[i].Real += cmplx.Magnitude;
  }
}
```

This code has ComplexNumber::Real as a "write" site as well as a read site, and thus is not parallel, whereas the following code:

```
void DoLotsOfOtherThings(ComplexNumber[] cmplx) {
  for(int i=0;i<cmplx.Length;i++){
    cmplx[i].Real = cmplx[i].Imaginary;
  }
}
```

is parallel, because cmplx has "ComplexNumber::Real" as a write site and "ComplexNumber::Imaginary" as a write site, and since the two do not overlap, the loop is parallelizable.

Suppose we choose the distinctly non trivial example of a ray tracer:

```
Color[] Render(Scene s){
  Color[] color = new Color[width * height];
  for(int i = 0; i < width * height; i++){
    int x = i % width;
    int y = i / width;

    Ray ray = generateRay(x,y);
    color[i] = TraceRay(ray, Scene);
  }
  return color;
}
```

generateRay has no collisions by virtue of the fact that x and y are value types. Supposing generateRay does not store to static variables and does not cause side-effects, then generateRay is a parallelizable method call.

Suppose further that TraceRay does not mutate Scene, cause direct side-effects or write to static variables. If this is the case then the loop is parallelizable, even if TraceRay modifies ray (since "ray" is local to the loop scope). The assignment into color is parallelizable only because the index (in this case i) is used as the index into the array. If compiler could not prove that the assignment to color was parallelizable (for instance if the index was constant, or in some way obfuscated) then the loop would not be parallelizable.

Detecting and implementing non-loop based parallelism

Consider some code and its resulting call-graph to recursively compute the n^{th} element in the Fibonacci sequence:

```
static int fib(int n) {
  if(n == 1 || n == 2) return 1;
  return fib(n - 1) + fib(n - 2);
}
```

Example 1

It can clearly be seen that the function is parallelizable, since each call creates two successive calls to a function, each of which may be expensive and each of which is independent of the other – i.e. each operand to the addition can be performed concurrently.

In order to detect such parallelism, we first reduce the function to its *XIL* equivilent (annotated in Example 2)

```
      static int Fib(int n):
 1                            ; if n == 1 then return 1
 2    LdNum   1
 3    LdParam 0
 4    Neq     .l1
 5    LdNum   1
 6    RetVal
 7                            ; if n == 2 then return 1
 8  .l1:
 9    LdNum   2
10    LdParam 0
11    Neq     .l2
12    LdNum   1
13    RetVal
14  .l2:
15                            ; first operand - fib ( n - 1 )
16    LdParam 0
17    LdNum   1
18    Sub
19    Call    Fib
20                            ; second operand - fib ( n - 2 )
21    LdParam n
22    LdNum   2
23    Sub
24    Call    Fib
25                            ; adds the two operands
26    Add
27    RetVal
```

Example 2

The mechanism for detecting non-loop based parallelism opportunities works thus:

1. Decompose the function into basic blocks.
2. For each basic block
    a. Traverse the basic block, travelling upwards from the end of the block towards the start.
    b. If an instruction is detected with at least two expression-operands and the expression operands both have the following properties:
        i. The instructions are pair-wise memory disjoint
        ii. The instructions are side-effect free

    Then the instruction is a parallel-operand opportunity.

Looking again at Example 2 we can see that there are basic linear blocks – the first is lines 2-6 inclusive, the second is 8-14 and the third is 16-27. In the first basic block we transverse the block backwards. Instruction 6 is a **RetVal** instruction which has only one expression argument. Instruction 5 is a load and has no expression arguments. The next instruction, instruction 4 is an **neq** instruction which takes two expression arguments. Furthermore, the two operands (at instruction 2 and instruction 3 respectively) are independent of each other and cause no side-effects, and thus instruction 4 of the first basic-block represents a parallel-operand opportunity.

The second linear block is similar to the first, and has a further parallel-operand opportunity on instruction 11.

For the third block we once again start at the end of the block going backwards. The first instruction with two expression-operands is **Add** on line 26, and that this has two expression operands – the block 16-19 inclusive and the block 21-24 inclusive. Since the **Call** instruction on line 19 and line 24 refers to a function which causes no direct side-effects and takes no parameters by reference, the function is pure. There are also no stores during either of the two operands, and thus the operands are disjoint. It therefore follows that instruction 26 is another parallel-operand opportunity.

By continuing through this process we find that there are also an additional two parallel operand opportunities on lines 18 and 23 respectively.

Although we could in principle expand every parallel-operand opportunity into an out-of-order Future<T> construct (shown in Example 3), doing so often leads to slower code when the overheads of the out-of-order Future<T> construct outweighs the potential savings of the code being offloaded via the Future<T> construct[11]. Although not currently implemented in the *Xinaos* compiler, it is also possible for the notion of parallel-operand opportunities to be used in architecture specific optimizations to code explicitly for pre-emptive and even speculative loads at compile-time [1].

When it comes to implementing parallel-operand opportunities for Future<T> offloading, there are few heuristics that are a good rule-of-thumb and which are used by the *Xinaos* compiler:

1. If the operand is generated neither from a call nor from a loop (after inlining) then it is unlikely that the parallelism is worthwhile.
2. If the operand is binary and commutative, then at most one operand should be offloaded via the Future<T> construct. If the operand chosen is the first operand, the order of the operands can be reversed.
3. If the operand is not commutative, then at most one operand should be offloaded via the Future<T> construct. If the operand chosen is the first operand, the Future<T> should be constructed first, the second operand should be stored into a temporary local variable, and then the operation should be

[11] It should be noted that this is still real parallelism that has been detected. Despite our not being able to use such micro- parallelism, the parallelism on this scale is often automatically detected and exploited by out-of-order processors which obtain the parallelism at the instruction-level.

performed using the temporary local variable and the value of the Future<T>. In practice the temporary local variable will be stored on a register, and thus no cost will be associated with the reversal.

```
static int fib(int n) {
  Future<int> _o1, _o2, _o3, _o4, _o5;
  _o1 = new Future<int>( () => 1);
  _o2 = new Future<int>( () => 2);
  _o3 = new Future<int>( () => 1);
  _o4 = new Future<int>( () => 2);
  _o5 = new Future<int>( () => fib(n - _o4.Value));
  if(n == _o1.Value || n == _o2.Value) return 1;
  return fib(n - _o3.Value) + _o5.Value;
}
```

Example 3: Exploiting all of the parallelism operand opportunities in a method is rarely a good idea

Looking bottom-up for the third optimization block we find **Add** is the first multi-operand instruction. If the operand is distinct from the first operand (i.e. the first operand does not store into an operand used by the second and the first operand is side-effect free) then the second operand (highlighted above in bold) can be executed in parallel and out of order with the previous operand.

In order to migrate the code into a parallelizable segment, the code segment needs to wrapped into a delegate closure. The code itself can then be replaced with a new-object instantiation of Future<T>, passing the new method as a delegate signature to the constructor of Future<T>, followed by a call to Future<T>::.getValue(). The new-object instance is then free to be moved upwards through the method code by the optimizer.

In the above code, we can optimize thus:

```
Fib(int n) returns int:
  .local Future<int> _o2;

  LdNum       1                   ;
  LdParam     n                   ;
  Eq
  JmpIfFalse   label1             ; If n == 1, return 1
  LdNum       1                   ;
  RetVal                          ;

; if n == 2 then return 1
label1:
  LdNum    2              ;
  LdParam  n              ;
  Eq
  JmpIfFalse label2       ; If n == 2, return 1
  LdNum    1              ;
  RetVal                  ;
label2:
  ldParam  n              ;  } - Form the closure using a parameter
  NewDelegate FibHelper   ;  }     new delegate() { Fib( n - 1) }
  NewObject Future<T>::.ctor(delegate)
                          ; new Future<int> passing the closure as an argument
  StLocal  _o2            ; store in a temp (_o2)
  LdParam  n
  LdNum    1
  Sub
```

```
  Call     Fib
  LdLocal  _o2                    ; _o2.Value
  Call Future<T>::.get_Value()    ;
  Add
  RetVal

Fibhelper(int n) returns int:
  LdParam  n
  LdNum    2
  Sub
  Call     Fib
  RetVal
```

Example 4

which is analogous to the code:

```
int Fib(int n){
  if(n == 1 || n == 2) return 1;
  Future<int> _o2 = new Future<int>(new delegate(){ return pfib(n - 2) });
  return pfib(n - 1) + _o2.Value;
}
```

Example 5

Notice that since Future<int> silences background exceptions until the .Value is called, Future<T> can also be used speculatively (analogous to speculative prefetching and speculative execution at the processor level [1]), although care needs to be taken to explicitly cancel the Future on paths where the Future is not needed to avoid infinite regression on the background threads – for instance in the previous example the Future<T> is not speculative (since execution paths in which _o2 is initialized also demand _o2's value), however if we were to move Future<T> before the *if* construct, we would have to take care to cancel the Future<T> before returning 1 in the case where the *if* branch is taken, thus:

```
int Fib(int n){
  Future<int> _o2 = new Future<int>(new delegate(){ return pfib(n - 2) });
  if(n == 1 || n == 2) {
    _o2.Cancel();
    return 1;
  }
  return pfib(n - 1) + _o2.Value;
}
```

Example 6

Note that failing to cancel _o2 could potentially lead to a stack-overflow exception on a background thread, which could cause the application to fail[12]. More generally, the form shown in Example 6, which

[12] **Rationale**: Call-frames are not cleaned up until the try..catch..finally block exits since the details of the open stack frames are kept in memory by the garbage collector to more proactively search for dead code when the try..catch block terminates. Consequently any local variable assignment or function call inside the body of the catch, or any subsumed finally block would cause a further stack overflow exception to be thrown (non-deterministically). For consistency in implementation, StackOverflowExceptions cannot be caught by managed code.

while looking nicer in code terms, actually poses a serious runtime slowdown by preventing the potential inlining of the function, as well as making exception rollback more expensive.

The *Xinaos* compiler can utilize parallel-operand opportunities to parallelize code, but it does not compute operands speculatively.

```
int Fib(int n){
  using(Future<int> _o2 = new Future<int>(new delegate(){ return pfib(n - 2) })){
    if(n == 1 || n == 2) {
      return 1;
    }
    return pfib(n - 1) + _o2.Value;
  }
}
```

Example 7

In practice this optimization should only be done when both operands are expensive to compute; -(n-1)*(n-2) has parallelizable parameters but the overhead of performing them out-of-order dramatically outweighs the

The *Xinaos* Garbage Collector

The Xinaos garbage collector (GC) is responsible for the memory management of all managed objects in the Xinaos runtime environment, and is a substantial program in its own right. The Xinaos garbage collector is capable of interrupting live program flow to look for dead objects, allocating new objects, and moving and compacting the heap.

The Xinaos garbage collector is a live garbage collector, in that it alters pointers on the stack during program execution, and consequently has some sections written entirely in x86 assembly language for going through and altering the live stack.

The garbage collector is partitioned into several parts:

- The GC class can be called into by both managed and unmanaged code, and is responsible for the public-facing interface to the garbage collector. It manages policy, allocations and finalization routines.
- The GCHeap class is used by the GC as a way of managing contiguous large blocks of memory which is held wholesale by the GC and given away in parts as managed allocations require.
- The LiveObjects class is used by the GC to describe an object which exists on the heap. It contains the UID for the object, the object type, heap pointer, creation time stamp, array length (if applicable), size on heap and whether or not the object should be finalized.
- The DeadSpace class is used by the GC to manage and combine areas on the heap (other than at the end of the heap) where allocations can take place.
- The GCPtr class holds a reference to the heap and a type-id to be used in managed code. The GCPtr class is only used in unmanaged code inside the runtime interacting with managed objects, and although managed code sees all managed objects as strongly typed, the GCPtr class contains no compile-time available strong-typing to the managed type of the object referred to; that is to say GCPtr is semantically equivalent to the System.Object type. Native methods typically receive their arguments as GCPtrs when called from managed code.
- The gcptr<T> class extends the GCPtr class so that it is compile-time strongly typed to the underlying managed type of the object on the heap; that is to say gcptr<T> is semantically equivalent to the type T, where T is a type managed by the Xinaos runtime.
- The gcptr_array<T> and gcptr_array_val <T> classes use C++ operator overloading to allow clearer syntax statements in unmanaged code (such as array[i] = T)
- The lock_t class is used to allow entering and leaving of critical regions inside the garbage collector.

Testing the garbage collector

Testing the garbage collector is a significantly non-trivial task. Although testing was performed both with regression testing and a live testing suite, my confidence in the garbage collector comes having proved its correctness – the code for the garbage collector is liberally strewn with assert_weak macro statements and comments which both formally and informally prove the correctness of each step in the garbage collector.

The garbage collector was built in parts, each one of which was tested independent of each other with a custom set of programs designed for testing each feature of the garbage collector, and each part of the

garbage collector was independently proved correct – giving confidence that the final collector is also correct.

When assert_weak statements failed during debugging – showing an error in a step of the garbage collector – debugging typically consisted of looking at memory dumps from the stack or heap (or both) and carefully assessing where pointers and their associated data were.

Having completed the garbage collector, I wrote a small program which uses PInvokes and a number of managed classes to create and maintain a window and draw a box. The window reacts to the cursor, creating numerous objects as the cursor moves over the window. These objects eventually fill up the heap, and the garbage collector either compacts or migrates the heap appropriately. Although the program is a toy program and serves no real purpose, it was built subsequent to the garbage collector being built, and required no change to the garbage collector – that is to say the garbage collector is not a "build as you need it" framework, but rather a complete and provably correct program in its own right.

Correctness in the Garbage Collector

The garbage collector is a huge program which performs highly complex operations at difficult to determine times in ways that debuggers are not able to debug. A consequence of this is that the garbage collector has a number of files and macros dedicated to asserting correctness in the garbage collector. The most important are:

BREAKPOINT

This macro calls the x86 interrupt 0x03 which triggers a breakpoint in the attached debugger.

assert(x)

This macro forces a breakpoint if the argument does not evaluate to logical TRUE, followed by a halt in the program. This macro is used whenever the statement x must be true for a meaningful continuation of program execution – for instance the argument to GC::Create must be non-null.

assert_weak(x)

This macro behaves as assert(x) when in debug mode, but is disabled on release builds. Unlike assert(x) this macro is used to document behavior which will always be true – and is used to both guarantee correctness in regression testing and as a self-documenting feature in the code as a whole, for instance asserting that the pointer yielded from GC::Create exists entirely on the current live heap.

In addition to these "correctness" macros there are a number of other useful methods – in particular GC::_unaccountedForSpace() computes the difference between the total number of bytes in live objects on the heap and the total amount of heap space currently in use. This method is of particular use in checking that the garbage collector has not "lost" any data, and liberal use of it in assert_weak statements helped find almost impossible-to-detect errors in the garbage collector.

Garbage collector configuration settings

CONFIG.hpp contains a number of macros which define the garbage collector behavior. This includes constan-ts which can be “tweaked” for efficiency as well as definitions which dictate if certain additional tasks should be performed to aid debugging. These options are:

PAGE_SIZE = 4096

Defines the basic allocation size (in bytes) used by the GC-heap. The size must be a power of 2. Higher numbers results in fewer initial collections, but can lead to program taking more memory than it needs.

DEBUG_CLEAR_HEAPS (set for debugging heap moves)

When defined, the GC will reset the memory in the heap once it has migrated to a new heap. This is expensive and typically unnecessary, since newly created managed objects on the heap have their memory regions automatically zeroed on creation, but it can be set to help debug GC issues by clearing data at pointers that point to the old heap (this does not happen normally in the GC, but can be checked when debugging heap migrations)

DEBUG_CLEAR_HEAPS_VALUE = 0xcdcdcdcd

The value to clear heaps with. It doesn't really matter what it is but it makes sense to choose something that's obvious when it happens, but is unlikely to happen naturally (i.e 0 is a bad choice). This option is ignored if DEBUG_CLEAR_HEAPS is not set.

INITIAL_HEAP_SIZE = (1*PAGE_SIZE)

When the GC first starts, this is the number of bytes given to its initial heaps. This should be a multiple of PAGE_SIZE.

HEAP_OBJECT_ALIGN = 16

When an object is allocated on the heap, its pointer is aligned to this value. This value must be a power of 2. HEAP_OBJECT_ALIGN = 1 has minimum heap waste, HEAP_OBJECT_ALIGN = sizeof(void*) has significant memory access speedup, by ensuring that pointers are dword aligned. Larger values of this reduce memory fragmentation on the heap by giving more opportunities for heaps to reuse dead-space between objects on the heap (since dead-spaces are always a multiple of HEAP_OBJECT_ALIGN bytes long)

You should not use less than sizeof(void*) for this value.

MINIMUM_SPACE_FACTOR = 2

When the GC is performing a forced expansion (and heap migration), the size of the new heap is the product of this factor and the total number of bytes in live objects on the heap, rounded up to the nearest page-size. For obvious reasons this must be more than 1 (and may be floating point)

MEMORY_CLEANUP_ON_EXIT (set for debugging memory leaks)

When defined, the GC will be sure to explicitly free memory resources such as the heaps, vectors and sets back to the operating system when exiting the program.

This should be turned on if using tools to check for memory leaks in the program. Otherwise it should be disabled, since memory is automatically reclaimed by the operating system when the program

terminates, and it does so by just reclaiming the memory pages in use by the program (which it has to do anyway). Unsetting this value therefore speeds up program exit.

FINAL_RESOURCE_CHECK_ON_EXIT (set on debugging new native handle wrappers)

When defined, the runtime will perform additional checks to ensure that native pointers have been released. This helps to catch resource leaks, and should be turned ON when defining new managed wrappers, and turned OFF when running code normally. For normal programs, this task is handled by Finalizers.

GCPTR_CLEANUP_ON_SCOPE_CHANGE (set for ensuring fine-grained GC::Collects)

When defined, GCPtr, gcptr<X> and gcarray<X> will implement an inline destructor that ensures that their space on the stack is zeroed when they go out of scope, so that a GC::Collect() will recognize them as disposed inside the same function frame. When unset we save a memory operation whenever a GCPtr, gcptr<X> or gcarray<X> goes out of scope, but we potentially lose some efficiency on the heap since we will not recognize the object as being dead until the function returns, and will end up with a less fine-grained GC::Collect() as a result. This is only important for native-methods interacting with managed code.

ALLOW_GC_HEAP_END_REWIND (unset for debugging natural GC heap moves)

When defined, GC::Collect will combine deadspace at the end of a GCHeap by simply reducing the curptr of the heap (zeroing if DEBUG_CLEAR_HEAPS is set).

This has the effect that all dead regions at the end of the heap are reclaimed for free. This is cheap and should always be set, but can be disabled when debugging the GC in order to force more GC heap migrations.

PROCESSOR_BITS = 32

Either 32 or 64, depending on the processor. This is used by the PINVOKE signatures, and should match the operating system, rather than the processor.

DUMP_ON_ABNORMAL_EXIT (set for debugging abnormal terminations)

If the program calls terminate() or abort(), the GC will not attempt to run finalizers (since terminate() and abort() must come from native code or the operating system and thus the GC may be in inconsistent state). If set, the program will attempt to perform a heap and object dump for debugging purposes during an abnormal exit.

PAUSE_ON_EXIT (set for debugging GC::Exit)

If the program exits naturally, either by reaching the end of gcmain() or by calling exit(), then GC::Exit() will be called, which will clear up all live objects and run finalizers and then exit (regardless of whether PAUSE_ON_EXIT is set).

If PAUSE_ON_EXIT is defined, the console will additionally wait for the user to press ENTER before exiting – allowing the user to check messages printed by GC::Exit(), including finalization warnings. This should be defined when debugging GC::Exit(), but disabled for all other programs.

PERFORM_WEAK_ASSERT (set for performing regression tests)

The program uses two forms of assert – assert(_Expression) and assert_weak(_Expression).

assert(X) checks that X is true before continuing – that is if X is not true, the program is broken and execution should halt.

assert_weak(x) in contrast says that X is the case, and a failure of assert_weak is a failure of the garbage collector rather than of the program.

If PERFORM_WEAK_ASSERT is defined, assert_weak is checked during runtime to help catch and diagnose errors – in particular regression errors. It should be undefined on all other builds to avoid superfluous checking.

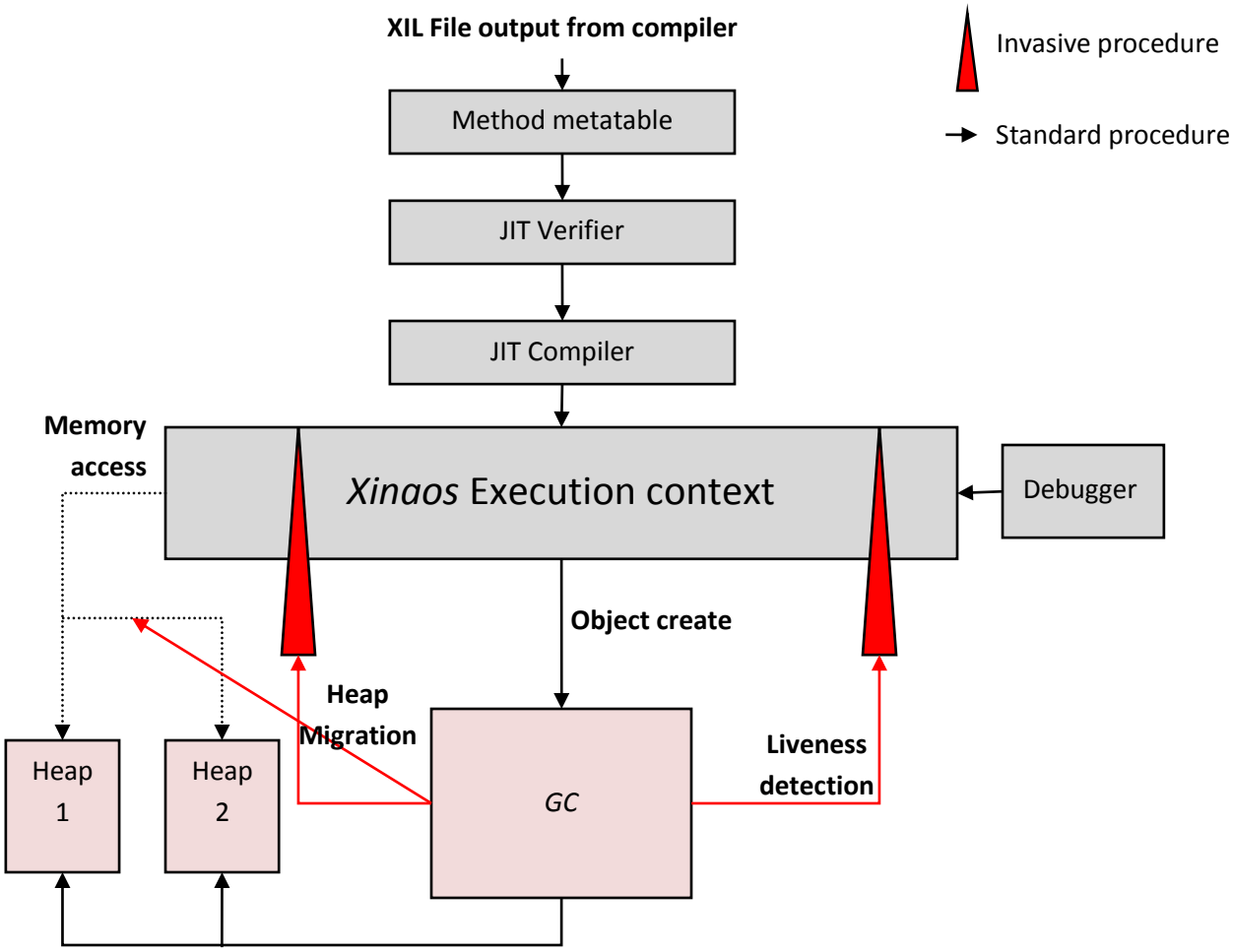
RUNTIME_WARN_ON_FINALIZER_CALL (set for testing libraries)

Finalizers are a last line of defense against dropping native handles, and ensure that native handles are cleared up on native-wrappers which fall out of scope. This said, finalizers are called non-deterministically and force the object to be kept in the working set (to ensure that their native pointer is not lost) for longer than would be ideal. This is doubly bad when you consider that this means objects due for finalization must be migrated to a new heap during a heap migration (see Finalization chapter).

While it is therefore good and proper that finalizers exist (it is better to run a finalizer than to drop a native pointer), it is generally better to deterministically dispose the object via the Dispose pattern – the object calls dispose just before it leaves the program scope for the last time, which then releases the native resources and calls GC::SuppressFinalize(this).

If RUNTIME_WARN_ON_FINALIZER_CALL is set and an object goes out of scope and is due to be finalized, the GC will print diagnostic information to the screen (and will additionally pause during GC::Exit if any object is still live with a finalizer that has yet to be called).

Figure 15: Layout of the Garbage collector with respect to the *Xinaos* runtime



The GCHeap class

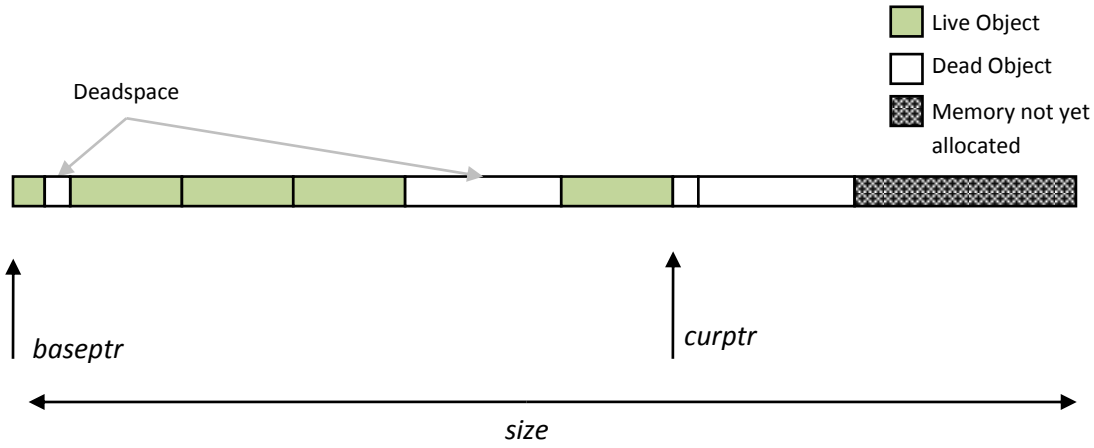


Figure 16: Heap layout in the garbage collector

The Garbage collector is responsible for the allocation of all memory resources used by managed code inside the *Xinaos* runtime – however it is more efficient to preallocate a large block of memory and to partition it into smaller objects than to make a separate memory allocation for each new object. In the *Xinaos* garbage collector, each runtime instance has two heaps which are contiguous large allocations of memory used for storing all of the objects that are needed by the runtime. Each heap is managed by the GCHeap class.

The heap itself is a relatively simple class – a pointer *baseptr* defines the base of the heap, which ultimately is derived from a call to the operating system's memory manager, *curptr* – a pointer to the end of the heap which is the smallest pointer such that all currently live objects exist between *baseptr* and *curptr* and *length*, which ensures that we do not overrun the length of the heap.

The GCHeap class does not deal with heap compaction or heap migrations – this is handled by the GC class, however the GCHeap class is responsible for the fundamental memory allocation and de-allocation at the operating system level, and thus is able to expand the heap, albeit losing all data held on the heap in doing so. The loss of data is not necessary, although insisting on keeping data would involve a complete copy of heap memory which would be both unnecessary in theory, since the heap is typically expanded after all objects have moved to the swap heap, and expensive in practice.

The GCHeap class has one major function which is to allocate a region of memory on the heap. It takes a single parameter which is the size of the allocation in bytes (which must be less than the size remaining on the heap minus (**HEAP_OBJECT_ALIGN** – 1)). The GCHeap rounds the requested size to the nearest HEAP_OBJECT_ALIGN block, and increments the *curptr* of the heap by this value. The previous value of the *curptr* then lies within the heap, and is returned.

In debug builds the GCHeap additionally checks that the heap is non-null, checks the parameter for range, checks the resulting pointer for containment in the heap and ensures that the object does not overlap any other object that has been previously allocated on the heap.

GC::_allocate

GC::_allocate is the central shared memory allocator used by the *Xinaos* garbage collector. It provides common functionality to both the forms of GC::Create and effectively works as the GC's form of the standard malloc call. It takes a parameter expecting *size* number of bytes to allocate and returns a pointer into the heap where at least *size* bytes have been allocated.

GC::_allocate begins by using heuristics to determine if a collect should be run. If it chooses to perform a collect, GC::_allocate first attempts to flush the pending finalizers to increase the number of dead objects before calling GC::Collect.

GC::_allocate first attempts to allocate memory from the deadspace directory – that is memory from dead objects that lies between live objects on the heap. Allocations from deadspace are managed by the GC::_allocFromDeadSpace method.

If GC::_allocate cannot allocate from memory held between live objects on the heap, it attempts to allocate data from the end of the live heap. This task requires first checking that there is sufficient space on the heap, followed by a request to the current live heap via the GCHeap::Alloc method.

In the event that GC::_allocate cannot obtain memory either from space held between objects or from the end of the current live heap, the garbage collector forces a complete heap migrate via the GC::_forceMigrate method. Having done so, GC::_allocate clears the deadspace directory (since a heap migration also forces a heap compaction, so no deadspace exists after the migration) and then allocates the object from the end of the heap.

Deadspace management

The garbage collector has two main sources of memory – the first ("clean memory") comes from the end of the heap. The second is deadspace which is recycled memory from dead objects which lies *between* objects on the heap. The maintenance and auditing of deadspace regions is necessary to reduce memory fragmentation on the heap as well as to reduce the number of heap migrations that are necessary.

Deadspace management is the term whereby dead regions between live objects are audited and managed, and contiguous regions of deadspace are combined to form fewer large regions of deadspace.

Deadspace is discovered by the GC::Collect method after it discovers an object which is no longer reachable. The GC::Collect method calls the GC::_markAsDead method, passing the size and pointer of the object when the object is no longer live. Because the GC::Collect function typically discovers numerous dead objects at once, the deadspace is not immediately merged until the GC::Collect calls the GC::_mergeDeadSpace method once all dead objects have been marked.

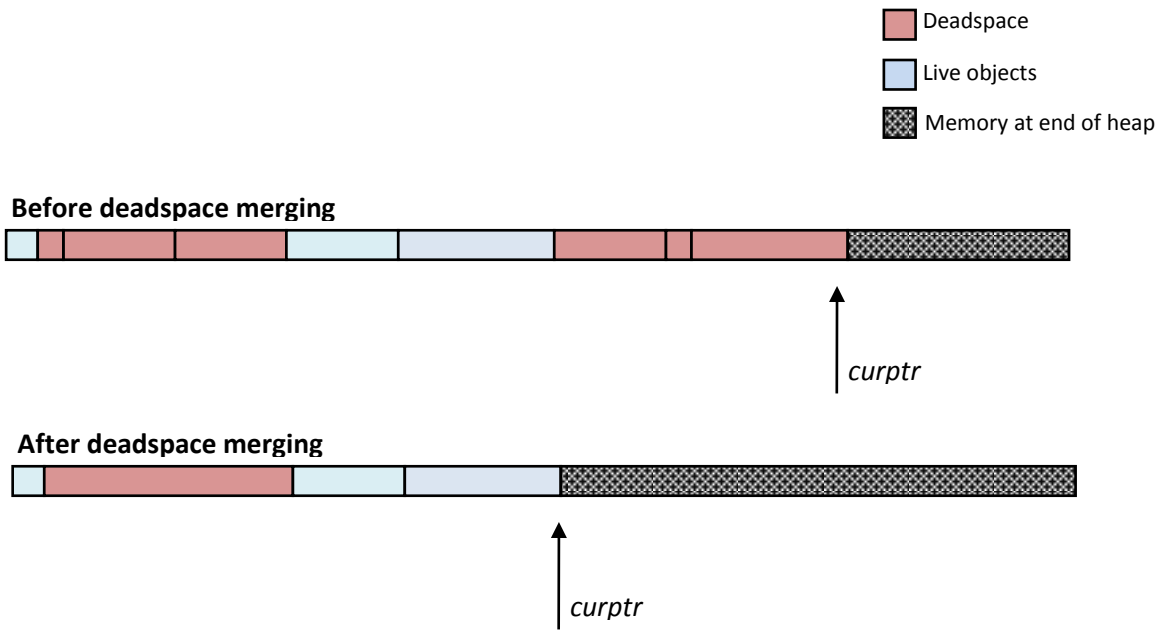


Figure 17: The effect of merging deadspace on the heap

Marking deadspace is a reasonably simple matter. In debug builds the pointer is checked for consistency and optionally cleared (if DEBUG_CLEAR_HEAPS is set), however the general working is simply to add the object into a list of heap gaps to be merged when GC::_mergeDeadSpace is called.

Merging the deadspace is a more significant endeavor, and relies on a number of tricks to speed it up. The first such trick is to notice that many objects which die lie adjacently on the heap. This is due to the simple fact that they have been allocated and forgotten within a single GC::Collect cycle – perhaps by existing as a short-lived local variable. For this reason GC::_mergeDeadSpace initially condenses objects which were marked as dead in the same order that they exist on the heap.

The second stage relies on using a red-black search tree to order the regions by pointer in an efficient manner. At this stage the existing deadspace regions are added to the list of heap gaps.

The third stage iterates through the search-tree in order and combines adjacent heap gaps and adding them to the deadspace directory.

The final stage takes note of the special case that deadspace exists at the end of the heap. In this case the deadspace can be simply merged with the rest of the heap by rewinding the *curptr* on the heap.

In debug mode, the GC::_mergeDeadSpace takes special precautions to ensure that no space is lost or unaccounted, that all space reclaimed is cleared and that all pointers marked as dead are valid heap pointers.

Allocating from the deadspace is a reasonably simple matter. Deadspace is stored in GC::_deadspace as a std::set<Deadspace> ordered by size. This means finding the smallest deadspace region of size greater than or equal to the requested size a simple matter in $O(\log_2 n)$ time where n is the number of

deadspace regions available. If a region is available of exactly the right size, we simply remove the deadspace region from the list and yield the associated pointer to the caller. If the region is not exactly the right size (i.e. larger than the requested size) we de-allocate the region from the directory, decrement it by the appropriate size and return the pointer at the end of the region. The remaining region is then returned to the deadspace directory.

GC::Create

GC::Create is the managed way of creating an object in the *Xinaos* runtime. There are two forms of the method – the first takes a pointer to the metatable describing the managed type to be created for creating new managed objects; the second form takes two parameters detailing the managed type and number of elements in the managed array to create.

The general operation of both methods is reasonably similar. Firstly the size of the object is calculated. In the case of an object creation this is simple – it is simply a property of the type. For arrays the length is first sanity-checked to check that it is non-negative, and then the type is checked to determine if the array is a value or a reference type. If the type is a reference type, the array size is sizeof(GCPtr)*length bytes long. If the type is a value type, the total size is type->size * length bytes long.

Having determined the number of bytes to be allocated and having sanity-checked the inputs to the method, GC::_allocate is called upon to allocate a block of memory on the heap.

GC::Create then creates a new LiveObject instance in the live-object directory which stores information about the object, including the type, size and heap pointer before returning.

Garbage Collection

Garbage collection is the art of finding which objects are “dead” and reclaiming their memory and adding it to the deadspace directory. In order to fully understand how to implement a garbage collector – and in particular the GC::Collect method, we first need to understand what it means for an object to be “dead” and no longer accessible to the runtime.

In traditional native code, objects on the (non-managed) heap are accessed by pointer. The pointer itself is held either in a register or on the stack, or occasionally held as a field to another object and is accessed by dereferencing the container object (for instance a linked-list). The difficulty with native code derives from the fact that a pointer is simply an index into main memory – that is a pointer can be treated as an integer and an integer can be treated as a pointer. This leads to complications when pointers are “created” by pointer-arithmetic – indeed as soon as we allow pointer arithmetic to generate new pointers, every location in memory is, in principle, accessible to the runtime, and thus all objects in memory would remain live.

In order to compensate for this problem, we do not (as a matter of course) allow pointers to be generated by pointer arithmetic in managed code. This said, pointer arithmetic is not in itself bad – the garbage collector makes heavy use of it as a mechanism to subdivide the heap for example – it is simply that programs which make heavy use of it are difficult to analyze statically, and difficult to dynamically garbage collect.

The next problem is to determine what it means for an object to be live. An object is said to be live if it is accessible to the program – that is that there exists a managed or unmanaged pointer to the object either in a register, on the stack (held as a local either in this function or some function higher in the call stack) or held as a managed field in an object which is itself live. An additional special case is with static objects which are always accessible, and thus are always live.

Although detecting dead objects is the task of the garbage collector – doing so directly is substantially more complicated than detecting the live objects, so in the *Xinaos* garbage collector, dead objects are found by negating the set of live objects from the set of all objects.

The StackFrame class

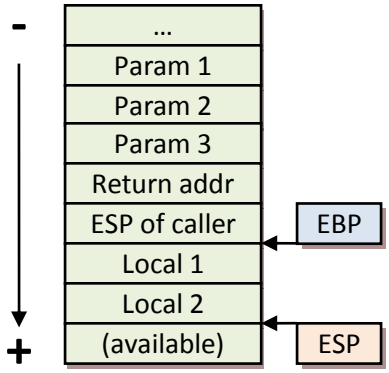


Figure 18: Stack layout in an x86 machine

One of the places we look for live objects is on the stack, however accessing the stack is not an easy task. In the *Xinaos* runtime this task is subcontracted to the StackFrame class. It should be noted that the StackFrame class is highly architecture dependent, and requires both that the program is compiled using the __cdecl calling convention and uses x86 assembler code to obtain information about the current stack frame.

The StackFrame class provides no public constructors, but provides two static methods – StackFrame::Current obtains the current stack frame, and StackFrame::SetBaseFrame informs the StackFrame class of the lowest managed function frame, and is called by the *Xinaos* runtime during program startup.

StackFrame::Current obtains the current stackframe from the perspective of the caller, rather than the perspective of the StackFrame::Current method (that is to say calling StackFrame::Current gets the current stack frame, but the function call to StackFrame::Current means we need to go up one stack frame to preserve meaning). Doing so requires us to realize that under the __cdecl calling convention prior to calling a method the return address is pushed to the stack, and on arriving at the new method, the current stack pointer (esp) is moved to the ebp register so that the esp register can be incremented to make room for the locals of the new stack frame. We can then obtain the return address by

inspecting the ebp register (which holds the old value of esp), incrementing by 4[13] to obtain the return address of the function.

In any stack frame, the current value held in ebp is the value which was being held in the esp of the previous function. This tuple of return-address and previous esp value give a unique identification to the stack-frame, and is returned as the result of StackFrame::Current.

StackFrame instances expose the value of esp and the eip register of the stackframe as void pointers, and expose one method – StackFrame::Parent. Obtaining the parent of a StackFrame instance takes the esp value of the current frame, and considers it as the ebp register of the parent frame. Finding the esp and return address of the parent frame follows a similar procedure to before – the return address is [ebp + 4] and the esp is simply [ebp].

Detecting liveness

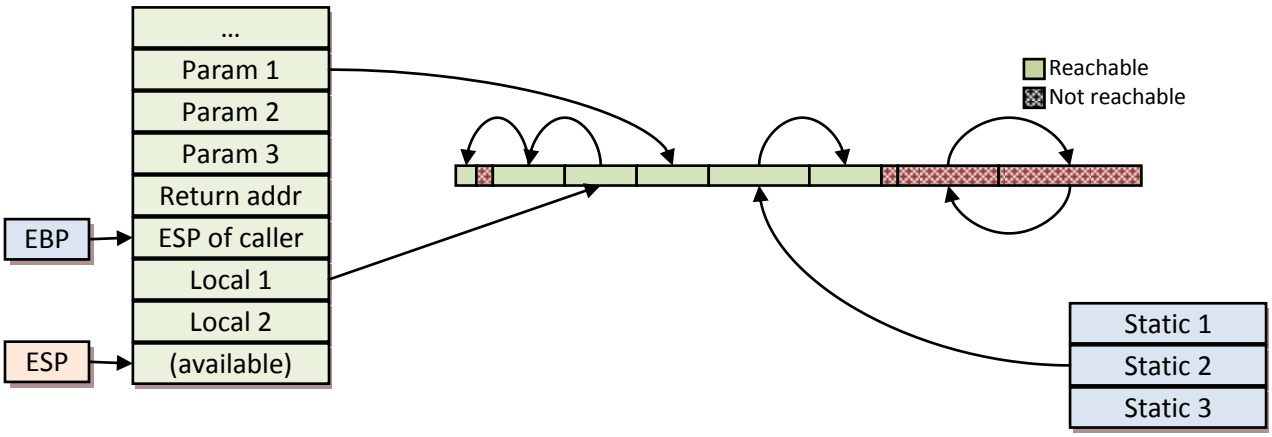


Figure 19: Detecting liveness in the garbage collector

Normally speaking, the garbage collector keeps track of all live objects in GC::_liveObjects as a set ordered by heap pointer.[14] Detecting liveness is carried out by GC::_findLiveObjects which makes use of the GC::_markRecursively helper method.

In order to efficiently compute which objects are live and which are not, taking into account the cost of set insertion and removal, GC::_findLiveObjects makes use of a temporary second object directory, GC::_liveObjectsSwap. Objects are removed from GC::_liveObjects and placed in GC::_liveObjectsSwap when they are detected to be live, meaning that when GC::_findLiveObjects returns, GC::_liveObjectsSwap contains all live objects and GC::_liveObjects contains (counter intuitively) all dead objects.

[13] All integer registers on x86 machines have width of 4 bytes.

[14] This set is stored as a red-black search tree to allow efficient insertion and $\log_2 n$ searches for object by heap address

The GC::_markRecursively method is responsible for adding an object to the set of live objects. Doing so is relatively straightforward. Firstly the object is searched for in the GC::_liveObjects set. If the object is not found, it has already been marked as live (or is not a valid heap pointer[15]), and thus no further action is needed. Otherwise the object is removed from GC::_liveObjects[16] and added to GC::_liveObjectsSwap. All managed fields in the object are then interrogated in-place on the heap, and any non-null references are recursively added to the list of live objects.

Despite the misleading name, GC::_markRecursively now uses an iterative rather than a recursive approach.[17]

The GC::_findLiveObjects method is simple. We need to add all locals, parameters and static variables to the set of live objects (all the way up the call-stack), and marking them recursively via the GC::_markRecursive function. Static variables are managed using the GC::AddStaticStorageToWorkingSet(void** location) method, which is called by the runtime during the just-in-time compiler's initialization phase before static initializers are run for the first time and by the static_gcptr native class. These sites are stored as a list in the GC and are simply iterated and marked recursively.

Detecting locals and parameters is a more complex task, but with a cursory knowledge of system architecture we can notice that all locals and parameter are stored on the stack, and exist between the stack pointer of the base-managed frame and the current stack-pointer. Since both of these pieces of information are readily available[18], we can simply traverse the data between the two pointers. Because of the way stacks are designed from an architectural point of view, all stack variables are dword-aligned, and thus enumerating the values on the entire managed stack is equivalent to enumerating all dword-aligned pointers between the base and current stack pointers. If at any dword-aligned position between these two pointers we find a number which lies between the beginning and end of the heap, we treat the position as a heap pointer, and add the element to the set of live objects.

GC::Collect

GC::Collect is the high-level entry point to the general collection phase of the garbage collector. There are two forms – one of which takes a StackFrame parameter in order to allow exclusion of high-level (native) function frames from having their local variables treated as live.

---

[15] This is not, strictly speaking, an error. Since searching for heap pointers occurs by interrogating the stack, and the stack makes no distinction between managed objects, pointers and any other bit-field (such as integers and floats), there is no guarantee that just because something *looks like* a heap pointer, that it actually is. A result of which is that it is possible for false-positives to be picked up – that is that some objects are marked as live when they are in fact dead. This is not too serious (since at worst it means an object stays alive longer than it should), and this eventuality occurs very infrequently due to the distribution of stored values in typical programs.

[16] This can be done in $O(1)$ time, since the search for the object can be cached. This means this operation takes at most two pointer operations.

[17] Since September 9th 2008. An error involving recursive marking of very large linked-objects (e.g. a linked-list) can result in a stack overflow if unchecked. By refactoring to an iterative solution, the function also avoids superfluous method calls.

[18] Via StackFrame::Current().stack_pointer and StackFrame::baseFrame.stack_pointer respectively.

GC::Collect first makes a request to GC::_findLiveObjects. When this method returns, all dead objects are held on GC::_liveObjects and all live objects are held on GC::_liveObjectsSwap. GC::Collect then iterates through all dead objects to see if any need finalization. Objects which have gone out of scope and still require finalization are added to the GC::_objectsPendingFinalization vector, and are then added to the set of live objects. The rationale here is discussed in the finalization section and is to do with the fact that the finalization method may make the object live again, and that all reachable fields from the object may be needed during the finalization method.

All objects which are subsequently left remaining on the GC::_liveObjects list are then, by definition, no longer accessible from the program. These objects are not accessible from any local, parameter, static variable or finalizer method, and thus can be immediately reclaimed. These objects are passed to the deadspace manager via GC::_markAsDead method, and then the GC::_mergeDeadSpace method is invoked to merge any regions of contiguous deadspace together. This space can then be used in future allocation operations in the garbage collector.

Once this stage has been completed, all objects on the GC::_liveObjects list have been reclaimed, and the objects on the GC::_liveObjectsSwap form the complete set of live objects in the program. We then clear the GC::_liveObjects list and swap the two sets over.

When GC::Collect returns, GC::_liveObjects contains the list of all potentially reachable objects, GC::_objectsPendingFinalization contains live objects which went out of scope but have yet to be finalized and all dead objects have been reclaimed by the deadspace manager.

Object Finalization

So far we have talked mainly about reclaiming memory when objects go out of scope and how to eliminate the problems associated with leaking memory when pointers are lost. In reality, however it is not simply losing memory which occurs when resources are dropped, but also failing to properly clear up system resources. Failure to properly close a FILE pointer, a WinSock handle or a HWND instance means never being able to reopen the file or socket again or interact with the window until the entire process exits, at which point the operating system will typically clear up the resources.

While native handle leaking is simply an irritation to most users, (leading to the “turn it off, turn it on” mantra for fixing bugs, many of which were caused by resource leaking), for power users – in particular large servers or applications which must run for long periods of time such as scientific computations and simulations, these resource leaks are no longer simply an irritation, but a critical point of failure for the system. If a server runs out of sockets or database handles it simply cannot perform its job. If a scientific application cannot write its results to the output file, it fails at its core task.

Finalization is the managed mechanism to deal with this problem. Because of the way the garbage collector works, we can (and do) detect when objects have been “dropped”, and as a consequence we can give classes a “last chance” function to clean up their resources before they are collected for deadspace. Of particular importance here is the notion that finalization can be built close to the encapsulated system resource which should be released. In native code if a vector of FILE classes is lost, the destructor for all of the FILE classes would never be called, since it is the responsibility of the vector

to clean up its elements. With finalization, dropping a list of FILEs means that each of the individual elements and the list itself arrive on the dead objects list, so the finalizer for each FILE can still run.

The question of implementing finalization is itself a moral dilemma. Finalization by its very nature is clearing up resources which the programmer has failed to clear up, and thus the provision of finalizers may help encourage sloppy behavior from programmers who may choose to rely on finalization rather than a deterministic disposal method such as the C++ destructor paradigm or the Dispose/finalize paradigm in C#. It should be noted that it is *not* the intention of the author to make such paradigms obsolete, or indeed to discourage their use. Finalization is by its very nature non-deterministic – the order of finalization is not well defined and a heavy reliance on finalization leads to lower performance both in terms of heap utilization and computational time spent on overheads. That said, in some circumstances it is not possible to rely on the programmer having exhausted all paths to look for resource leakages, and in such circumstances finalization is a necessary evil.

A particularly nasty problem to do with finalization is the so called “problem of resurrection”. Since finalization is a general method, it is possible that at some point during the finalizer call some live object inherits a reference to the object. At this point the object which was previously outside of the working set has re-entered the working set, including all of its associated fields.

The second problem to consider is that a collection of objects may go out of scope together – and in such cases it is not clear which order to run the finalizers in. Consider the case of an encrypted file writer (shown in Figure 20: Finalization call graph), which makes use of a file writer which makes use of a file stream. In order to reduce calls to underlying objects, each instance makes use of a buffer array. Because the buffer is intended to be transparent to the programmer who is using the object at each level, each object would want to implement a finalizer which would push the buffer to the underlying object before allowing it to be disposed. The ideal operation would then be that the encrypted file writer is finalized first, pushing its buffer to the file writer which is finalized second, before the file stream is finalized last, pushing its buffer to the operating system via the system call and clearing up the file pointer. Suppose this order were reversed. We can see that our first finalization disposes the file pointer and closes the file, at which point all the data held in the buffers of the various “higher-up” objects must be trashed, since there is no open file into which the data can go. At best we have lost the transparency of the buffering. At worst we have introduced a (potentially non-deterministic) null-reference exception when we try and push the data to the underlying object.

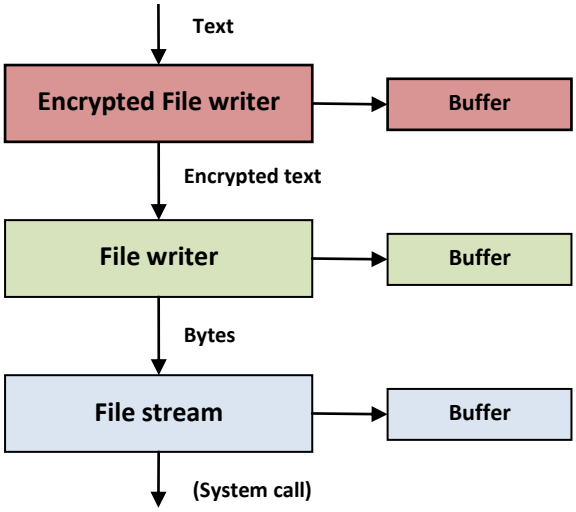


Figure 20: Finalization call graph

There are a few ways of attempting to redress this issue. The first (and arguably best) way is for each level to suppress the finalizer of the lower down objects. Creation of an encrypted file writer would create and then suppress the finalizer of the file writer which would create and then suppress the finalizer of the file stream. Now when an encrypted file writer goes out of scope, only one finalizer is executed (that of the encrypted file writer) which should take care to properly call the Dispose method on the underlying object to allow a fully deterministic disposal behavior.

A second, less manual but substantially more expensive way is to interrogate each object and build the graph above. By choosing an order of finalization such that no object is finalized before an object that references it (directly or indirectly) we are effectively "choosing" at runtime the optimal order of finalization. In the event that the graph contains loops, then the order of finalization of the loop part of the graph is entirely arbitrary, since each object is reachable from the others, and a non-deterministic order of finalization cannot be avoided.

In the *Xinaos* framework the order of finalization is dependent on the creation time of the object. Objects are disposed in the order oldest to newest. Although this has its downsides, the ordering is fair, quick to compute and reasonably simple for the programmer to understand, and takes account of object ownership models such as those in Figure 20 where the child objects are typically created in the constructor of the parent object – i.e. the child object is newer than the parent object.

In the *Xinaos* framework, objects contain a field detailing whether or not their finalizer should be suppressed. On objects which derive from classes whose inheritance tree does not contain any finalization routines, this is set to true on object-construction, and defaults to false otherwise. This field is the field which is altered by GC::SuppressFinalize and GC::ReRegisterForFinalize.

Objects are detected for finalization during the GC::Collect method, which makes use of the LiveObjects::suppressFinalize field to determine whether the object should be kept as a pseudo-live variable and queued for finalization or should be immediately reclaimed. The finalization itself occurs in

the GC::RunFinalizers method. This method effectively runs through each of the elements queued on the GC::_objectsPendingFinalization list and suppresses the finalizer on the instance before calling the types' finalizer on the object. This order is important – since a finalizer can re-register the instance for finalization, although it should be noted that incorrect use of this feature may cause an object to never be collected, since it will always be queued for finalization. If this feature is abused, the *Xinaos* runtime will never successfully reclaim the objects memory until the program exits. The *Xinaos* runtime does not encounter an infinite loop if this occurs, but rather will run the finalizer at non-deterministic points forever in the future. It is therefore strongly suggested that finalizers should never re-register themselves for finalization unless it is clear that the object has re-entered the working set.

Because finalization is an overhead, and because dead objects on the finalization queue and objects reachable from them cannot be immediately trashed, GC::RunFinalizers is always called before a heap migration in order to minimize the number of objects that need to be moved to the new heap. Because there are additionally no restrictions on the complexity of any given finalizer, it would in principle be possible for an infinite loop to be set up during a heap migration, where the finalizer would create objects leading to a further heap migration request and so on. For this reason the *Xinaos* runtime ensures that GC::RunFinalizers is non-reentrant, so if a finalizer being run creates objects that lead to a heap migration request, all objects on the heap and on the finalization queue are migrated to the new heap.

Heap Migration

So far we have discussed how the garbage collector detects dead objects, how these objects are finalized and recycled and how the garbage collector creates new objects from both the end of the heap and the deadspace between objects on the heap. What we have not discussed is what happens if an allocation is requested which cannot be serviced either from the deadspace or the end of the heap. In these circumstances we perform what is termed a heap migration – where a new heap is allocated and all live objects are moved to the new heap and re-aliased to the new heap.

This is the single most complex task in the garbage collector, and on its own took over two weeks of extensive testing and debugging, which mainly involved hex-dumps of the heap and stack.

In the *Xinaos* runtime, heap migration is performed by the GC::_forceMigrate method which takes as two parameters the stack-frame which is the most recent stack-frame whose locals must be re-aliased after the heap migration (so at least the last managed stack frame) and additionally a parameter detailing a minimum number of bytes that must be immediately allocatable as a block when the function returns.

The first task of GC::_forceMigrate is to estimate how big the new heap should be. To do so, the method first takes the sum of all sizes of live objects on the heap and adds to this the allocation-size-preference parameter to the method. Notice that this does not include the deadspace as part of the calculation. Having done so, GC::_forceMigrate asks the second GCHeap to resize to this new value. The GCHeap will expand if necessary, and round the size up to the nearest PAGE_SIZE.

We now clear the GC::_liveObjectsSwap list – we will use this list to hold the LiveObjects with their heap-pointers moved to the new heap. We additionally create a set of MigrationPairings, which is a tuple of current-heap-pointer to new-heap-pointers, ordered by old-heap-pointer.

Copying the data across is equivalent to taking each object in turn and allocating it on the new heap and copying the data between the old heap and the new heap. Note that because objects are stored in order oldest to newest in the live-objects directory, this has the effect of compacting old objects together at the start of the heap, which reduces memory fragmentation. We also add an entry to the migration pairings set detailing the new position of the object.

At this point the swap heap has been populated with all live objects, and the live-objects-swap list contains updated live objects (with updated heap pointers). The migration pairings contains a complete mapping between pointer on the old heap and pointer on the new heap. The swap heap contains all the data from the old heap (although this has not been re-aliased) and the swap heap has minimum fragmentation.

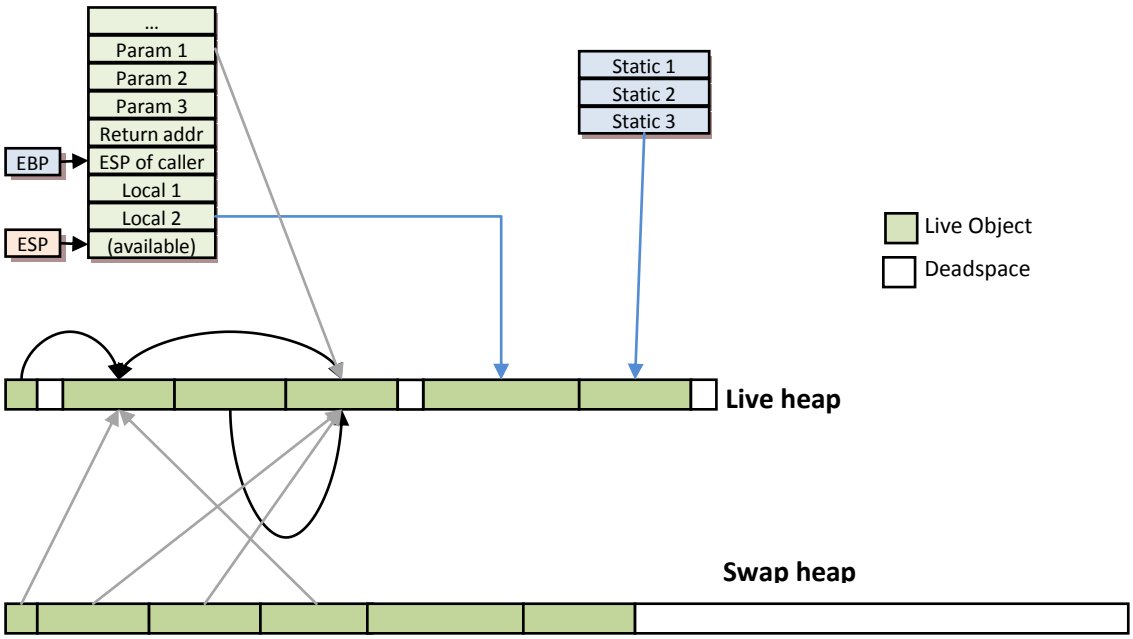


Figure 21: Heap migration stage after all objects have been copied the new heap. No fragmentation exists on the new heap, but objects on the new heap still point to the old heap, and stack and static variables have not been updated.

We now need to migrate all of the managed references in the heap to point to the equivalent data on the new heap. To do so we iterate through all live objects. If the object is a straightforward managed object, we simply inspect all of the object's managed fields (as described by the managed type-metatable) and changing the value in the field to the re-aliased equivalent on the new heap. If the object is an array of managed objects we need to re-alias each element of the array, and if the array is an unmanaged type we do not need to re-alias the data.

At this point the swap-heap is self-consistent and should have no managed references pointing outside of the swap-heap.

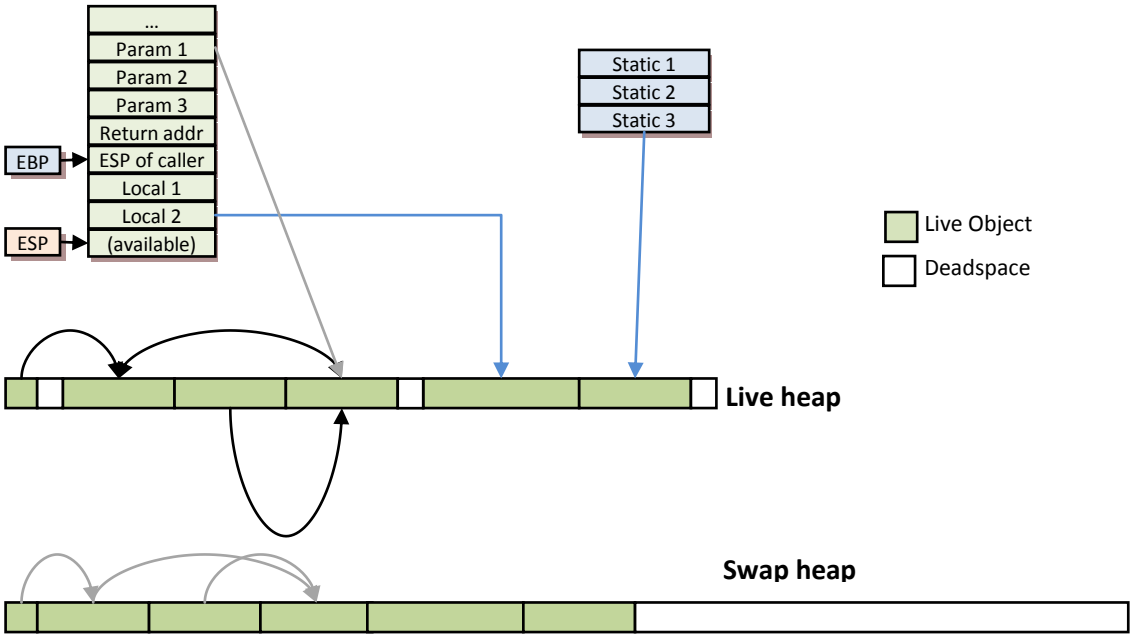


Figure 22: Heap migration stage 2: The swap heap is now self-consistent with no pointers pointing outside of the heap.

The next task is to update the working stack. This is all of the locals and parameters in the program. To do so we interrogate the stack between the stack-pointer of the stack frame passed in as a parameter and the base stack frame's stack pointer. Every value which lies on this stack that is aliased to an object in the old heap is updated to point to the equivalent object on the swap heap.

It should be noted that there is currently a problem in the *Xinaos* runtime implementation in that – at least in theory – it is possible for a non-pointer object to be accidentally updated. This is a largely a theoretical concern, since the distribution on values on the stack is such that any integer or bit-field that looks like a pointer almost certainly is – iteration indexes are the most common form of large numbers appearing on the stack and iterations over iteration spaces where the iteration field looks like a pointer are very rare so that this in practice never happens (except perhaps in constructed circumstances). Although this issue is not of paramount concern – it may be possible to avoid in future by inspecting the type of locals and parameters on the call stack using meta-information (although this would also the collector).

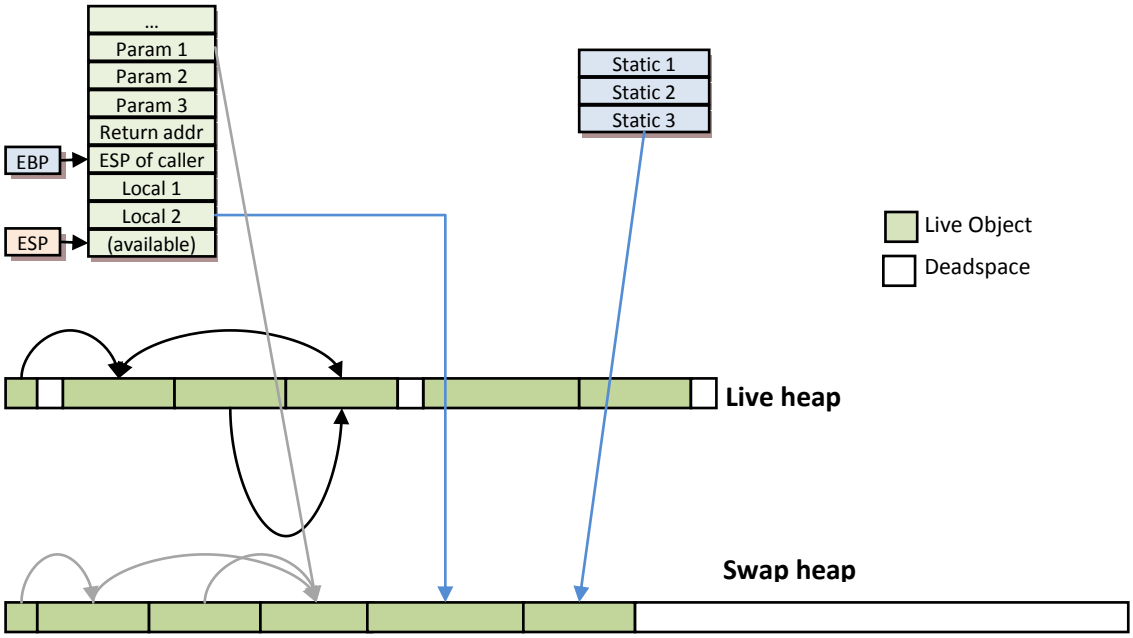


Figure 23: Heap migration stage 3: After stack and static re-aliasing. No live pointers now point inside the live heap, so we make the swap heap live and trash the old live heap

Having performed the stack re-aliasing, we are in a position where the swap heap is internally consistent and the program has been corrected so that all managed references point to the swap heap rather than the old heap. The final part to check is the static variables. This is simple, since we need only interrogate the static slots which are created via the GC::AddStaticStorageToWorkingSet method.

At this point the swap heap is internally consistent; all managed pointers in the program have been re-aliased to the swap heap and all static variables point to the swap heap, so we proceed to swap the live heap with the swap heap and the live object directory with the live object swap set. At this stage we can clear or release the swap heap (which is the old live heap) and return.

Program exit

The only part of the garbage collector that has not yet been detailed is the order of execution when the garbage collector shuts down during program exit. The first thing to do is to run all outstanding finalizers, perform a full collect and then run the finalizers from the final collect. We then run all static destructors, and at this stage if any objects are left alive we dispose them (optionally running finalizers) in the order newest to oldest. At this stage we ignore any further requests to GC::ReregisterForFinalize, and having disposed all objects, the garbage collector final task is to release the heaps.

The *Xinaos* Just in time compiler

The *Xinaos* Just-In-Time compiler (JIT) is responsible for the conversion of XIL code to x86 executable code at runtime. The *Xinaos* JIT is written mostly in C++ with many blocks (for instance relating to calling-convention conversion) written directly in x86 assembler.

When the runtime first loads, it does the following:

1. The runtime loads and interprets the file's XIL method and type table
2. The runtime "compiles" the list of structures and types in the XIL file to obtain useful information such as each type's size on both the stack and the heap, and various field offsets of the various types.
3. The runtime creates a list of structures whose layout corresponds to the layout of a managed string, and populates the list with pointer and length values taken from the string table. Note that the char pointer in the structure points to the string table data heap, and not the garbage collected heap. These strings are not referenced by the garbage collector since they are immutable and cannot reference garbage collected objects. This optimization allows string literals to be used without creating a new object for each string literal. Note that strings which are created, e.g. as a result of a concatenation operation exist on the garbage collected heap.
4. The runtime then inspects the file's XILGlobalFileHeader structure (Appendix F) to determine the entry point for the program, and calls the Just-in-time compiler for that method. The entry-point method is then invoked and the program begins.

The Just-in-time compilation of methods works thus:

1. The XIL method is first verified up to types to ensure that the method is valid[19]
2. The XIL method is interrogated for parameters and local variables, and a pre-computed locals stack is built (see later section on the pre-computed locals stack)
3. The XIL method is inspected for attributes and contracts. Attributes are decorators for methods which give additional information about how they should be used or compiled (for example, the DllImportAttribute informs the Just-in-time compiler that the method should call into a named method in a named dll, rather than running the code in the method). Because Attributes are simply a list of XILInstructions, the attribute code is immediately compiled and executed, and a pointer to the attribute is held in the JittedMethod class (the attribute itself exists on the GC heap).
4. The XIL method is interrogated for flags such as the EXTERN flag, in which case a dedicated compilation takes place dependent on the compiled attributes of the method. This is discussed later in the chapter.
5. If the XIL method is not marked with flags such as the EXTERN flag, the method is immediately compiled using the JITMethodCompiler and JITAssembler classes.
6. The JITAssembler estimates its length. This length is used to obtain a blob of memory that lies before the heap (to avoid buffer overruns in native code). The JITAssembler then compiles the x86 instructions into that memory. The memory is marked as non-writable and executable, and the method returns.

The Pre-computed Locals stack / "Shadow" Stack

One "feature" of native languages which does not hold into most managed languages is the notion of what value to expect when reading from not-yet-assigned variables; consider the C code:

[19] This feature is incomplete at the time of submitting this report.

```
int main(){
  int i;
  printf("%d", i);
  return 0;
}
```

The output of this code to the console is dependent on the value of *i*, however this value is undefined. The value is not random, however – the variable site of *i* is on the stack, and the value of *i* is simply what was on the appropriate position on the stack when main was called.

For managed languages this is clearly unsuitable. Variables sites on the stack may well be object references, and those references must be correctly aliased to an object on the heap. This can be in two ways. Firstly the compiler may use assignedness analysis to insist that a variable is not used before it has been definitely assigned. This is the mechanism used by both Microsoft's C# compiler and most Java compilers.

In the *Xinaos* compiler, we introduce the *shadow-stack*, which holds a compile-time computed image of the function stack. This has two main features. The first is that all values on the stack have definite value, set by this precomputed image, before the function is entered. Secondly we can use this feature in the compiler to set compile-time known first values into variables "for free". Aggressive compilers would also be able to make use of the shadow-stack to speculate on variable first values over if-blocks – and this could even be used to eliminate some branches entirely, without more complex branching analysis; consider the code:

```
static void foo(){
  int i;
  if( condition() ){
    i = 2;
  } else {
    i = 7;
  }
}
```

We can speculate that the branch will be taken, to generate the shorter code, shown below.

```
static void foo(){
  int i; ( = 2; )
  if( !condition() ){
    i = 7;
  }
}
```

This feature occurs almost no additional cost over the C# and Java (and in some cases, even C++) mechanism of clearing the stack with zeros, since clearing the stack with zeros is not much more expensive than clearing it with anything else.

Another interesting feature of the pre-computed stack is that the code itself becomes smaller – many initializing writes to the stack get promoted into becoming part of the pre-computed stack, and this in turn leads to smaller code size. Even better perhaps is the possibility that the just-in-time compiler can play with the values held on the pre-computed stack, and can thus promote even some managed

invariant objects onto the pre-computed stack (although this would, by definition, be a strictly a runtime optimization).

How the JIT compiler works

There are four main classes used by the JIT compiler; JITCompiler, JITMethodCompiler, JITAssembler and JittedMethod. The JITCompiler class is the main class of the JIT, and is held as a reference by the runtime, and maintains a list of compiled methods and associated tables.

The JITCompiler is also responsible for computing the shadow-stack of methods, as well as generating the parameter, local, attribute and contract meta-information relating to the methods.

The JittedMethod class contains all of the information relating to a particular compiled method, including the pointer to the executable code, the number, types and offsets of parameters, the number, types and offsets of local variables as well as the size and value of the shadow-stack.

The JITMethodCompiler class contains a register allocator and JITAssembler and is responsible for compiling the XILInstructions list into x86 instructions in the JITAssembler.

The JITAssembler class maintains a list of labels and x86 instructions, and is responsible for estimating the final size of the compiled method as well as compiling the method to executable-bytes.

The JIT Assembler

The JITAssembler class is used by the JITCompiler, and essentially acts as a collection of x86 instructions which can be added to the JITAssembler before being compiled. The JITAssembler then assembles the instructions into x86-machine code, which is then written to a pointer passed into the JITAssembler.

It is common practice in modern operating systems to prevent memory pages from being simultaneously executable and writable. This ensures that in the event that a buffer-overrun fault occurs at any point in the program, it is not possible for the fault to be maliciously exploited to run executable code via a buffer-injection attack. This poses a significant problem to programs which wish to write their own instructions before executing them – something that lies at the heart of how a JIT-compiler works. To avoid the entire runtime being halted by the Data-Execution Prevention policy on Microsoft Windows, we make use of the VirtualAlloc and VirtualProtect methods to encapsulate and protect JIT-methods. Although buffer overrun attacks (provably) cannot occur in managed code, we cannot guard against malicious attacks on unmanaged DLLs linked by the program. For this reason we ensure that JIT-methods are always located at logically lower addresses than the heap. This ensures that even if a buffer overrun were to occur on a heap object, it could potentially trash the heap, but not the method code.

Furthermore, buffer overruns on stack objects remain protected from Data-execution-prevention, and thus malicious buffer-injection attacks do not work on the *Xinaos* runtime.

In order to provide even greater ease-of-mind, the use of VirtualAlloc and VirtualProtect is such that we always guarantee that the JIT-methods' pages are never simultaneously writable and executable, that is to say when the JITAssembler is writing to their pages, they are marked as read/writable, and when the

JITAssembler has completed and before a call is made to the new method, it is marked as executable and non-writable.

Although the *Xinaos* framework is currently only compatible with Microsoft Windows, all Windows-specific code is abstracted to the winhooks.cpp file, which references windows.h. Method signatures for winhooks.cpp are defined in oshooks.hpp, and thus making the entire *Xinaos* framework available on other operating systems is as simple as implementing the method signatures of oshooks.hpp in the target operating system of choice[20].

Working out the machine op-codes for x86 is less trivial than you'd think – the Intel manuals [1] [2] are less than helpful in describing anything other than the human-readable NASM [3]/MASM syntax. To this end there are two thunks in the JITMethodCompiler class which ask the C++ compiler to output assembler code which can be searched for in the binary-output and used to compute the binary result of an assembler instruction (these are disabled on release builds).

Interestingly there are a number of bizarre inconsistencies in the Intel machine-codes. For example, storing the value held on EAX into a value offset by eight from another register is shown in Figure 24. On the right of the table is the instruction and on the left of the table is the bytes that correspond to that instruction (shown as hexadecimal pairs). The instructions are ordered by register identifier value (which is *not* lexicographic order). The 89 byte corresponds to the "move register into memory given by other register" instruction. The first two bits of the second byte correspond to "small constant offset". The next three bits correspond to "from register EAX" and the low order three bits correspond to the "offset from …" register. The final byte corresponds to "offset by 8 bytes".

Such general rules can often be obtained by inspection, however sometimes, as here, Intel throw a spanner in the works by randomly deviating from the general pattern and adding an additional byte if we're offset from ESP – in this case the byte 0x24.

The Just-in-time assembler takes even these bizarre cases into account when compiling instructions, however it can (and did) lead to some fairly major headaches when it comes to debugging just-in-time compiled code.

Machine code value (in hexadecimal)	*instruction (shown in increasing register value)*
89 40 08	mov dword ptr [eax+8],eax
89 41 08	mov dword ptr [ecx+8],eax
89 42 08	mov dword ptr [edx+8],eax
89 43 08	mov dword ptr [ebx+8],eax
89 44 24 08	mov dword ptr [esp+8],eax
89 45 08	mov dword ptr [ebp+8],eax
89 46 08	mov dword ptr [esi+8],eax
89 47 08	mov dword ptr [edi+8],eax

[20] Although the *Xinaos* could be easily migrated to other operating systems, it would require a new implementation of the JITCompiler and JITAssembler class to port the *Xinaos* project to an entirely different architecture.

Figure 24: An example of the inconsistent machine-code outputs. Notice that storing into the memory location given by ESP has a completely unexpected 0x24 byte which is completely out of step with the other instructions

Another point that should be made is that the x86 instruction set provides no opcodes for jumping or calling to absolute addresses. The rationale is because when "normal" programs are compiled, all the instructions that the program will ever want to run are already statically known, and thus by using relative offsets for everything the program can remain agnostic about where it is loaded to in memory.

The downside to this is that when instructions are not known and haven't even been compiled by the time the executable has been loaded into memory, we can have some problems emitting opcodes that get to the right address. One solution would be to load the address into a register and perform a virtual call (call /r32) to get to the correct address, however this has its own problems. Virtual calls not only destroy a processor's ability to branch predict – thus slowing the program down – but they also require a free register and additional move instruction, which slows down and complicates the program yet further.

In the *Xinaos* compiler this means that the exact form of an instruction such as JMP or CALL is not known until it has begun writing to the executable memory, which is necessarily after it has estimated the length of the assembled code. For this reason, all branching instructions are pessimistic about how long they will be and pad their length out with NOP instructions if they find they can use a shorter form of the x86 instruction later. Because label offsets are computed during the length-estimation phase, this means that all jumps to labels beyond the current instruction (but not backward looking jumps); or calls or jumps to absolute addresses use the long form of their respective machine instructions, even if a shorter one could otherwise have been used.

Jit Method Compiler

At the base of the JIT method compiler lies a JITAssembler and a register allocator which are responsible for storing the x86 output of the method and the virtual-physical register translation respectively. Compiling the methods is then reasonably straightforward. For example, to compile the LDNUM *n* instruction which is responsible for loading the value *n* onto the top of the virtual stack, we do the following:

```
void JITMethodCompiler::opcode_LDNUM(DWORD value){
	x86Register dst = new_reg();

	if(value == 0)
		jitAssembler.Append(new x86_XorRegReg(dst, dst));
	else
		jitAssembler.Append(new x86_MovRegImm(dst, value));
}
```

The *new_reg* function obtains an unused register from the register allocator (if all the available registers are currently in use, it is responsible for pushing the least-recently used register to the stack and reusing it – the register allocator also ensures that it is popped again before it is used).

We then add a new move instruction that clears the new register with the value-to-be-loaded, although we make the additional observation that `xor r1, r1` is equivalent but slightly more efficient than `mov r1, 0`.

Each opcode has its own method implementation, ranging from the very simple (LdTrue translates to Mov new_reg, 1) to the very complex. NewObject, for instance needs to push the final argument to the constructor, push all in-use registers, compile the constructor, insert a method thunk to create the object, call the constructor and then put the result in a new register before popping the in-use registers again.

Some instructions are completely virtual – for instance the POP, DUP and NOARGS instructions translate into actions for the register allocator but do not yield any actual x86 instructions.

Not all of the x86 instruction set are register agnostic. A consequence of this is that the register allocator allows instructions to demand particular registers. In these cases the register allocator looks at the state of the register.

For example, the XIL instruction *MOD* uses the *IDIV* machine-instruction which takes the numerator on the EDX-EAX pair and the denominator on some other register and returns the modulus on EDX. The MOD instruction cannot simply obtain any new register, but must demand specifically that it obtains EDX from the register allocator.

If EDX is already free, it can be marked as in use and returned. If the register is not free, but is already mapped to a position (or this pointer) on the virtual stack, the register allocator allocates a new register (e.g. ESI) and adds an instruction which copies EDX into the new register. The register allocator then simply updates it's virtual to physical register stack so that any previous stack positions that held their

value on EDX now look to ESI for their value. This eliminates the need to move the result back once the MOD instruction no longer needs EDX.

Calling convention of JIT-methods

JIT-ed methods are entirely constructed in x86 assembler, and thus do not automatically inherit the C/C++ *cdecl* calling conventions. The JIT-methods use a calling convention similar to *stdcall*, but with some minor alterations to allow for passing 'this' parameters.

Summary of JIT method calling conventions

Element	Implementation
Argument-passing-order	Arguments are passed in reverse order (right-to-left)
Argument-evaluation order	Arguments are evaluated in left-to-right order.
Argument-passing convention	All arguments are widened to 32-bits when they are passed. All parameters are passed by value, unless the parameter's type is a reference-type, or the parameter site is marked using the *ref* or *out* keyword, in which case a pointer is passed on the stack instead.
Stack-maintenance responsibility	The called function pops the arguments from the stack.
'This' parameter-passing	Passed as a pointer on EBX for methods which are not marked *static*.
Return values	All results are widened to 32-bits if necessary and return on EAX, except if the result is an 8-byte structures in which case the method returns on the EDX:EAX register pair. If the result is more than 8-bytes, EAX returns a pointer to a hidden return structure (which lies beyond the end of calling function's stack)
Preserved registers	ESP, EBP and EIP are the only registers preserved across a call.
Variable length parameters	Variable length parameters are not supported in the *Xinaos* runtime. There are no plans to introduce this feature in future.

Parameter passing

For JITted methods which require parameters, all parameters are placed on the stack in right-to-left order by the caller. Note that this is **not** the same as in XIL, where parameters are passed left-to-right, and note also that this does **not** violate the semantic that parameters are evaluated left-to-right. Parameters *are* evaluated left-to-right, but are *passed* right-to-left.

Parameters which are less than the processor-word-length[21] are extended to the processor-word length before being passed.

Parameters which are reference types, or which are passed using the *ref* or *out* parameter modifiers are passed as pointers to the object data. Reference types which are passed using the *ref* or *out* parameter modifiers are passed as pointers to the variable-site which stores the object reference (a double pointer to the object data).

[21] Currently only 32bit processor-word lengths are supported

As with the *stdcall* calling convention, it is the responsibility of the called method to clean up the stack frame.

The JIT compiler is allowed to modify the calling convention for various specific methods (in particular with regard to which registers are preserved across function calls, or expecting parameters on registers to improve the speed of method calls) if doing so would benefit the program as a whole, however if it does so, the JIT compiler must be careful to ensure that all methods which call into that method are aware of the change, and further that if the method is referenced either virtually, via a delegate or via a function pointer for the operating system, that the method is still able to be called without double-thunking (in practice this means that virtual methods and delegates must check that their referenced method strictly adheres to the calling convention above, and if it does not, must compile a version of the method which does).

The reason for using the stdcall calling convention is no accident. Some calls to the operating system require method pointers (for example the WndProc procedure when creating window handles) and these are called by the operating system expecting the stdcall calling convention. Although the stdcall calling convention is not mandatory (the Microsoft .NET runtime uses double-thunking to get around this problem) the efficiency loss is significant.

'This' parameters

Instance calls, which are called using the *Call* or *CallVirt* opcodes send a pointer to the object (either as a managed pointer to the heap or a pointer to a local stack frame's reference of a value type object) on the EBX register.

A consequence of this is that when a new object is created on the heap, the gcnew thunk returns on EBX rather than EAX, in order to allow call-chaining to a constructor.

When performing calls on value-type objects, the PassByRef opcode must be used after the 'this' parameter. *Rationale:* Although value-type objects are typically passed by value, member invocations on value-types should be able to alter the left-hand-side of the invocation via the 'this' parameter *end rationale*.

Delegates

Delegates always refer to static methods, and use the *CallDelegate* opcode to call the delegate. Delegates are obtained from method signatures using the *NewDelegate* opcode, which returns a pointer to the JITted-code of the method. Note that this necessarily means that just-in-time compiling a method which contains a *NewDelegate* opcode will cause a just-in-time compilation of the target method.

Return values

Methods return on EAX.

If the result is a reference type, EAX holds a pointer to a managed object on the heap.

If the result is a value type less than 4-bytes, the result is sign-extended to 32-bits and returned on EAX.

If the result is an 8-byte value type (e.g. an Int64 structure), the result is passed out on the EDX:EAX pair (this pair is chosen because most 8-byte processor instructions return on EDX:EAX).

For all other value types, EAX holds a pointer to a hidden return structure (which is not guaranteed to be valid beyond the next *call*-type instruction, *throw*-type instruction or *AllocObject, NewObject* or *NewArray* instruction (and thus must not be used as a parameter, or stored into an instance or static field), but which is guaranteed to be valid beyond an immediate *retval* instruction, so can be used for tail-returning objects).

Rationale: The actual position of the hidden return structure is implementation defined, but with the current implantation is held on the local stack of the called function, and remains valid data until the next call-frame deeper than the current one is obtained (or the current function returns and a new call is issued). Because the structure lies beyond the end of the stack, the garbage collector's heap migration routine would not correctly migrate managed pointers held inside the structure during the heap migration, however this such a migration is guaranteed not to occur unless you specifically ask the GC to perform a collect (which would nessistate a call) or a new object to be created on the heap. *End rationale*

Register preserving

The EAX, EBX, ECX, EDX, ESI and EDI are by-default not preserved across method calls (although this can be negotiated on a method-by-method basis by the JIT compiler). EIP and EBP are always preserved across method calls. EFLAGS is not preserved across method calls. ESP is not preserved over a method call, but when the method returns ESP points to the position on the stack immediately before the parameter list (i.e. the called function has cleaned up the parameters from the stack)

Variable-length parameter lists

Variable length parameter lists may not be used in XIL method signatures, however, strongly-typed variable length parameter lists are valid in C#. In general this is performed by constructing an array and passing the array of parameters to the method, however the optimizer may make multiple copies of the target method with varying lengths of the parameter list, which may be further optimized based on the length of the parameters passed, as well as a general form of the method which takes an array. This optimization can reduce the need to construct arrays to be passed on the heap.

Function prologue and epilogue

The method prologue and epilogue follow almost exactly from the *cdecl* definition, with the only variation that they copy the shadow-stack to where the local-variables will be stored as part of the function prologue:

General case function prologue and epilogue

```
method:
  push ebp          ; save caller's EBP register
  mov ebp, esp      ; setup the current stack frame's EBP

  sub esp, shadow_stack_length          ; create space on the stack for the locals
  mov esi, addr_method_shadow_stack     ; this is constant at JIT-time
  lea edi, [esp]
  mov ecx, shadow_stack_length          ; number of times REP should run (in bytes)
  rep movsd                             ; copy the shadow stack to ESP.

  ; method code

  mov esp, ebp      ; restore the caller's ESP
  pop ebp           ; restore the caller's EBP
  ret PARAMSIZE     ; restore the caller's EIP and clean up the parameters
```

Function prologue/epilogue for methods with no local variables

```
method:
  push ebp          ; save caller's EBP register
  mov ebp, esp      ; setup the current stack frame's EBP

  ; method code

  mov esp, ebp      ; restore the caller's ESP
  pop ebp           ; restore the caller's EBP
  ret PARAMSIZE     ; restore the caller's EIP and clean up the parameters
```

Function prologue/epilogue for methods with no local variables or parameters

Note that this type of method may still take a THIS parameter, which is passed on EBX and not on the stack. This function signature is quite rare, as methods simple enough to benefit from this function signature are typically small enough to be inlined at the call site.

```
method:

  ; method code

  ret PARAMSIZE     ; restore the caller's EIP and clean up the parameters
```

Runtime-to-JIT calls

All of the data relating to JITted methods, including the code and stack pointers are held in the runtime by the JittedMethod class, which exposes the *Run* method, which takes as arguments a pointer to a list of arguments and the number of arguments to pass.

The method acts as an inline assembly thunk to the jitted method, and performs an effective calling-convention translation between the JIT method and the *cdecl* calling convention of the runtime as a whole.

JIT-to-JIT calls

Compiled methods which reference other compiled methods are inserted as simple call instructions in the emitted binary executable code, thus jit-to-compiled jit code is as efficient as a call between two methods in compiled C code. Compiled methods which reference other as-yet-uncompiled methods go via a method thunk which takes the method index on EAX and returns on EAX.

The method thunk preserves the EBX register (which may be needed for instance calls) before passing the method index onto the JIT compiler. When the JIT compiler returns on EAX, the thunk restores EBX and performs an unconditional jump to the method which allows the newly-compiled code to return directly to the caller of the method thunk.

Because of the potential for re-entrant method thunks, the JIT compiler uses mutual exclusion to only allow one compilation of a method at once. If a method thunk is called and the JIT compiler is compiling that target method, the method stalls until the thunk is ready.

The JIT-compiler can and does recompile methods to make them more efficient, however doing so requires careful analysis of the call-stack to ensure that the return sites for methods are properly aligned (so that if a method were to return to a poorly-compiled JIT-method and a better JIT-method that corresponds to the same managed code is constructed, the call-stack should be modified so that the program will eventually return to the better JIT-method in a place that is semantically equivalent to where it would otherwise return to in the older JIT-method).

Non-managed methods

There are two types of allowable call from C# to non-C# methods. There are "internal calls" which call from managed code to the runtime, for example GC::Collect and Environment::FailFast where the runtime knows that the call comes from managed code and can take this into account. There are also "external calls" which call from managed code to system dll files, for example calling into the *kernel32, user32* and *gdi32* family of DLLs for performing calls to the operating system for example in relation to creating, modifying and drawing the program window.

All external methods are marked with the EXTERN flag in the XILMethodHeader descriptor (Appendix F) and thus when the runtime comes to compile the method, it checks the attributes of the method to see how it should handle the method compilation.

Extern methods decorated with the InternalMethod attribute are deferred either to internal thunks which correspond to the appropriate internal method, or for simple methods, just direct inline assembler code to be compiled in lieu of the method itself. For example, if a method is decorated with the InternalMethod attribute whose parameter is 2, the method defers execution to the Array.Length method thunk which returns the length of the array on EAX.

Extern methods decorated with the DllImportAttribute contain two managed strings as parameters which correspond to the dll and method name of the PInvoke signature respectively. Unlike internal methods, however, PInvoke signatures do not have the benefit of knowing about managed types, and as such all PInvoke signatures go via a thunk which translates certain managed types into unmanaged equivalents.

Let's demonstrate using a concrete example. Suppose we write and compile the following code in C#.

```
static void Main(){
  MessageBoxA(0, "Hello World", "Caption", 0);
}
[DllImport("user32.dll", "MessageBoxA")]
static extern void MessageBoxA(int h, string text, string caption, int p);
```

In the runtime we compile and run the main function, which loads the parameters to MessageBoxA on the stack. Note that the strings are loaded as pointers to the string table (and loading them does not create any objects) but that the strings are stored as managed strings which are length-prefixed rather than nul-terminated.

We then reach the method call. Because the method MessageBoxA had not been compiled when we entered the Main function, this will be via a call thunk that attempts to compile and run MessageBoxA.

The first thing that the JIT compiler does when it sees MessageBoxA is it attempts to read the meta-information about MessageBoxA. It will find that it takes four parameters at offsets 4, 8, 12 and 16 offsets from EBP. It also discovers that it has an attribute, and compiles and runs the partial-method associated with the attribute.

The partial method associated with the attribute is simple, and corresponds to loading two strings on the virtual stack before calling NewObject DllImportAttribute. This creates the DllImportAttribute on the heap, as well as calling the DllImportAttribute's constructor (which is compiled at this point if necessary). Ultimately the partial method returns with a pointer to the DllImportAttribute on the heap which is then stored in the JittedMethod corresponding to the MessageBoxA.

To avoid the DllImportAttribute being garbage collected, the JittedMethod's attribute sites are registered to the garbage collector as static storage locations which can hold managed objects; thus if the DllImportAttribute were to be moved on the heap, the pointer to the managed object in the JittedMethod would be updated accordingly.

We now return to the JIT compiler responsible for compiling the MessageBoxA function. By inspecting the flags associated with the method we discover that the method is marked with the EXTERN flag. We then inspect the attributes, to discover there is exactly one attribute which corresponds to the DllImportAttribute object type. By knowing the layout of the DllImportAttribute we can extract pointers to the two strings which correspond to name of the dll and method that the extern method should point to. Under Windows these are converted to nul-terminated strings and sent to the LoadLibrary and GetProcAddress methods respectively to obtain a function pointer to the appropriate function. If this yields the value NULL then the method does not exist and we halt execution with a fault (PInvoke signature does not exist).

Otherwise we now have a pointer to the function that we should ultimately call, but even at this stage we can't immediately jump to this location. Instead we compile a mini-thunk which will convert the managed parameters to native ones before we perform the call. The mini-thunk puts the address of the method on EDI and a pointer to the jittedMethod on EBX before jumping to the managed-to-unmanaged parameter conversion thunk. This mini-thunk is compiled and returns.

We now return all the way back to the original call method thunk which was called from Main. This now calls into what is the newly compiled mini-thunk which loads EDI and EBX and jumps to the managed-to-unmanaged parameter conversion thunk. This uses the jittedMethod (which is helpfully sent on EBX from the mini-thunk) to work out the parameter types being sent. It then iterates through the parameter list, doctoring those which correspond to managed types which have a native conversion; for example strings are converted from length-prefixed syntax to nul-terminated syntax. The parameters are then reloaded on the stack and the native function pointer (given on ESI) is called.

When the native pointer returns, we perform a few checks to ensure that ESP and EBP have been properly restored (an inconsistent ESP typically means that the PInvoke signature is declaring its parameters differently to how the DLL defines them) and clears up any native pointers generated as part of the conversion routine. We then have to do the winding-up-the-stack bit of the *stdcall* routine manually, since the parameter conversion thunk has no idea how many actual parameters we sent it.

We return from the method (storing the result on EAX) having removed parameters from the stack, and we return to the original Jitted Main function which can or discard the result before continuing along its executable code.

Although it would be a fair criticism to say that this is an expensive mechanism for calling between managed and unmanaged code – it is not unreasonably so. Storing strings as length-pointer pairs not only makes discovering the length of a string an $O(1)$ operation, it also means that forming all string substrings and selected string concatenations do not need to make a copy of the data – reducing both the memory imprint of the executable as a whole as well as reducing the work (and thus increasing the speed) of the garbage collector.

The cost of performing compile-and-go attribute blocks could perhaps be reduced in exchange for losing generality, however the compile-and-go system allows attributes to be complex structures and use real methods and constructors as part of their execution.

Moreover, while the cost of performing the first PInvoke is high, subsequent calls to the native method only require a managed-to-native parameter conversion – the attributes, mini-thunk and function-pointer-acquisition do not need to be recomputed.

Parallel Runtime library

The parallelism runtime library is responsible for the translation of compiler-detected parallelism opportunities into parallel executable blocks at runtime. It is responsible for the creation and maintenance of parallel threads and is responsible for ensuring that tasks are evenly spread across the processor-cores available in the machine.

The parallelism runtime is also accessible by the programmer to use explicitly, although the main intention is to use the parallel runtime to implement the parallelism that the compiler detects in the code. The code is shown as UML in Figure 25.

The syntax for use of the library is heavily based on the public facing API in the Microsoft Parallel Library, although the implementation and actual behavior of the library are new.

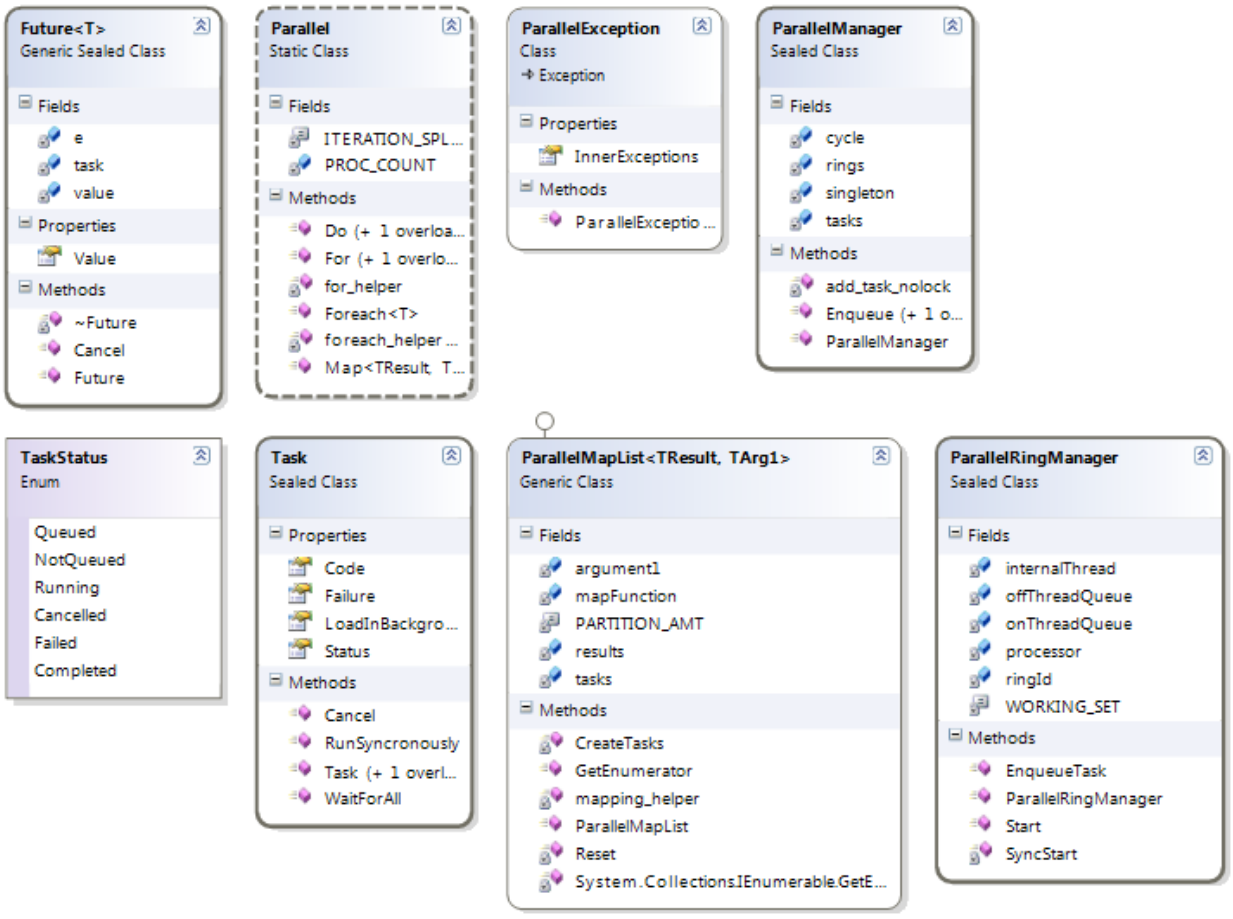


Figure 25

Tasks

The basic block of the parallel runtime is the **Task**, which represents some piece of code to be scheduled on one of the cores at any point in future, and works by encapsulating a delegate with no arguments

and no result. All **Task**s are mutually independent of each other when scheduled by the compiler, and thus their running order is not important and they are automatically queued by the **ParallelManager** when created.

Tasks generally perform their actions asynchronously to the main application thread, although they can be run synchronously if their completion must occur before the next statement on the application thread. If the **Task** has yet to be performed, the **Task** is de-queued and performed synchronously on the application thread. If the Task is currently being actively performed in the background then the application thread blocks until the Task has completed. If the Task threw an exception when operating in the background, the exception is raised at this point on the application thread and if the Task has already successfully completed then the application continues normally from this point.

Task-Scheduling - diagrams

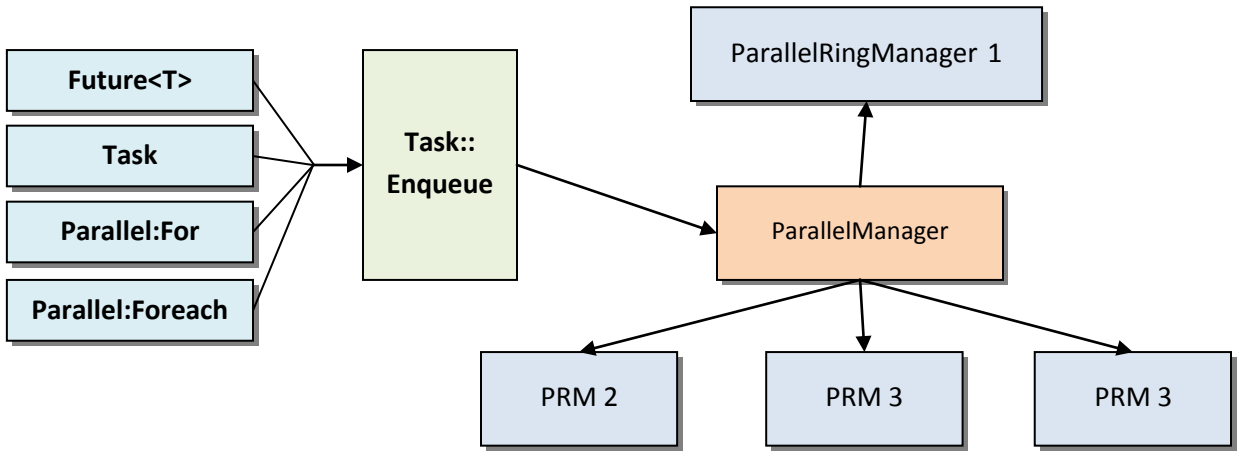


Figure 26: Tasks are allocated onto the various processor-cores

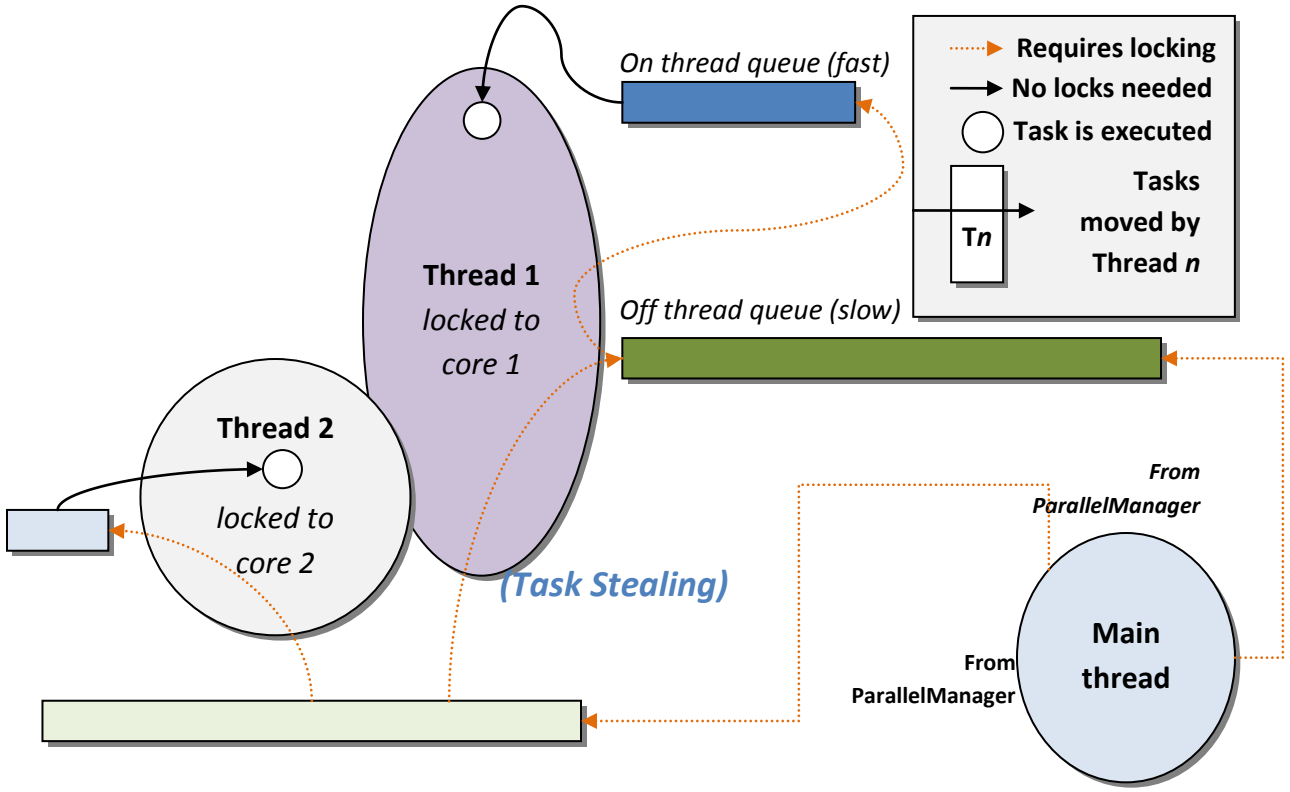


Figure 27: Movement of tasks between cores, including locking detail

Task scheduling

Two classes exist for the sole purpose of scheduling tasks – ParallelManager is the singleton static controller for everything task-related and the ParallelRingManager class which encapsulates all of the objects needed for each core.

During program instantiation the ParallelManager class constructs a number of ParallelRingManagers which each set up two queues and a processor-locked thread on which to run the tasks. As the program progresses, a number of different tasks will be constructed, and as each one is created they automatically forward themselves to the ParallelManager to be queued. The ParallelManager attempts to maintain a reasonable spread of tasks between cores by considering the ParallelRingManagers as a ring and always passing the next task to the next element in the ring.

The ParallelRingManagers set up a processor-locked thread (the "task execution thread") as part of their instantiation whose purpose is to dequeue tasks and to run them. Because the off-thread queue is accessed by more than one thread, all accesses to it, either for enqueuing or dequeuing must be guarded by an exclusive lock. One of the problems that showed up during testing is that this locking procedure is expensive and actually reduces the effectivity of the parallelization so as to make parallelism redundant. It is for this reason that the ParallelRingManagers maintain an additional queue – the on-thread queue – which is owned exclusively by the task execution thread. When this on-thread queue runs out of tasks to execute, the task execution thread obtains an exclusive lock on the off-thread queue and dequeues a collection of tasks from the off-thread-queue, moves them onto the on-thread queue and releases the exclusive lock. By doing this we eliminate the need for an exclusive lock when dequeuing elements from the on-thread queue as well as reducing the total number of locks acquired by "bunching" locks together as we dequeue a collection of objects at once.

Despite the best efforts of the ParallelManager to keep tasks evenly spread, the natural variation in task execution length means that over time the task queues of the various ParallelRingManagers will get out of sync. At these times it is possible for one ParallelRingManager and thus one core to run out of tasks and idle while another ParallelRingManager has a number of tasks still waiting to run. For this reason we implement a task-stealing mechanism, whereby the idling task execution thread will inspect the other executing threads in turn to find any thread with a large number of elements waiting on its off-thread queue (this inspection does not require a lock). If another off-thread queue *does* contain an abundance of tasks, the task execution thread obtains an exclusive lock on that thread's off thread queue and migrates half of the queue to its own off-thread queue. The thread now has elements on its off thread queue, and can obtain a new lock, move some tasks to its on thread queue, release the lock and begin work.

The worst-case idling delay is proportional to the product of the on-thread queue length and the mean task execution. The worst-case locking delay is inversely proportional to the product of the on-thread-queue length and the mean task execution. The frequency of thread stealing is proportional to the square of the task-time variance. Note that when there is a high enough throughputs of tasks all cores will always be kept busy. Over a series of tests it was determined that for an average program the

optimal number of tasks to exist on the on-thread queue was between 8 and 15. In the *Xinaos* parallel runtime library the on-thread queue length is 12.

The Parallel static class

Although writing code in terms of Tasks is possible, Tasks are typically created by other parts of the parallel runtime, such as the parallel static class and the out-of-order Future<T> class.

The parallel static class effectively breaks up loops of the form

```
// natural for loop
for(int i=START; i<END; i+= INCR) {
  code(i);
}
// backwards for loop
for(int i=START; i>END; i-= DECR) {
  code(i);
}
// foreach loop
foreach(var obj in collection) {
  code(obj);
}
```

Example 8: loop variants

Into their parallel equvilent:

```
// parallel natural for loop
Parallel.For(START, END, INCR, (i)=> {
  code(i);
});
// parallel backwards for loop
Parallel.For(START, END, -DECR, (i)=> {
  code(i);
});
// parallel foreach loop
Parallel.Foreach(collection, (obj) => {
  code(obj);
});
```

Example 9: parallel equivalent loop variants using C# lambda syntax

Although a naive approach would be to break up each loop iteration into an individual Task, in practice grouping many iterations together reduces the total overhead and thus improves performance. The *Xinaos* parallel library breaks the loops into blocks whose size is determined by the number of iterations in the loop and the number of processors available on which code can be run.

- If there is just one processor available (the machine is a single-core), or the number of iterations is less than 10, then parallelism is ignored, and the code is just executed synchronously.
- If the code has more than 10 iterations, then the iteration space is broken into blocks of (iterations to be performed / (processors * 10)) or 10 – whichever is the larger.
- If the code is over a foreach and the collection is an enumerator whose size cannot be computed without iterating through the enumerator (e.g. a foreach over a filtered list, over all lines in a file, all tokens in a

stream etc) then the enumerator is enumerated into buckets of length 10; each bucket is then iterated over as a block by a single task.

The parallel loop manager does not return until the loop has completed (and thus all created tasks for that loop finished successfully or threw an exception). The calling thread does not idle in this time – it is also utilized to execute the tasks in the loop.

One problem with performing a parallel loop out-of-order is that even when individual-iterations are independent, the potential for errors to occur is still a problem since these should also occur deterministically. It is at this point that manual and compiler-generated parallel loops differ. Where the programmer has explicitly asked for parallelism to be performed, the runtime merely halts or cancels all tasks yet to be issued for the loop, and returns a ParallelException which aggregates any exceptions that occurred during the loop.

Where the parallelism has been generated by the compiler, this is not an option – since the optimization must be transparent to the programmer and a failure on the n^{th} iteration should not occur before the $(n-a)^{th}$ iteration[22], and nor should the exception change from some specific exception to an aggregate exception without the programmer's explicit approval. For this reason, exceptions which are thrown from inside compiler-generated parallel loops do not cancel all running tasks – merely all tasks of a higher iteration value than the task on which the exception was thrown. When all lower iterations have completed, they are all inspected to find the lowest iteration which threw an exception, which is then rethrown by the runtime. This fulfills the semantic that optimizations should not change the in-order nature of loops, even up to exceptions. Additionally since compiler-generated parallelism is only ever performed where iterations can be performed out of order, the iterations do not cause side-effects, and the lowest iteration to throw an exception would have thrown an exception on a non-parallel execution of the loop, and thus rethrowing that exception is a valid optimization to perform.

Out of order parallelism (non-loop based)

Another important task of the Parallel Runtime library is to schedule and allow out-of-order work to be executed in the background. To see the importance of this as a concept, consider the following recursive code to compute the n^{th} value of the Fibonacci sequence:

```
int fib(int n){
  if(n == 1 || n == 2) return 1;
  return fib(n - 1) + fib(n - 2);
}
```

Example 10

Although there are no loops which can expose parallelism in this example, there is clearly an example of potential parallelism implicit in the code – both operands to the addition can be computed independently, and thus can be done in parallel.

[22] $\forall a > 0$

This is the reason for the introduction of the Future<T> generic class which takes as input a generator function which yields a value of type T. The Future<T> has only one property – Future<T>.Value which yields the result of the generator (or throws the exception yielded by the generator if an error occurs).

Future<T> operates by scheduling a task which runs the generator and puts the result into a field owned by the Future<T>. When Future<T>.Value is accessed, the Future<T> places a *demand* request on the Task, forcing the Task to be completed before continuing, either by checking to make sure that the Task has completed, failed or by running the Task synchronously on the calling thread. If the Task failed, the Future<T>.Value immediately throws the corresponding exception. If not, and the Task completed successfully then the value of the field which was set by the Task is returned.

This allows some expressions which need to be computed to be migrated earlier in the program code by the optimizer in order to achieve higher levels of parallelism, and is directly analogous to out-of-order loads and operations in out-of-order processors.

Let us look once again at the code in Example 10, and consider the call-graph of the function when called with some large value. Each call generates two subsequent calls to the same method, both with smaller arguments, both taking some reasonably long period of time to compute before returning. Consider also the function when called with small arguments – the function here obtains both operands of the addition reasonably quickly, and thus is able to return quickly.

Consider now what would happen if we offload the computation of the second operand to a secondary processor via the Future<T> construct. In this case we would find that the parallel manager and the cores quickly become saturated with work to be performed from the high levels of the call stack where the tasks are long and the parallelism well worth the effort. In contrast, further down the call stack where the overhead of performing operations in parallel dominates over the actual running time of the task, the parallel queues are already full, and thus the tasks do not run on separate cores but rather sit in the parallel queue and are demanded almost immediately, thus running synchronously on the originating core.

We find therefore that although some overhead is incurred by creating and allocating the Future<T>, Task and the delegate that forms the generator for the Future<T>, the spread of load is actually near optimal when the second operand is offloaded via the Future<T> construct.

The *Xinaos* compiler will soon be capable of detecting such parallel opportunities and converting them automatically in the compiler to the form shown in Example 11. The algorithm for detecting and implementing these optimizations is discussed in more detail in the *Detecting and implementing non-loop based parallelism* chapter on page 57.

```
int parallelfib(int n){
  if(n == 1 || n == 2) return 1;
  Future<int> _o2 = new Future<int>( () => fib(n - 2) );
  return fib(n - 1) + _o2.Value;
}
```

Example 11

The Parallel library as a component of the *Xinaos* compiler

It is easy to show hand-picked benchmarks which demonstrate the effectiveness of new libraries, compilers or algorithms; however the problem with such benchmarks is that by carefully choosing the benchmark it is often difficult to assess the real-world usefulness of the product. Indeed benchmarks can typically be chosen and tuned in such a way to show truly spectacular and completely unrealistic figures if the author is unscrupulous enough, or does not understand the similarity of the benchmarks to real-world usage of the product.

For this reason the Parallel library is used in the *Xinaos* compiler as a real world example of it in action. Most of the parallelism is in the assembly-builder and CodeProvider which parallelize the compiler assembly phases and the file-to-syntax tree generations respectively.

It should be noted that the XIL files output by the compiler do differ from each other when parallelism is enabled; however this is merely a different ordering in the type tables and method tables of the methods and types with respect to each other due to the relative order in which files are processed. It is important to note that no code (even at runtime e.g. for static constructor order) relies on the relative ordering of types or methods – this is a completely superficial alteration.

The parallelism can be disabled for debug or benchmarking purposes by adding the NO_PARALLEL symbol to the Xinaos.Compilers project.

The syntactic similarity between a parallel for statement and an actual for statement means that code using parallel equivalents of loop constructs are no less readable than their serial counterparts.

Testing the parallel library

The Parallel library is a substantial library in its own right, and used its own testsuite to guarrantee both correctness and speed. Compared to a syncronous execution of 16x16 matrix multiplications, tree searches and other tests, the data for which being randomly generated (and being checked synchronously), the parallel library was correct on all tests. On a dual-core machine the parallel implementation averaged 65% the speed of the serial implementation of 16x16 matrix multiplication and on a quad-core machine the parallel implementation averaged 48% the speed of the serial implementation on the same machine.

Conclusions of the *Xinaos* compiler

Ray tracing is a great, but computationally expensive way to produce photo-realistic renderings. Thankfully each ray can be cast independent of the other rays, and consequently ray tracing is a good example to try the *Xinaos* compiler / framework on.

The following ray trace is based on Luke Hoban's ray-tracer for .NET, which one of the same benchmark used by the Microsoft Parallel FX CTP team.

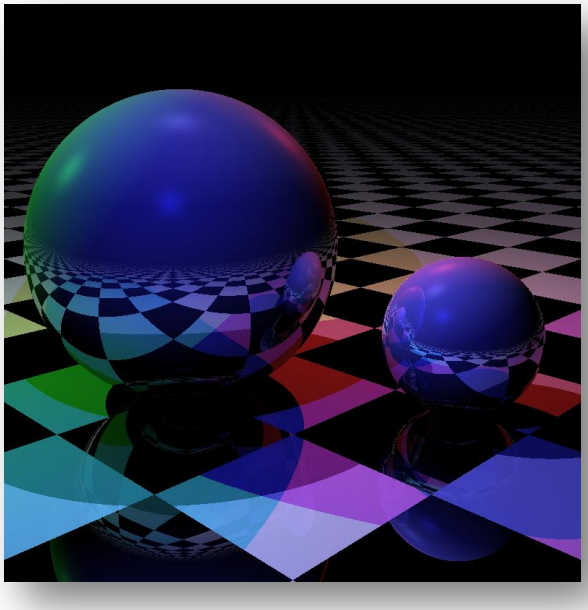


Figure 28: Result of a raytrace using the *Xinaos* compiler/runtime

The ray tracer uses large amounts of recursion and floating point; and the current slow implementation of floating point in the *Xinaos* compiler means that the parallel *Xinaos* framework only just outperforms the .NET framework with all optimizations on[23], although it does so only by a margin of 10% or so.

Ray-tracing an image 800x800 on a modified version of the original ray-tracer (so that no virtual calls are performed, and re-implementing System.Bitmap in the target language with copious amounts of PInvokes so the result can be saved), the results are as follows:

	Mean Time[24]	*Improvement over no optimisations*
No optimisations	*103.8s*	-
+ Linear optimizations	*97.6s*	*5.9% speedup*
+ Control flow optimizations	*91.6s*	*11% speedup*
+ Inlining	*89.1s*	*14% speedup*
+ Parallel	*54.2s*	*47% speedup*

[23] .NET mean time for 800x800 was 61.2s. When parallelized, the .NET mean time was 30.2s (49.3% speedup)
[24] Mean over 20 iterations on AMD Turion Dual-Core RM-70 2.00GHz with 2.00GB RAM

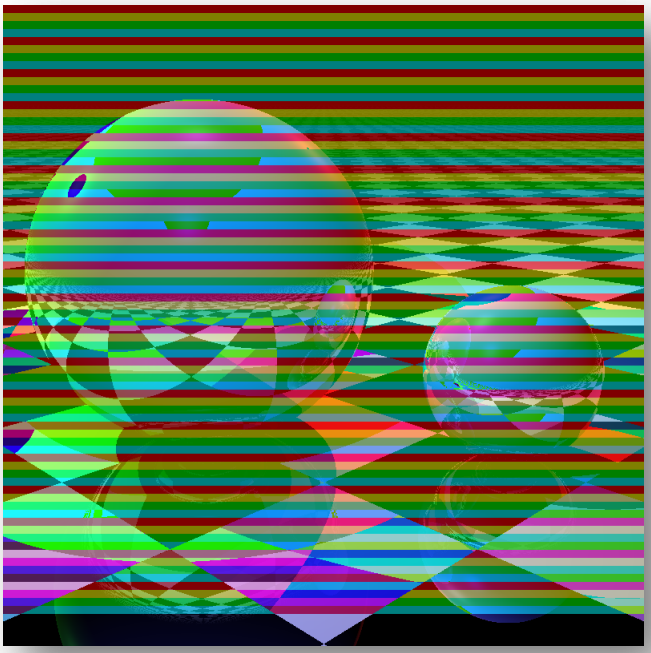


Figure 29: Distribution of work over the ray trace.

The different colored bands correspond to different threads, with the last (uncolored) block being completed synchronously by the main thread.

Appendix A: Terms and definitions

Abstract method

A method defined on an abstract type which has no method implementation. The method implementation is defined on an inheriting type.

Abstract type

A type from which other types can inherit, but from which no object can be instantiated

ANSI character

A character from an 8-bit character set whose first 128 values correspond exactly to those specified in ISO/IEC 10646

ANSI string

A string of ANSI characters, followed by a nul character

Attribute

A characteristic of a type, member or field which contains descriptive information to be conveyed either to the compiler, runtime environment or generated code at runtime via reflection

Assembly; XIL

A collection of XIL types and methods which form a complete program which can be later executed by the XIL runtime

Boxing

A conversion up the inheritance tree of some type – in particular when casting some value type to the type System.Object by copying the value to the heap

Constructor

A method called automatically upon creation of an object instance, typically in order to initialize the object fields of the type before the object's first use.

Contract

See **method contract**

Dead code

Code which cannot be reached by any path of execution through the method

Dead object

An object whose contents are no longer accessible to the program through any direct pointer or managed reference, and whose resources and memory can be reclaimed by the garbage collector without impacting on the correctness of the program

Deadspace

A region of memory on a garbage collected heap between live objects which is not currently in use.

Deadspace directory

The management list of deadspace on the current heap.

Delegate

A managed reference which encapsulates a method invocation. Delegates are strongly-typed, and can only hold method signatures of methods which adhere to the method contract of the delegate, which includes return type, number and type of parameters and zero-or-more method contracts.

Destructor

A method which is called deterministically upon the destruction of an object instance. In C# and the *Xinaos* runtime this is approximated by the Dispose/Finalize pattern (see chapter on garbage collection)

Direct pointer

A pointer which is not obtained via pointer-arithmetic, other than a bounds-checked array access

DWORD

An unsigned integer which is exactly four bytes long

End-user

The person or computer which runs the compiled XIL code

***Ensures* contract**

A property of the returned value of the method which is always true if the method returns normally, regardless of execution path through the method

Expression

A self-contained piece of code which yields a value of an instantiable type, for example a literal, constant, arithmetic result or non-void method call

Field

A class member that designates a typed memory location inside a managed object

Finalizer

An instance method defined on a reference type to be called where an instance of the class is no longer reachable by the program. The finalizer is invoked by the garbage collector when the instance goes out of scope. The finalizer for an instance can be manually enabled or disabled via the GC::ReregisterForFinalize and GC::SuppressFinalize methods.

Garbage collection

The process by which memory for managed data is collected and released automatically

Garbage collector

The part of the runtime program which is responsible for the management, allocation and de-allocation of managed objects.

Generic argument

The actual type used to instantiate a generic type or generic method, for example in List<int>, int is the generic argument which takes the place of the generic parameter T in the class definition List<T>.

Generic parameter

The placeholder in a generic class definition which is qualified by a generic argument, for example in List<T>, T is a generic parameter.

Heap; garbage collected

A region of contiguous memory held and managed by the garbage collector which is sub-allocated into the various managed objects as they are allocated.

Heap migration

The process by which live objects are transferred from one heap to another. This process additionally removes any fragmentation on the heap that may exist and additionally realigns pointers on the stack, in managed fields and in static variables to point to their newly relocated position. This process is performed by the garbage collector.

Inheritance; type

The mechanism by which new types are defined as extensions of pre-existing types. Child types inherit the types and fields of their parent object and can optionally overwrite their parent's methods. Child-types may always be boxed to their parent type, and their parent type may always be unboxed to the child type.

Instance method

A method defined as being associated with a particular type instance and thus requiring a non-null instance of the type to correctly operate.

Invariant; loop

An expression whose value is independent of loop iteration, and can be safely computed and cached prior to loop entry

Invariant; class

A property on a type which holds at the start and end of every public method for any instance of that type

L-Value

Any expression which can be treated as a storage location, and thus whose address can be taken and into whose value an assignment can be made

Live object

An object which is not dead

Managed code

Code written in the XIL instruction set, which is executed by the execution environment and whose memory is automatically managed by the garbage collector. Managed code accesses objects through managed references rather than pointers, and does not engage in pointer arithmetic.

Member

The fields, array elements, methods and properties of a type

Method

A member that describes an operation to be performed on values of an exact type

Method contract

A *requires* or an *ensures* contract on a method

Native code

Code which is not managed

Nul character

The first entry in the ASCII table of values

Null

See Null reference

Null reference

A reference object which has no value

Object

An instance of a reference type or boxed value type. An object is self-typing; i.e. its type is explicitly stored in its representation and a unique object identifier. An object additionally has a dedicated region of memory for storing its representation on the heap made up of contiguous slots defined by the instance fields of the object type.

Pointer arithmetic

The process by which a new pointer is created by casting an integer to a pointer-type, for example taking some pointer and adding an integer value to it, and treating the result as a pointer to memory

Programming user

A user who writes code which is translated by the *Xinaos* compiler into an XIL file which can be run by an end-user at a later stage

Property

A member that defines a named value and the methods that are used to access or set that value

QWORD

An unsigned integer which is exactly 8 bytes long

Resurrection; Object

An object which, having gone out of scope and being registered for finalization, becomes live again through interacting with the live-set during its finalizer – usually by adding a reference to itself through a static live variable.

***Requires* contract**

A property on one-or-more parameters to a method which is always true when the method is called

Runtime checked

A property whose validity cannot be determined by inspection and whose validity must be checked at the point where the statement is used by inserting appropriate code into the method at the point where the property must be valid

Runtime Environment

The system which implements and runs the XIL code. The Execution environment is responsible for loading and running programs written in XIL, as well as providing the services needed to execute the garbage collector, inbuilt methods and just-in-time compiler

Stack; expression

1. The architecture-independent representation of currently-held values in the *Xinaos* virtual machine, onto which values are loaded and stored by the various XIL instructions during program execution
2. The series of intermediary expression arguments in Djikstra's shunting-yard algorithm for expression evaluation

Stack pointer

The value held in the stack register (ESP register on x86 machines). The stack holds some fast-access program data such as register-overflow values, value-types and references to the heap.

Statement

A self-contained instruction of code which directs the computer to perform some action, such as setting a variable, creating an object or calling a method. Statements have no "result", and thus must perform some visible action.

Static method

A method which is not an instance method

Statically checked (statement)

A statement whose validity can be fully determined by inspection, and thus whose corresponding code does not need to be run for the statement to be checked

Un-boxing

The conversion of an object down the inheritance tree from a type to a child of that type – in particular when converting between System.Object and some value type by copying the data from the heap-stored value reference onto a value-type location held on the stack.

Unmanaged code

Code which is not managed

WORD

An unsigned integer which is exactly two bytes long

Working set (of objects)

The collection of all objects which are reachable at any given time. This is exactly equivalent to the set of live objects.

Value type

A type defined by a specific value rather than a specific instance. Value types assigned into value type slots are copied by value rather than by reference. Value types can be boxed into reference types. Value types do not (by default) exist on the heap, and are automatically cleaned up when leaving method scope, thus reducing the strain on the garbage collector.

Verification; XIL

The checking of XIL code and the XIL metadata to ensure that the program description is valid

Verification; method contract

The checking of an XIL method code to ensure that the *ensures* contract of the method and any *requires* contracts of any method calls inside the method are upheld.

Virtual method

A method whose invocation depends on the object instance on which the method is being invoked

Virtual Machine; *Xinaos*

See Runtime Environment

Appendix B: Abbreviations and acronyms

CIL

Common Intermediary Language

The intermediary language used by the Microsoft .NET™ Framework

GC

Garbage collector

JIT

Just-in-time

Usually refers to the Just-in-time compiler, which takes code at runtime written in an intermediary language and generates machine code and immediately executes it

JVM

Java-virtual machine

MSIL

Microsoft Intermediary Language™. Also Common Intermediate Language (CIL)

See **CIL**

XIL

Xinaos intermediary language

The intermediary language used by the *Xinaos* compiler and interpreted by the just-in-time compiler

Appendix C – Example Heap Migration

The following is an example (from the raytracer in Chapter 9) of migrating a heap to a larger heap, and is taken from a real world example of a ray tracer. Because a heap-migration starts with a full collect, some objects towards the start of the heap died immediately prior to the heap dump being called. The GC's options are configured in this example to clear dead objects immediately with the CDCDCDCD bit pattern so as to make their demise more obvious. This is an important point to make as the deadspace towards the start of the heap is therefore actually newly acquired, and not evidence of a poor allocation policy of inherent fragmentation of the heap.

Although going through heap dumps is a laborious process (and I do not expect the reader to go over them in any great detail) the heap dumps shown over the next few pages demonstrate some of the key features of the migration.

The first page shows the heap before the migration. The second page shows the migration pairings made by the GC, including the type of the object at each location, and where the object went from and to in the old and new heaps respectively.

The third page shows the heap immediately after the migration.

The heap dumps are decorated with additional information to increase legibility, and it can therefore be more easily verified that the heap migration is correct.

The migration routine is complete – that is no data that is not accessible is copied over (except where policies such as objects reachable from finalizer objects demand that they are copied as a special case). The migration routine is also correct – that is no data that is accessible is accidentally trashed. This can be ascertained by performing the GC::Collect routine immediately after the migration has completed. By verifying that the number of live objects remains the same, we can have confidence that links are not destroyed by this method.

All objects are 16-byte aligned to reduce fragmentation.

There are a few things that the reader should check to confirm to themselves that the migration routine is correct.

Key to heap dumps:

```
ADDRESS         DWORD DWORD DWORD DWORD

>DWORD<   indicates start of object
[DWORD]   indicates a dword that points to the start of an object on the heap
?DWORD?   indicates a dword that looks like it points to an object on
          the heap, but does not point to the start of an object (i.e.
          a misalignment)
!DWORD!   indicates a dword that points to the swap heap (i.e. a pointer
          wasn't carried over during a heap migration properly)
```

```
Heap Dump: (live objects = 29, heap base = 008F0000)

0x008F0000:     >00000064<   00000064   [008F0010]   CDCDCDCD
0x008F0010:    >[008F0100]< [008F00E0]  [008F00D0]   CDCDCDCD
0x008F0020:     >40400000<   40000000    40800000    CDCDCDCD
0x008F0030:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0040:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0050:     >00000000<   00000000    00000000    CDCDCDCD
0x008F0060:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0070:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0080:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0090:     >00000000<   00000000    00000000    CDCDCDCD
0x008F00A0:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F00B0:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F00C0:     >00000000<   00000000    00000000    CDCDCDCD
0x008F00D0:    >[008F0020]< [008F0050]  [008F00C0]  [008F0090]
0x008F00E0:     >00000004<  [008F0110]  [008F0140]  [008F0170]
0x008F00F0:     [008F01A0]   CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0100:     >00000001<  [008F01E0]   CDCDCDCD    CDCDCDCD
0x008F0110:    >[008F0130]< [008F0120]   CDCDCDCD    CDCDCDCD
0x008F0120:     >3EFAE148<   3F333333    3F333333    CDCDCDCD
0x008F0130:     >C0000000<   40200000    00000000    CDCDCDCD
0x008F0140:    >[008F0160]< [008F0150]   CDCDCDCD    CDCDCDCD
0x008F0150:     >3F333333<   3F333333    3EFAE148    CDCDCDCD
0x008F0160:     >3FC00000<   40200000    3FC00000    CDCDCDCD
0x008F0170:    >[008F0180]< [008F0190]   CDCDCDCD    CDCDCDCD
0x008F0180:     >3FC00000<   40200000    BFC00000    CDCDCDCD
0x008F0190:     >3F333333<   3EFAE148    3F35C28F    CDCDCDCD
0x008F01A0:    >[008F01B0]< [008F01C0]   CDCDCDCD    CDCDCDCD
0x008F01B0:     >00000000<   40600000    00000000    CDCDCDCD
0x008F01C0:     >3E570A3D<   3E570A3D    3EB33333    CDCDCDCD
0x008F01D0:     >00000000<   3F800000    00000000    CDCDCDCD
0x008F01E0:    >[008F01D0]<  00000000    CDCDCDCD    CDCDCDCD
0x008F01F0:    >[008F0200]<  018A2270    CDCDCDCD    CDCDCDCD
0x008F0200:     >95E64F3B<   CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0210:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0220:      CDCDCDCD    CDCDCDCD    CDCDCDCD    CDCDCDCD
0x008F0230:     >00259680<   0025968C    00000001    CDCDCDCD
0x008F0240:     >00259680<   00259698    00000001    CDCDCDCD
 ... (40004 bytes) ...

Stack:
(ESP:0x001BF334): [0x008F0000]
(ESP:0x001BF33C): [0x008F0000]
(ESP:0x001BF348): [0x008FFFC0]
(ESP:0x001BF354): [0x008F0250]
(ESP:0x001BF358): [0x008F01F0]
(ESP:0x001BF398): [0x008F0000]

Static variables:
0x00945FA0 (static #0): [0x008F0230]
0x0094BC00 (static #1): [0x008F0240]
```

```
Performing a full heap move
Expanding heap (heap was 65536 bytes, now 131072 bytes)

008F0000 -> 00980000 (System.Program.RayTracer)
008F0010 -> 00980010 (System.Program.Scene)
008F0020 -> 00980020 (System.Program.Vector)
008F0050 -> 00980030 (System.Program.Vector)
008F0090 -> 00980040 (System.Program.Vector)
008F00C0 -> 00980050 (System.Program.Vector)
008F00D0 -> 00980060 (System.Program.Camera)
008F00E0 -> 00980070 (System.Program.Light[4])
008F0100 -> 00980090 (System.Program.Plane[1])
008F0110 -> 009800A0 (System.Program.Light)
008F0120 -> 009800B0 (System.Drawing.ColorF)
008F0130 -> 009800C0 (System.Program.Vector)
008F0140 -> 009800D0 (System.Program.Light)
008F0150 -> 009800E0 (System.Drawing.ColorF)
008F0160 -> 009800F0 (System.Program.Vector)
008F0170 -> 00980100 (System.Program.Light)
008F0180 -> 00980110 (System.Program.Vector)
008F0190 -> 00980120 (System.Drawing.ColorF)
008F01A0 -> 00980130 (System.Program.Light)
008F01B0 -> 00980140 (System.Program.Vector)
008F01C0 -> 00980150 (System.Drawing.ColorF)
008F01D0 -> 00980160 (System.Program.Vector)
008F01E0 -> 00980170 (System.Program.Plane)
008F01F0 -> 00980180 (System.Bitmap)
008F0200 -> 00980190 (System.GDIPlusHandle)
008F0230 -> 009801A0 (System.Runtime.CompilerServices.DllImportAttribute)
008F0240 -> 009801B0 (System.Runtime.CompilerServices.DllImportAttribute)
008F0250 -> 009801C0 (System.Drawing.ColorF[10000])
008FFFC0 -> 00989E10 (System.Drawing.ColorF)
```

```
Heap Dump: (live objects = 29, heap base = 00980000)

0x00980000:     >00000064<   00000064   [00980010]   CDCDCDCD
0x00980010:    >[00980090]< [00980070]  [00980060]   CDCDCDCD
0x00980020:     >40400000<   40000000    40800000    CDCDCDCD
0x00980030:     >00000000<   00000000    00000000    CDCDCDCD
0x00980040:     >00000000<   00000000    00000000    CDCDCDCD
0x00980050:     >00000000<   00000000    00000000    CDCDCDCD
0x00980060:    >[00980020]< [00980030]  [00980050]  [00980040]
0x00980070:     >00000004<  [009800A0]  [009800D0]  [00980100]
0x00980080:     [00980130]   CDCDCDCD    CDCDCDCD    CDCDCDCD
0x00980090:     >00000001<  [00980170]   CDCDCDCD    CDCDCDCD
0x009800A0:    >[009800C0]< [009800B0]   CDCDCDCD    CDCDCDCD
0x009800B0:     >3EFAE148<   3F333333    3F333333    CDCDCDCD
0x009800C0:     >C0000000<   40200000    00000000    CDCDCDCD
0x009800D0:    >[009800F0]< [009800E0]   CDCDCDCD    CDCDCDCD
0x009800E0:     >3F333333<   3F333333    3EFAE148    CDCDCDCD
0x009800F0:     >3FC00000<   40200000    3FC00000    CDCDCDCD
0x00980100:    >[00980110]< [00980120]   CDCDCDCD    CDCDCDCD
0x00980110:     >3FC00000<   40200000    BFC00000    CDCDCDCD
0x00980120:     >3F333333<   3EFAE148    3F35C28F    CDCDCDCD
0x00980130:    >[00980140]< [00980150]   CDCDCDCD    CDCDCDCD
0x00980140:     >00000000<   40600000    00000000    CDCDCDCD
0x00980150:     >3E570A3D<   3E570A3D    3EB33333    CDCDCDCD
0x00980160:     >00000000<   3F800000    00000000    CDCDCDCD
0x00980170:    >[00980160]<  00000000    CDCDCDCD    CDCDCDCD
0x00980180:    >[00980190]<  018A2270    CDCDCDCD    CDCDCDCD
0x00980190:     >95E64F3B<   CDCDCDCD    CDCDCDCD    CDCDCDCD
0x009801A0:     >00259680<   0025968C    00000001    CDCDCDCD
0x009801B0:     >00259680<   00259698    00000001    CDCDCDCD
 ... (40004 bytes) ...

Stack:
(ESP:0x001BF334): [0x00980000]
(ESP:0x001BF33C): [0x00980000]
(ESP:0x001BF348): [0x00989E10]
(ESP:0x001BF354): [0x009801C0]
(ESP:0x001BF358): [0x00980180]
(ESP:0x001BF398): [0x00980000]

Statics:
0x00945FA0 (static #0): [0x009801A0]
0x0094BC00 (static #1): [0x009801B0]
```

Appendix D: *XIL* File Format

XIL File Format

In the XIL file format, data is stored in two kinds of structure: tables (arrays of data) and indexed heaps.

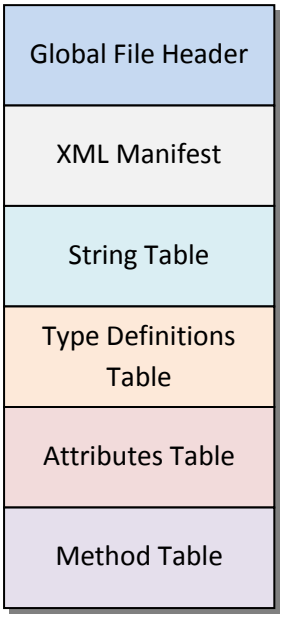


Figure 30: XIL File Format

Global File Header

The Global File Header contains global information about the program used by the *Xinaos* runtime to load the file. The table is exactly 256 bytes long and always sits at the beginning of the file.

```
struct XILGlobalFileHeader {
    // XIL info
    DWORD XIL_magic_number;
    WORD  XIL_version[4];
    DWORD XIL_flags[2];

    // Program info
    WORD  program_version[4];

    // Offsets into the file where the directories start
    DWORD offset_xml_manifest;
    DWORD offset_stringtable;
    DWORD offset_typedefs;
    DWORD offset_customattrs;
    DWORD offset_methoddefs;
    DWORD offset_eof;

    // the method which is the program starting point.
    DWORD program_entry;

    BYTE  Reserved[200];
};
```

XIL Magic Number

Must be 0x58494c30 (ASCII 'XIL0')

Rationale: By having a unique magic number at the start of every file, the incidence of misnamed files masquerading as XIL executables is reduced, allowing a quick termination if the XIL runtime starts up with a clearly non-XIL executable as input.

XIL Version

This field contains four WORDs which describe the minimum runtime version expected by the compiler and the XIL program. The words correspond to the values for major number, minor number, revision number, and build number respectively.

The runtime ***must*** stop with an error if the version number is greater than its own (parsed lexically). The runtime ***may*** stop with an error if the version number is not equal to its own.

Rationale: In future versions of the XIL runtime, different structures, instruction sets or even global layouts may exist in the file which would cause confusion or incorrect behavior in older runtimes. On the other hand, newer runtimes may be designed to be backwards binary-compatible with an older runtime file format, and thus newer runtimes may be able to continue directly or emulate the runtime for an older file format, and thus may choose to run the program even if the version number is not equal to its own.

XIL Flags

64 flags Reserved for future use

Program Version
Four WORDS which describe the version of the XIL program. This field is largely ignored by the runtime itself, although in future it may be used when linking external modules (which are not currently supported by the *Xinaos* framework).

Offset XML Manifest
The zero-indexed byte position of the file where the XML manifest begins

Offset String Table
The zero-indexed byte position of the file where the string table begins

Offset Type Table
The zero-indexed byte position of the file where the type table begins

Offset Attributes Table
The zero-indexed byte position of the file where the attributes table begins

Offset Method Definition table
The zero-indexed byte position of the file where the method table begins

Offset EOF
The total number of bytes in the file

File-size
The length in bytes of the file

Rationale: This field helps check that the file is complete and has not been truncated for any reason.

Reserved
A series of bytes that are currently unused, but which may be used in future versions of the XIL file format. These values should be set to 0.

XML Manifest

This section contains an XML manifest for the program which can contain additional meta-information about the program in an extendible way. The XML manifest **must** be ASCII-encoded and nul-terminated (and contain no other nul-characters).

```
<xilprogram>
  <author>Matt Tait</author>
  <name>Example program</name>
  <uac>RunAsAdministrator</uac>
</xilprogram>
```

Code Fragment 1: Example XML manifest

The XML manifest is not currently used in the *Xinaos* runtime, but in future will be used to articulate strong permissions demanded by the operating system (e.g. permissions to write to specific registry keys or to alter system settings) as well as some finer-grained permissions which would be handled by the *Xinaos* framework rather than the operating system – for instance IO permissions to certain folders, to access certain URLs or to run unsafe code.

Rationale:

XML is chosen as the medium of choice here because it is easy to parse from outside of the *Xinaos* framework, for instance by supporting programs or the operating system which may wish to inspect the file to check what potential side-effects may be caused by executing the file.

It is possible, for instance, that if the *Xinaos* framework were protected from mutation (either by being an integral part of the operating system or by appropriate permissions) then the XML specification would become a valid and simple way of checking and assigning permissions to programs on a user-by-user basis, and to help users determine what potential damage a program (e.g. a program downloaded from the internet or from an email attachment) may cause before running it.

This should in principle avoid the *Xinaos* framework file format ever being used to launch virus attacks, since viruses would need to declare what permissions they need to operate, and thus users would be presented with enough information prior to running the program to determine that running the file may not be safe, without jeopardizing the potential for the *Xinaos* framework to execute potentially unsafe code where such execution is warranted, for example in virus-checkers or operating system addons where the user is aware of the risks.

This combination of fine-grained permissions and user control, means that programs written for the *Xinaos* framework are both more secure and more informative for end-users than their .NET, Java or native-compiled counterparts without sacrificing any of the functionality that these other frameworks provide, and would hopefully give programs a more transparent effectivity for end-users.

End rationale.

String Table

The string table contains all of the literal data to be used by the program. Despite the name, the string table can additionally contain resources to be used by the file. Strings in the string table are Unicode-encoded, although other data may choose other format encodings.

The string table also contains the names of methods and types. The XIL Type and Method data structures contain offsets into the string table indexes for their names rather than storing their name explicitly.

There is no ordering imposed on the string table. The string table contains only constant data which is never mutated by the program, and thus it is a valid (and indeed encouraged) optimization to pool identical strings or data blobs together.

Some readers may also notice the potential to pool not only entire strings, but also string parts – or even pool across string literals by introducing dummy records which compensate for the lack of *length* field in the string table. Although such optimizations may be tempting, the action of the *Xinaos* runtime on finding such records is not well defined, and thus such optimizations should be avoided.

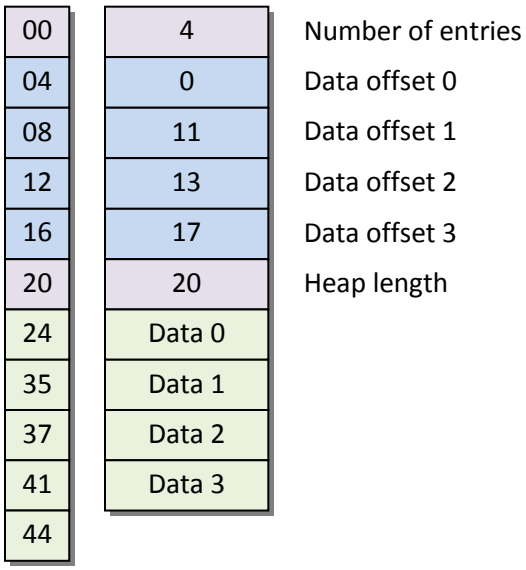


Figure 31: XIL File Heap/directory layout

The string table begins with a DWORD which is the total number of entries in the string table (*num_entries)*. It is then followed by *num_entries* DWORDs which represent the index into the heap of the data corresponding to that entry. Since the table is ordered, the length of the entry can be taken by subtracting the following index's offset from the current one, and for this reason the string table index is followed by a single DWORD which points to the end of the heap.

Rationale: This allows the length of the final element in the string table to be computed in the same way – the heap length minus the heap offset of the final string is the length of the final string.

The string table then contains all of the data entries next to each other. Data and strings are not nul-terminated in this list.

```
/*                                                                          */
/* Getting entries out of heaps stored in the XIL File specification format */
/*                                                                          */

int getHeapEntries(DWORD* table){
  return *table;
}

char* getHeapEntry(int n, DWORD* table){
  DWORD num_entries = *table;
  table++;
  assert(n >= 0 && n < num_entries);

  DWORD offset = table[n];

  // skip over the final entry, which is the heap length
  BYTE* heap = &table[num_entries + 1]

  // + 1 because of the num_entries at the beginning.
  return heap[offset];
}

int getEntryLength(int n, DWORD* table){
  DWORD num_entries = *table;
  table++;
  assert(n >= 0 && n < num_entries);

  return table[n+1] - table[n];
}
```

Type definitions table

The type definitions table contains the definition of every type in the program. The header is 64 bytes long.

```
struct XILTypeDefinitionHeader {
   // type indexes
    DWORD inherits;

    // type meta information
    DWORD name;
    DWORD flags[2];

    DWORD static_constructor_method_index;
    DWORD finalizer_index;

    // field and method information
    DWORD num_instance_fields;
    DWORD num_static_fields;

    WORD  primitiveCode;

    BYTE reserved[30];

    XILFieldHeader instanceFields[num_instance_fields];
    XILFieldHeader staticFields[num_static_fields];
};
```

Inherits

An offset into the type definitions table which contains the type from which this type inherits. This is ignored only on the class System.Object. There should be no cyclical inheritances.

Name

An offset into the string table which contains the full name of the type, or 0xFFFFFFFF if the information is not available (e.g. if symbols have been stripped)

Flags

Bit offset	Description
0	1 if the type is a primitive, 0 otherwise
1-2	0 if the type is a class type 1 if the type is an abstract type 2 if the type is a struct type 3 if the type is an interface type *Note:* Bit 2 set to 1 implies that the type cannot be instantiated *Note 2:* Abstract types and interface types are always classes (reference types)
3-64	Reserved for future use

Static constructor method

An offset into the methods table which contains the static constructor for the type. The method must set the flag static and constructor and must expect no parameters and return System.Void.

This should be set to 0xFFFFFFFFFFFF if a static constructor is not required to run before the first instantiation of a type.

Num_instance_fields

The number of fields owned by an instance of the object

Num_static_fields

The number of fields owned by the type itself, and is held globally.

Primitive Code

Zero if the type is not a primitive or one of the following values:

Type	First byte (type code)	Second byte (size on stack)
(Not a primitive)	0x00	0x00
System.Void	0x01	0x00
System.Boolean	0x18	0x01
System.Byte	0x02	0x01
System.Int16	0x03	0x02
System.Int32	0x04	0x04
System.Int64	0x05	0x08
System.Single	0x06	0x04
System.Double	0x07	0x08
System.Char	0x08	0x01
System.SByte	0x09	0x01
System.UInt16	0x0A	0x02
System.UInt32	0x0B	0x04
System.UInt64	0x0C	0x08
System.IntPtr	0x0D	0xFF[25]
System.UIntPtr	0x0E	0xFF
System.Enum	0x0F	0x04[26]
System.String	0x10	0x00[27]
System.Delegate	0x11	0x00
System.MulticastDelegate	0x12	0x00
System.Array	0x13	0x00
System.Exception	0x14	0x00
System.Type	0x15	0x00
System.Attribute	0x16	0x00
System.IDisposable	0x17	0x00

Reserved

Reserved for future use and for use by the runtime

XIL Fields

```
struct XILFieldHeader {
  DWORD name;
```

[25] The size on the stack depends on the the processor native word size

[26] Depending on the exact enum defined, this may not be size on the stack

[27] For reference types, the size on the stack is implementation defined, and this value is ignored.

```
  DWORD flags;
  QWORD initialvalue;
  XILType type;
};
```

Name

An offset into the string table for the field name, or 0xFFFFFFFF if the name is not available (e.g. if symbols have been stripped)

Flags

Bit offset	Description
0	0: The field is an instance field 1: The field is a static field
1-2	0: The field is marked as private 1: The field is marked as protected 2: The field is marked as public
3-64	Reserved for future use

Initial Value

If the XILField is a primitive type, this field can be used to store the initial value of the primitive before the static or instance constructor respectively is called. If the XILField is not a primitive type, this field is ignored.

If the primitive type is less than 8 bytes, the low order bytes are used.

Type

The type of the field, as an XILType sequence.

XIL Attributes

```
struct XILAttribute {
  DWORD cbCodeSize;
  BYTE reserved[28];
  BYTE data[cbCodeSize];
};
```

Code Size

The size in bytes of the data section of the XIL attribute

Reserved

These bytes are reserved for future use

Data

XIL-instructions which correspond to creating the custom attribute, for example in the method signature:

```
[DllImport("User32.dll", "MessageBoxA")]
public static extern int MessageBox(int h, string m, string c, int type);
```

The attribute corresponds to the code-signature creating the DllImport attribute, equivalent to the code fragment

```
new DllImportAttribute("User32.dll", "MessageBoxA");
```

i.e the four XIL instructions

```
LdString "User32.dll"
LdString "MessageBoxA"
Comma
NewObject DllImportAttribute::DllImportAttribute(string, string)
```

This code fragment **must not** contain any references to objects or methods outside of the attribute – that is to say no method calls may be made and no static variables may be referenced during the attribute creation. The code fragment **should not** attempt to inline the object creation.

XIL Methods

```
struct XILMethod {
  struct XILMethodHeader {
    DWORD name_index;
    WORD num_locals;
    WORD num_params;
    WORD num_try_catch_blocks;
    BYTE reserved1[2];

    DWORD flags[2];
    DWORD cbCodeSize;

    WORD num_contract_ensures;
    WORD num_contract_requires;

    WORD num_attributes;

    BYTE reserved2[2];
  } header;

  DWORD attributeIndexes[num_attributes]
  XILParameter parameters[num_params];
  XILLocal locals[num_locals];
  XILContract ensures[num_contract_ensures];
  XILContract requires[num_contract_requires];
  XILTryCatchBlock exceptionblockinfo[num_try_catch_blocks];

  XILType returnType;
  DWORD magic_func_prefix;
  BYTE code[cbCodeSize];
};
```

Num locals

The number of local variables in the method

Num params

The number of parameters to the method

Num try/catch blocks

The number of try..catch, try..finally and try..catch..finally blocks in the method.

Flags

Bit offset	Description
0	If set, the method corresponds to a method-style function in the source code
1	If set, the method is abstract and has no body
2	If set, the method is extern and has no body
3	If set, the method is a finalizer
4	If set, the method is a constructor
5	If set, the method corresponds to a get accessor in the source code.
6	If set, the method corresponds to a set accessor in the source code
7	If set, the method corresponds to an index getter in the source code
8	If set, the method corresponds to an index setter in the source code
9	0: The method is a static method 1: The method is an instance method

Bit offset	Description
10	If set, the method is a virtual method
11-12	0: The method is public 1 :The method is private 2: The method is protected
14-64	Reserved for future use

cbCodeSize

The size in bytes of the XIL instructions which make up the method

Num contract ensures, num contract requires

The number of ensures and requires contracts respectively marked on the method

Reserved1

Bytes reserved for future use. This field additionally ensures that the later elements in the structure are DWORD aligned.

Reserved2

Bytes reserved for future use

Return type

The return type of the method

Parameters

The list of parameters which the method expects

Locals

The list of local variables that the method expects

Ensures

The list of ensures contracts defined on the method

Requires

The list of requires contracts defined on the method

Exception information

A list of num_try_catch_blocks exception-information blocks which are used by the runtime to divert control flow during managed exceptions.

Magic function prefix

A four-byte prefix to the start of the method body. This *must* be the hexadecimal value 0x46554e43, which corresponds to the ASCII string 'FUNC'.

Code

The XIL instructions which make up the method body

XIL Parameters

```
struct XILParameter {
  DWORD name;
  DWORD flags;
  XILType type;
}
```

Name

The offset into the string table which is the name of the parameter, or 0xFFFFFFFF if no value is available (e.g. if symbols have been stripped)

Flags

Bit offset	Description
0-1	0: parameter should be passed normally (the parameter is passed by value for reading only) 1: the parameter is marked as OUT (the address of the parameter is passed for writing only) 2: the parameter is marked as REF (the address of the parameter is passed for reading and writing) 3: the parameter is marked as PARAMS, and is passed as an array.
2-32	Reserved for future use.

Type

The type of the parameter

XIL Locals

```
struct XILLocal {
  QWORD initialvalue;
  XILType type;
}
```

Initial Value

If the XILLocal has primitive type, this field can be used to store the initial value of the local before the method is called. If the XILLocal is not a primitive type, this field is ignored.

If the primitive type is less than 8 bytes, the low order bytes are used
Rationale: This is equivalent to casting the primitive type to QWORD before storage. *End rationale.*

Type

The type of the local variable

XIL Contracts

```
struct XILContract {
  DWORD cbSize;
  BYTE instructions[cbSize];
}
```

cbSize

The size in bytes of the contract

Instructions

The XIL instructions which make up the contract definition

XIL Try-catch blocks

```
struct XILTryCatchBlock {
  DWORD try_begins_offset;
  DWORD catch_begins_offset;
  DWORD finally_begins_offset;
  DWORD finally_ends_offset;
  XILType catchType;
}
```

Try begins offset

The offset into the method where the try block begins. The offset is in XIL instructions, not in bytes.

Catch begins offset

The offset into the method where the try block ends and the catch block begins. The offset is in XIL instructions, not in bytes.

If this block is a try..finally block, this is the end of the try block and has the same value as the finally begins offset field.

Finally begins offset

The offset into the method where the catch block ends and the finally block begins. If the block is a try..catch block with no finally clause, this has a value equal to the finally_ends_offset.

Finally ends offset

The offset into the method where the finally block ends (and thus the try..catch, try..finally or try..catch..finally block ends.)

Catch type

The type of exception which is handled by this catch block, or 0xFFFFFFFF if the block is a try..finally block.

XIL Type descriptor

An XIL Type is a closed sequence of XILType structures which describe the XIL type fully.

If the XILType uses flags to declare that the type is a type-extension (either a pointer, array or nullable/non-nullable type extension) then the XILType is followed by another XILType which must itself be closed (recursively) which qualifies the XILType. The cbSequenceSize field in the parent XILType must contain the size of the XILTypes which form arguments to the parent XILType.

If the XILType is declared as a generic closure, then the first argument to the XILType is the generic declaring type (e.g. List<T> in List<int>) followed by XILTypes which are the generic arguments to the type, (e.g. System.Int32 in List<int>). Note that the first argument to the XILType is not closed, but the parent XILType *is* closed. The number of generic arguments must equal the number of generic parameters to the method.

```
struct XILType {
  WORD cbSequenceSize;
  union {
    struct {
      WORD modifiers;
      WORD generic_parameter_id;
    } infoValue;
    DWORD typeValue;
  } data;

  XILType qualifiers[cbSequenceSize / sizeof(struct XILType)];
};
```

cbSequenceSize

The size in bytes of the XILTypes which are arguments to this XILType structure and which appear immediately after this XILType structure. (This is 0 if there are no arguments to the XILType structure).

```
char* skipOverXILType(XILType* type){
   return ((char*)type) + sizeof(XILType) + type->cbSequenceSize;
}
```

Modifiers / declaring type

This field contains an offset into the type table if the cbSequenceSize is 0, otherwise it contains flags which modify the XILType structure that immediately follows in the sequence. The modifiers are shown below:

Value	Meaning
0x01	The type forms a non-nullable form of the following class type, which is a reference type which cannot hold the value “NULL”. In all cases, the variable site must be definitely assigned before it can be used, and cannot have the value NULL assigned into it, thus the variable site will never yield NULL when read by managed code. If this modifier is used, the XILType structure is immediately followed by another XILType which is a reference type.
0x02	The type is a nullable field of the following class type, which is a value type that can additionally hold the value “NULL” It is not necessarily the case that the type exists on the heap, and this does not convert the value-type to a reference-type. It merely indicates that the value ‘NULL’ is a valid value for the type to have assigned to it, or for it to yield. The actual working of this parameter is implementation defined. If this modifier is used, the XILType structure is immediately followed by another XILType which is a value type.
0x03	The type is a generic closure of the following class type. If this modifier is used, the XILType structure is immediately followed by another XILType which is the type to be closed, which is then followed by XILType structures which are the generic parameters to the type.
0x04	The type is a one-dimentional array of the following type

Value	Meaning
	Note that the Xinaos runtime supports jagged arrays, but does not support multi-dimensional rectangular arrays in the way that C# does – instead, C# multi-dimentional arrays are converted to single-dimentional arrays
0x05	The type is an unmanaged pointer to the following type If this modifier is used, the XILType structure is immediately followed by another XILType which must be a value-type to which an unmanaged pointer will be obtained
0x06	The type is a generic argument- that is, this entire type is defined when the generic type is closed with a generic parameter. The generic parameter to use is given by the generic parameter field If this modifier is used, it is immediately followed by the generic type that defines the parameter, for example, if an internal class of Dictionary<TKey, TValue> can reference the generic parameters in its parent enclosing type by referring to its parent as the argument to this XILType sequence element.

Generic parameter id

In types which contain generic parameters (e.g. Dictionary<TKey, TValue>), the type TValue becomes a valid type on fields and methods. If the type TValue is translated into the equivalent XIL type, we first set the generic parameter flag on the flags and set the generic_parameter_id field to be the zero-indexed number into the generic parameter list which corresponds to TValue – in this case the second parameter, or zero-based index 1.

When the generic parameter id is set, exactly one qualifying type is needed which is the type which declared the generic parameter, in this case Dictionary<TKey, TValue>

Declaring Type

If the type is self-closing (e.g. int, object, ComplexNumber) or the type is the first argument to a generic closure (e.g. List<T>, IEnumerable<T>) then this contains the offset into the type table which corresponds to the appropriate type.

Examples:

```
// int:
XILType*  eg1 = {
   cbSequenceSize: 0,
   Data.declaringType: lookupType("System.Int32")
  };

// int*
```

```
XILType* eg2 = XILType[2] {
    {
      cbSequenceSize: sizeof(XILType),
      flags: FLAG_POINTER
    },
      {
        cbSequenceSize: 0,
        Data.declaringType: lookupType("System.Int32")
      }
  };

// Dictionary<int?, string[]>
XILType* eg3 = XILType[6] {
    { // Dictionary<int?, string[]>
      cbSequenceSize: sizeof(XILType) * 5,
      Flags: FLAG_GENERIC_CLOSURE
    },
      { // Dictionary<TKey, TValue>
        cbSequenceSize: sizeof(XILType) * 4,
        Data.declaringType: lookupType("System.Collections.Dictionary<,>")
      },
      { // int?
        cbSequenceSize: sizeof(XILType),
        Flags: FLAG_NULLABLE
      },
        { // int
          cbSequenceSize: 0,
          Data.declaringType: lookupType("System.Int32")
        },
      { // string[]
        cbSequenceSize: sizeof(XILType),
        Flags: FLAG_ARRAY
*      }
        { // string
          cbSequenceSize: 0,
          Data.declaringType: lookupType("System.String")
        }
  }
```