

Modernizing Game Development on Nintendo Wii Hardware

Sam Collier

BSc (Hons) Computer Games Technology

School of Design and Informatics
Abertay University

April 2026

Table of Contents

List of Figures	iii
List of Tables	iv
List of Listings	v
Acknowledgments	vi
Abstract	vii
1 Introduction	1
1.1 Overview	2
2 Literature Review	2
2.1 Dynamic Memory Allocation	2
2.1.1 Allocation Algorithms	3
2.1.2 Arena Allocation	4
2.1.3 Real-time Memory Allocator: TLSF	4
2.2 Debugging on Constrained Hardware	5
2.2.1 Win32 MiniDumps: Design Inspiration	6
2.2.2 The DWARF Standard	6
2.2.3 Crash Dump Collection on Embedded Systems	6
2.2.4 Shortcomings of DWARF: Stack Unwinding	7
2.3 Summary and Research Gaps	7
3 Methodology	8
3.1 Zig as an Integrated Toolchain	8
3.2 Allocator Implementations	9
3.2.1 Bump Allocator	10
3.2.2 Free List Allocator	10
3.2.3 TLSF Allocator	10
3.2.4 libc malloc	11
3.3 Benchmark Design	11
3.3.1 Measuring Fragmentation	11
3.3.2 Workloads	12
3.3.3 Benchmark Execution Flow	13
3.4 Metrics	13
3.5 Data Collection Pipeline	14
3.5.1 Tracy Profiling	14
3.6 Postmortem Crash Dump Pipeline	14
3.6.1 Exception Handler	14
3.6.2 Crash Dump Format	15
3.6.3 Crash Dump Transmission	15
3.6.4 Symbolization Tool	16
3.7 Stack Unwinding: PowerPC ABI Versus DWARF	17
3.8 Test Infrastructure	18
4 Results	18

4.1 Allocation Latency	19
4.2 External Fragmentation	20
4.3 Internal Fragmentation	21
4.4 Stack Unwinding Experiment	22
5 Discussion	22
5.1 Allocator Performance	22
5.1.1 Single Lifetime Workload	22
5.1.2 Mixed Lifetime Workload	23
5.2 Stack Unwinding Experiment	24
5.3 Limitations	25
6 Conclusion	25
6.1 Future Work	26
List of References	27

List of Figures

Figure 1	Main architecture diagram of the Nintendo Wii (Copetti, 2020)	1
Figure 2	Diagram of TLSF’s bitmap structure (Bhattarai, 2024)	5
Figure 3	A diagram of the Zig pipeline	8
Figure 4	External Fragmentation Equation	12
Figure 5	Internal Fragmentation Equation	12
Figure 6	Screenshot of Gitlab’s CI Unit Test Dashboard	18
Figure 7	Mean allocation latency per frame (single lifetime workload).	19
Figure 8	Mean allocation latency per frame (mixed workload).	19
Figure 9	Peak external fragmentation per frame (single lifetime workload).	20
Figure 10	Peak external fragmentation per frame (mixed workload).	20
Figure 11	Mean internal fragmentation per frame (single lifetime workload).	21
Figure 12	Mean internal fragmentation per frame (mixed workload).	21
Figure 13	Allocation failures per seed (mixed workload).	22

List of Tables

Table 1	CrashDump layout capturing CPU state. Many of these values aren't used in this project, but it's important that the entire CPU state is transmitted if possible.	15
---------	---	----

List of Listings

Listing 1 Pseudocode for crash dump transmission using <code>rfi</code>	16
Listing 2 FDE row scan in the DWARF unwinder (Collier, 2026)	25

Acknowledgments

I'm super thankful for my supervisor, Erin Hughes, who didn't laugh me out of the room when I proposed porting Unreal Engine 5 to the Wii for my dissertation.

From the beginning she took my insanely ambitious idea seriously and enthusiastically. Even though the project shrank consistently throughout the year, without her belief, I wouldn't have gotten anywhere.

I'd also like to thank my parents for regularly checking in on my sanity from over 3,000 miles away. I think I would've combusted from within otherwise.

Finally, thank you to the colleagues I've worked with in industry. You all had a large influence on me last year and fostered my new interest in low level systems which led to this project.

Abstract

This project’s ultimate objective is to port Unreal Engine 5 to the Nintendo Wii. However, before the engine can be ported, two questions must be answered. Which memory allocator best compliments the Wii’s physical memory constraints, and what toolchain approach best supports developer efficiency on hardware with no friendly debug tooling.

This dissertation aims to answer both questions empirically on Wii hardware.

Three allocators were implemented in Zig and benchmarked on two different workloads. One with a single object lifetime and one with mixed lifetimes to better represent a game engine’s allocation behavior. A crash dump pipeline was also implemented which sends a crash dump over TCP when a crash occurs. A custom tool written in Rust can then read the dump and point the developer to where in the source code the crash happened.

All three allocators measured almost zero external fragmentation on the single lifetime workload. On the mixed lifetime workload, TLSF’s splitting policy led to several allocation failures while the free list produced none. TLSF’s $O(1)$ latency guarantee holds in practice, but its higher splitting threshold is not suitable for an engine with mixed lifetime allocation behavior. The debugging experiment confirmed that Zig emits correct DWARF CFI on PowerPC, and that both unwinding techniques show accurate callstacks from a hardware exception. The experiment also highlighted that an incorrect DWARF unwinder implementation can produce incorrect but plausible output. Comparing the output of both methods was the only reliable way to detect issues with the DWARF unwinder.

Both contributions will inform a future attempt to port Unreal Engine 5 to the Wii.

1 Introduction

The Nintendo Wii was released in 2006. Despite launching alongside the Xbox 360 and PlayStation 3, the Wii's hardware was more similar to the previous generation. The CPU is a PowerPC 750CL, codenamed Broadway, which is single core and runs at 729 MHz (Copetti, 2020). The Wii has 88 MB of total system memory distributed between two separate physical banks. MEM1 is 24 MB of fast 1T-SRAM and MEM2 is 64 MB of slower GDDR3 (Copetti, 2020). The system on chip (SoC) is named Hollywood (Copetti, 2020). It houses Flipper which is the same fixed-function GPU from the GameCube, but 1.5x its original clock speed with 3MB of video memory (Copetti, 2020). Hollywood also houses MEM1 and a secondary ARM processor that handles I/O called Starlet (Copetti, 2020).

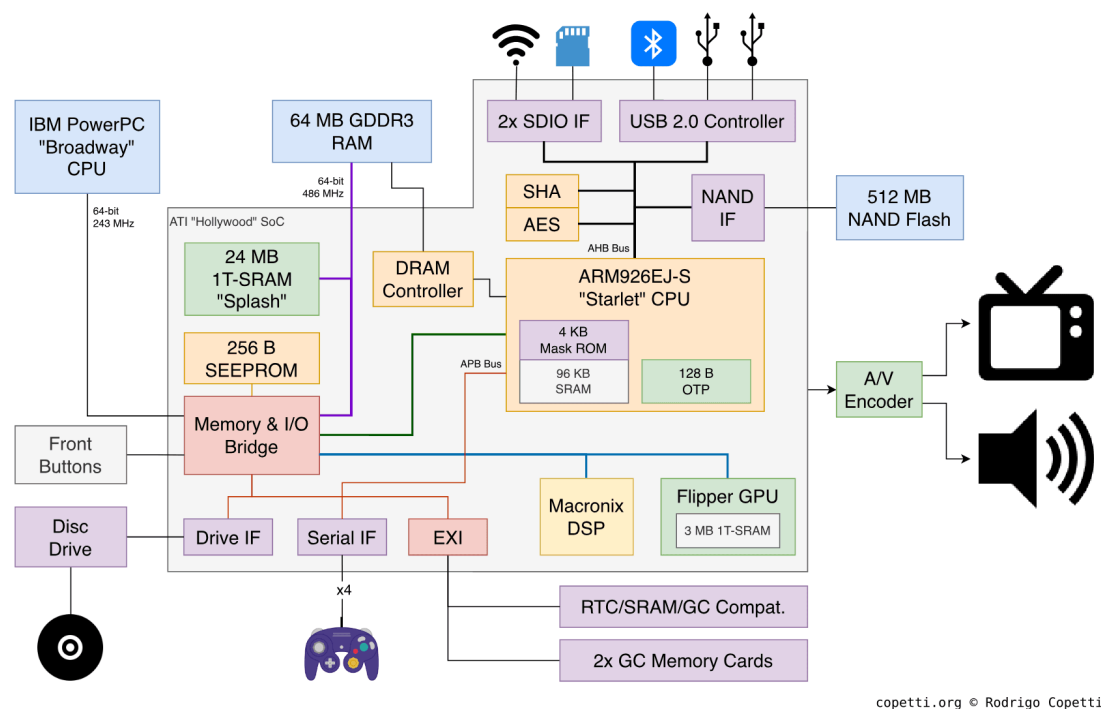


Figure 1: Main architecture diagram of the Nintendo Wii (Copetti, 2020)

Despite these archaic hardware limitations, an active homebrew community continues to develop software for it to this day. Homebrew development relies on devkitPro which provides a GCC PowerPC compiler and libraries to interact with the hardware (devkitPro, 2020). So every project has a dependency on C.

Unreal Engine 5 (Epic Games, 2020) is the industry standard game engine used for modern game development. It is intended for platforms starting in the console generation after the Wii (Meer, 2008), such as the PlayStation 4 and Xbox One, with gigabytes of RAM and programmable shader pipelines. Retrofitting Unreal Engine 5 onto the Wii would require working around most assumptions the engine makes about the platform it's on. Porting Unreal Engine 5 to the Wii is the ultimate goal (Collier, 2025a), but two prerequisite questions must be answered first.

First is figuring out memory allocation. Unreal Engine 5 allocates across a variety of lifetimes and sizes, and adds even more pressure by running a garbage collector for UObjects. On the Wii, there is no virtual memory abstraction to automatically handle allocation failures. An allocator under a constrained physical memory budget must handle mixed lifetime pressure without failing. The existing literature evaluates allocators under workloads that do not reflect the mixed lifetime patterns of a game engine. This dissertation benchmarks three allocators on Wii hardware using two workloads designed to simulate this pressure better.

The second is about development efficiency. When a program crashes on the Wii, there is no debugger to attach to it. Manually reproducing the exact state that caused a crash is tedious and not always possible. Instead, when a debugger isn't immediately available during a crash, modern developers analyze crashes by reading saved crash dump files after the fact. This dissertation implements this modern crash dump pipeline for the Wii.

The research question motivating this work asks what custom toolchain approaches best balance hardware performance and developer efficiency on platforms with unique memory constraints. This dissertation investigates that question on the Wii. The scope of this dissertation does not include a Wii port of Unreal Engine 5. That is future work described in Section 6.1. The contributions here lay some of the foundations for that future work.

1.1 Overview

Section 2 reviews the existing literature on dynamic memory allocation and postmortem debugging. Section 3 describes the allocator implementations, benchmark workloads, crash dump pipeline, and debugging experiment. Section 4 presents the benchmark measurements and debugging results. Section 5 interprets the results and discusses their implications. Section 6 draws conclusions from the study and lays out a plan to achieve the original goal.

2 Literature Review

This chapter reviews literature relevant to two of this project's evaluation pillars, dynamic memory management and postmortem debugging. These areas directly map to the two core concerns of the research question: hardware performance and developer efficiency. This section also identifies gaps in the research landscape to motivate the experiments conducted in this project.

2.1 Dynamic Memory Allocation

Memory is a very limited resource on the Wii. Every allocation decision directly impacts the capabilities and performance of the Wii. Both hardware performance and developer efficiency are coupled to memory.

2.1.1 Allocation Algorithms

When evaluating allocator implementations, a distinction between *policy* (which free block to use for a given request) and *mechanism* (the data structure chosen to get the most out of the *policy*) is made (Wilson *et al.*, 1995; Masmano *et al.*, 2004). This separation matters because the same *policy* can be implemented with various *mechanisms*, each with their own time and space costs, and the same *mechanism* can support multiple *policies*. Fragmentation measurements across eight C and C++ programs demonstrate that the *mechanism*'s implementation overhead, including block headers and alignment padding, has a greater impact on fragmentation results than differences in placement *policy* (Johnstone and Wilson, 1998).

When the *mechanism*'s overhead is unaccounted for, fragmentation measurements also show that conventional allocators achieve close to zero fragmentation. Two limitations of Johnstone and Wilson (1998)'s methodology include the exclusion of implementation overhead and choice of applications measured.

2.1.1.1 Implementation overhead is treated as noise

Fragmentation measurements capture placement efficiency but exclude per-object headers and alignment padding under the assumption that *policy* evaluation is the primary concern (Johnstone and Wilson, 1998). However, since *mechanism* overhead turns out to be more important for fragmentation, excluding it obfuscates the dominant source of memory waste creating a misleading and optimistic picture. On general-purpose systems, this cost is negligible relative to a large available address space, and excess memory pressure can be alleviated by the operating system through virtual memory abstraction (Bryant and O'Hallaron, 2015). However, on 88 megabytes of physical memory with no operating system, neither of these safeguards are available.

2.1.1.2 The programs tested are not representative of video game workloads

All the programs tested by Johnstone and Wilson (1998) are deterministic applications whose allocation behavior is fully determined by their input, like a C compiler. Other allocator studies have broadened the range of workloads to include server and database applications with variable allocation patterns driven by user demand (Durner *et al.*, 2019). However, existing work in this area is limited. Few studies evaluate allocators under workloads that combine bidirectional allocation variability, a hard real-time frame budget, and a fixed physical memory limit with no virtual memory fallback. Masmano *et al.* (2004) acknowledges using Johnstone and Wilson (1998)'s non real-time workloads and argues that real-time applications should not behave differently enough to invalidate the results. Whether the near zero fragmentation results hold under this specific combination of constraints is an open question awaiting empirical results.

2.1.2 Arena Allocation

Object lifetimes significantly impact allocator performance where such lifetimes are known in advance, allocation cost, measured in instructions per byte allocated, drops to nearly half that of general-purpose alternatives (Hanson, 1988).

Instead of searching for a suitable free block on each allocation, objects sharing a lifetime are served from a contiguous region by advancing a pointer (Hanson, 1988). When the region's lifetime ends, instead of freeing individual objects one by one, objects of the same lifetime are freed collectively in one fell swoop by simply resetting a pointer, eliminating per-object free cost entirely (Hanson, 1988; Hall, 2019). Allocators built on these design principles are known by many names: linear allocators, arena allocators and region-based allocators (Hall, 2019). This dissertation will refer to them as arena allocators.

This preference for taking advantage of lifetimes when allocating is well established in game development (Gregory, 2018). Custom allocators tailored to a game's object lifetimes are strongly preferred over general-purpose alternatives in performance-critical game code. `malloc`, `free`, and C++'s `new` and `delete` are explicitly discouraged in favor of allocators whose structure mirrors the game's allocation patterns (Gregory, 2018). Arena allocation maps well to a video game context. For example, objects allocated during the loading of a level could map to a region of memory whose lifetime would span the duration of the loaded level. The Wii's architecture also encourages organizing memory into regions since it has its two distinct physical regions, MEM1 and MEM2. MEM1 would be suitable for per-frame and per-level data, and MEM2 would be better for streaming large assets.

2.1.3 Real-time Memory Allocator: TLSF

The Two-Level Segregated Fit (TLSF) allocator (Masmano *et al.*, 2004) is specifically designed for real-time operating systems. Real-time operating systems are made for critical embedded hardware with strict timing constraints like medical devices. They prioritize determinism and predictability, so allocation latency must be known in advance. TLSF achieves constant time complexity ($O(1)$) for allocations and frees thanks to a two-level bitmap structure. Which means that the worst case allocator latency can be calculated ahead of time.

The implementation of TLSF provides a concrete example of the *policy* and *mechanism* distinction from Wilson *et al.* (1995). The *policy* is a slightly less precise variant of "best fit" (find the smallest possible block to satisfy the allocation). TLSF's strategy can be dubbed "good enough for rock and roll fit". It finds the smallest size class that can satisfy the request, and then takes any block from that class. So the quality of the best fit is less precise, but that precision is traded for something the *mechanism* can efficiently deliver. The *mechanism* is the two-level bitmap structure which is essential to the algorithm's $O(1)$ guarantee. As shown in

Figure 2, The first level separates free blocks by powers of two into buckets. The second level subdivides each of those buckets into another series of buckets. Each level is a bitmap where a set bit means that there is at least one free block in this bucket. So finding a suitable block is just a matter of masking the bitmap against the request size and then finding the first set bit, a single instruction on several architectures. The separation is clear because the constant time property of TLSF is a property of the *mechanism*. The *policy* was chosen because it's compatible with the *mechanism*. A true best fit *policy* would require searching all candidates which cannot be done with the chosen *mechanism*.

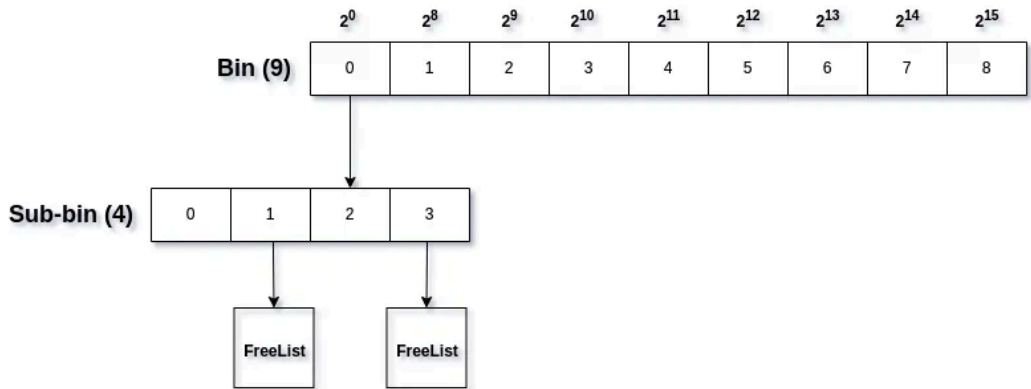


Figure 2: Diagram of TLSF's bitmap structure (Bhattarai, 2024)

In a game context, where a slow allocation during a frame produces a noticeable stutter, a predictable worst case allocation cost is a meaningful property that other general-purpose allocators don't have.

Pool allocators achieve constant-time allocation and zero fragmentation by fixing block size to exactly match the allocated object (Gregory, 2018). This eliminates fragmentation entirely at the cost of flexibility. TLSF has different tradeoffs. It accepts a predictable nonzero amount of fragmentation in exchange for handling variably sized allocations in constant time.

2.2 Debugging on Constrained Hardware

The development efficiency aspect of the research question also extends to what happens when things go wrong. A toolchain that cannot provide helpful crash diagnostics forces developers to reproduce bugs manually which adds significant friction to their development workflow because crashes may depend on runtime state that can't easily be replicated on demand. On the Wii, this problem is exacerbated even more because there is no system in place to manage a game's runtime exceptions like on modern operating systems. This section reviews literature relevant to postmortem debugging on constrained hardware, establishing both the design precedent for this project's crash dump solution and the open problems that this project's experiments address.

2.2.1 Win32 MiniDumps: Design Inspiration

Win32 MiniDumps (Microsoft, 2025) represent the modern crash dump workflow where only the necessary state is captured at the point of failure so it can be analyzed later on a different machine. The `MiniDumpWriteDump` function writes the minimum required crash information to a file, keeping it small so it can be sent over a network easily. The MiniDump file can then be opened in a debugger on a developer's machine to reconstruct the program's state at the point of failure. That debug pipeline is what inspires the Wii pipeline presented in this project: capture minimal crash state from the Wii, transmit it to a developer's machine, reconstruct the failure, and apply symbols to produce a useful diagnostic.

Both pipelines must balance dump size against the information needed to debug effectively. Simple problems like memory access violations probably only need callstack and local variables, but more difficult problems might need more information like global variable values or heap data. For Win32 MiniDumps, developers can control what's included in a MiniDump at runtime by passing flags through the `MINIDUMP_TYPE` flag set to `MiniDumpWriteDump`. On the Wii, there's no operating system to manage the program when it crashes. The exception handler runs with a very limited stack, so deciding what to include in the dump must be decided in advance at the toolchain level.

Also, MiniDumps don't necessarily have to be called in the context of a crash. As soon as anything bad is detected, `MiniDumpWriteDump` can be called to collect data for analysis offline. This capability is another design goal for the Wii pipeline. The crash handler should be able to be invoked on both hardware exceptions and other runtime events such as assertion failures or allocator exhaustion errors which can be captured before the system reaches a critical state.

2.2.2 The DWARF Standard

The DWARF Debugging Information Format (DWARF Debugging Information Format Committee, 2017) is a standard which compilers follow to encode source code debug information into compiled binaries. The Line Number Table maps code addresses to source locations. The Call Frame Information table provides the necessary data to unwind the callstack. Together, these two tables make it possible to take a raw crash address and get a corresponding source file, line number, and callstack. DWARF supports a wide range of architectures including the Wii's PowerPC platform, making it directly relevant to this project.

This project's toolchain embeds DWARF information into an Executable and Linkable Format (ELF) file which sits on the developer's machine.

2.2.3 Crash Dump Collection on Embedded Systems

The CrashDebug project (Green, no date) is a real world implementation of pairing a crash dump with a debug ELF sitting on the developer's machine for ARM

Cortex-M platforms without a live connection to the hardware. The dump and ELF then get loaded into The GNU Project Debugger (GDB) to provide a familiar debugging experience. Unfortunately, the PowerPC version of GDB used in this project (devkitPro, 2020) does not support core dumps, so there is no available tool which covers the Wii. This gap necessitates a custom symbolization tool for the Wii, another toolchain contribution for this dissertation described in Section 3.

2.2.4 Shortcomings of DWARF: Stack Unwinding

When a crash occurs, the CPU only knows the address of the instruction that failed. In order to reconstruct the callstack, the sequence of function calls that led to that instruction, one must apply a process known as stack unwinding. Stack unwinding is the process of walking backwards through the stack frames left by each function call in order to recover each function’s address. Having a callstack available is crucial for debugging anything nontrivial.

Bastian, Kell and Zappa Nardelli (2019) highlights that DWARF-based unwinding is unreliable, slow, and not always available. Their research contributions are attempts to bring stability into a DWARF-reliant setup. There are many cases of production compilers generating incorrect Call Frame Information tables. The Linux kernel avoids DWARF unwinding because it has also been proven unreliable in practice. The developer frustration DWARF’s unreliability caused is captured vividly by Torvalds (2012) responding to a proposal to reintroduce a DWARF unwinder:

“The whole (and only) point of unwinders is to make debugging easy when a bug occurs. But the f*cking dwarf unwinder had bugs itself, or our dwarf information had bugs, and in either case it actually turned several ‘trivial’ bugs into a total undebuggable hell... just follow the damn chain on the stack without the ‘smarts’ of an inevitably buggy piece of crap.”

— Torvalds (2012)

The PowerPC ABI provides a solution to stack unwinding which doesn’t rely on DWARF. The program must keep a linked list of stack pointers stored in a specific general purpose register (r1) at runtime (Zucker and Karhi, 1995). So developers on PowerPC can do as Torvalds (2012) suggested and follow the chain. This chain is a hard platform requirement rather than a wishful convention, making it reliable.

This project will compare the accuracy of DWARF unwinding with native ABI unwinding on a PowerPC program written in Zig.

2.3 Summary and Research Gaps

The literature reviewed in this chapter establishes the background for evaluating dynamic memory allocators and postmortem debugging workflows. However, gaps have emerged as well. Few studies exist which evaluate allocators under a game-

representative workload on constrained embedded systems. Also, there's no available tooling that supports postmortem debugging the Wii, and the reliability of DWARF unwinding on PowerPC remains an open question. The methodology in Section 3 will directly address both gaps.

3 Methodology

This chapter describes the experimental design and implementation choices used to address the two research gaps identified in Section 2. The first gap is addressed by an allocator benchmark suite that compares TLSF, a free list allocator, and a bump allocator across two workloads representative of game allocation patterns. Latency, fragmentation, and overhead are measured on physical Wii hardware. The second is addressed by a crash dump pipeline that captures CPU state on exception, transmits it over TCP, and symbolizes it against the project's ELF. An experiment then compares DWARF CFI and PowerPC ABI unwinding strategies. Both are driven by a single Zig command. An overview is shown in Figure 3.

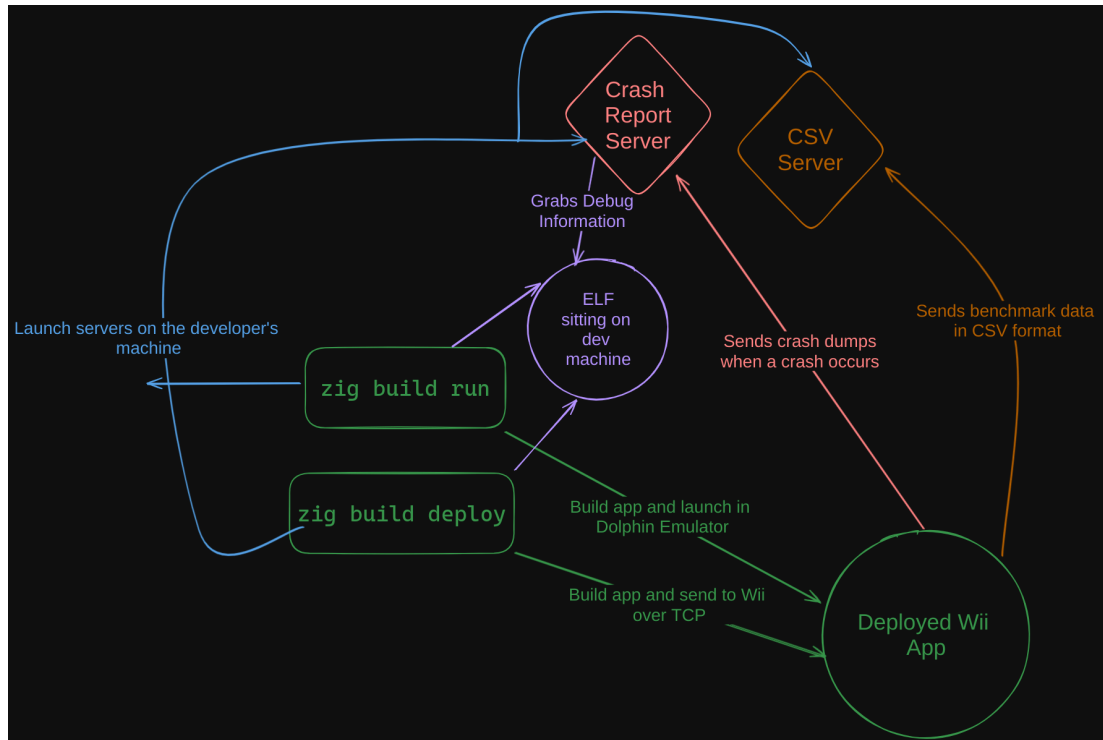


Figure 3: A diagram of the Zig pipeline

3.1 Zig as an Integrated Toolchain

Zig was chosen as the implementation language for two reasons that map directly to the research question. Firstly, it comes included with a build system and has support for cross-compilation out of the box. And secondly, it doesn't provide a default allocator like C's `malloc`. This design decision is justified in the language reference:

“The Zig language performs no memory management on behalf of the programmer. This is why Zig has no runtime, and why Zig code works seamlessly in so many environments, including real-time software, operating system kernels, embedded devices, and low latency servers.”

— Zig Language Reference (2025)

If a Zig function needs to allocate, then it must explicitly take in an allocator parameter which means that there’s no surprise allocations happening beneath functions, and it’s as easy as changing a function parameter to hotswap allocators. This property is essential to the benchmark design where the same workload code is run against each allocator without modification, ensuring identical measurement conditions across all comparisons.

The project targets the Wii with the help of devkitPPC (devkitPro, 2020) as a supporting toolchain. Zig handles cross-compilation to PowerPC using its own backend, and devkitPPC’s GCC is invoked purely as a linker, stitching Zig’s compiled object file together with libogc and other prebuilt Wii system libraries. The full build is driven by a single `zig build` command, which will download devkitPPC if necessary and build the entire project. The build produces both a DOL file for deployment to hardware or Dolphin, and an ELF file with DWARF debug information retained for the crash dump pipeline described in Section 3.6.

3.2 Allocator Implementations

Three allocators were implemented in Zig and evaluated against one another. Each one sits at a different point in the tradeoff space between allocation latency, fragmentation, and flexibility. All three allocators implement Zig’s standard allocator interface, so any allocator can be substituted as a parameter without modifying any benchmark code. This is the same design principle that makes Zig’s standard allocator interface useful, so the code being measured doesn’t care which allocator it’s talking to.

Each allocator runs one at a time claiming a contiguous region of physical memory from the Wii’s MEM1 arena. A Stats struct is maintained by each allocator, recording total capacity, current usage, peak usage, allocation and free counts, failure counts, total size of all the free blocks, and the size of the largest free block. The last two metrics are required to compute the fragmentation score defined in Section 3.3.1. Each allocator also tracks the overhead bytes consumed by internal bookkeeping. The bump allocator has no overhead while the free list allocator places a 12 bytes header before each payload and the TLSF allocator places a 24 byte header before each payload. Additionally, TLSF maintains a block pointer array and two bitmaps which are a fixed size throughout the life of the allocator.

3.2.1 Bump Allocator

The bump allocator is a type of arena allocator (Hall, 2019) being the fastest possible allocator with a fragmentation score of zero by nature. Each allocation involves just advancing a pointer through a contiguous memory region. The only way to reclaim memory is to call its unique reset function which restores the pointer to the start of the region, freeing all objects simultaneously in $O(1)$ time. Individual frees are no-ops. Because the free region is always one contiguous block, the fragmentation formula in Section 3.3.1 always evaluates to zero.

The bump allocator isn't a viable general-purpose allocator since it can't reclaim individual objects. But that's exactly what makes it a useful benchmark participant. It establishes what performance and fragmentation could look like when lifetimes are perfectly utilized.

The bump allocator also serves as the backing allocator for the benchmark's own data structures.

3.2.2 Free List Allocator

The free list allocator is the general-purpose baseline against which TLSF is compared. A linked list of free blocks is maintained with a first-fit placement *policy* (the first block large enough to satisfy the request (Wilson *et al.*, 1995)). A back-reference stored just before each payload allows the free operation to recover the block header from a raw pointer in $O(1)$ time. On free, the block is reinserted into the list and immediately coalesced with any adjacent free blocks.

Its $O(n)$ worst case allocation time is the cost of generality since finding a suitable block means scanning the list from the head. Latency grows with heap utilization and fragmentation, making it the expected worst performer under mixed lifetime conditions. It's included because it represents the simplest realistic general-purpose design and provides a meaningful contrast to the constant time guarantee offered by TLSF.

3.2.3 TLSF Allocator

The implementation follows the design described by Masmano *et al.* (2004) and the Zig implementation noted by Bhattarai (2024). TLSF is the primary candidate for video games. As reviewed in Section 2, it guarantees $O(1)$ worst case latency for both allocations and frees thanks to a two-level bitmap structure (Masmano *et al.*, 2004). A first level bitmap records which power of two size classes contain at least one free block (Masmano *et al.*, 2004). A second level array subdivides each power of two range into 32 sub-buckets and records which are nonempty (Masmano *et al.*, 2004). Finding a suitable free block is simplified to just masking the first-level bitmap against the request size and calling a count trailing zeros instruction to identify the smallest class to satisfy the request (Masmano *et al.*, 2004).

Two mapping functions handle the size to bucket translation (Bhattarai, 2024). `binmapUp` rounds a request size up to the next sub-bucket boundary, guaranteeing any block found is large enough. Without rounding up, it's possible to get a bucket whose blocks are slightly smaller than required. `binmapDown` maps a block's actual size to its true sub-bucket without rounding, so blocks are inserted at their exact size class. After a block is selected it's split if the remainder is large enough to form a new block. Otherwise, the remainder is absorbed by the payload. On free, physical neighbors are coalesced before the merged block is reinserted into the bitmap.

The implementation uses a first-level separation of $2^7 = 128$ bytes with 32 sub-buckets per power of two range, so the minimum allocation size is 128 bytes.

3.2.4 `libc malloc`

A fourth allocator wraps the `malloc` and `free` provided by `newlib`, the C standard library bundled with `devkitPPC` ("devkitPPC," no date). It is the allocator a developer using `libogc` would reach for by default. `Newlib's malloc` acquires memory by making an `sbrk` syscall, which on the Wii is not a thing. Instead, `libogc` provides a custom `sbrk` that advances the MEM1 arena's low pointer ("`sbrk.c`," no date). `Newlib's malloc` reuses previously freed blocks where possible, and only calls `sbrk` when no freed block is large enough to satisfy the request.

When MEM1 is exhausted, `sbrk` silently transitions into MEM2 and continues allocating there. The other three allocators are each initialized from a fixed pool in MEM1 and fail with an allocation error when that pool is exhausted. Although the `libc` allocator has access to more total memory under pressure, it should not affect the benchmark results reported here since peak usage on both workloads is well within MEM1. But it does mean that in a constrained low memory environment, the `libc` allocator cannot be used as a replacement for the fixed allocators without additional care.

Since there is no easy way to instrument what's going on under the hood of `malloc` and `free`, fragmentation scores, overhead bytes, and capacity statistics are unavailable.

3.3 Benchmark Design

The benchmark is structured around two workloads that each represent a distinct game allocation pattern. Workloads are run four times under different random seeds for each allocator.

3.3.1 Measuring Fragmentation

3.3.1.1 External Fragmentation

External fragmentation happens when there is enough total free memory for a request, but there are no individual free blocks big enough (Bryant and O'Hallaron,

2015). Johnstone and Wilson (1998) describe four ways to measure fragmentation, all ratios of memory used versus memory requested. These capture aggregate waste, which is sufficient on systems where the operating system manages physical contiguity through virtual memory. On the Wii, physical memory is fixed and there is no virtual memory fallback, so the concern is different. For the Wii, what's important is whether a contiguous block large enough for a future allocation will be available.

$$\text{external fragmentation} = 1 - \left(\frac{\text{size of largest free block}}{\text{total available memory}} \right)$$

Figure 4: External Fragmentation Equation

A score of 0.9 means that although free memory exists, almost none of it is likely to be usable as a single contiguous block.

3.3.1.2 Internal Fragmentation

Internal fragmentation occurs when an allocated block is larger than the payload it serves, due to alignment constraints or block header overhead (Bryant and O'Hallaron, 2015). Internal fragmentation measures bytes committed to the pool that don't serve any live allocation request.

$$\text{internal fragmentation} = \frac{\text{used bytes} - \text{requested bytes}}{\text{used bytes}}$$

Figure 5: Internal Fragmentation Equation

Where *used bytes* is total pool capacity minus total free bytes, and *requested bytes* is the sum of the originally requested sizes of all live allocations. A score of zero means every committed byte is serving a live request, and a nonzero score indicates overhead from metadata or alignment padding.

3.3.2 Workloads

Two workloads were designed to address the gap identified in Section 2.1.1.2. Both are driven by a seeded pseudo random number generator. Four seeds are used, and each workload is executed once per seed per allocator. Every allocator receives exactly the same sequence of allocation sizes and lifetimes for a given seed which ensures a fair comparison.

3.3.2.1 Single Lifetime Workload

The single lifetime workload simulates bursts of allocations of objects that only live within an iteration of the game loop. Sixty frames are simulated. Within each frame, between 20 and 150 objects are allocated with sizes drawn randomly between 64 and 1024 bytes. All objects allocated within a frame are freed before the next frame begins.

Fragmentation is sampled after each allocation and reported per frame.

This workload favors arena allocators like the bump allocator because the allocations all share a lifetime which can be exploited by the bump allocator’s easy ability to reset. This tests whether the free list and TLSF allocators can recover cleanly from short lived allocation bursts.

3.3.2.2 Mixed Lifetime Workload

The mixed lifetime workload is designed to produce realistic fragmentation by mixing up objects with different lifetimes on the heap. Thirty frames are simulated. Three persistent objects are allocated at startup to represent some persistent data such as level geometry that can’t be moved. Each subsequent frame, between 20 and 150 new objects are allocated with sizes between 16 and 4096 bytes and lifetimes between 2 and 25 frames. At the start of each frame, objects whose lifetime has expired are freed, and new objects are then allocated. The wider size range should stress TLSF’s bin selection across both its smallest and largest size classes, by producing patterns more representative of a real game’s mixed load.

The bump allocator can’t participate meaningfully here because it can’t free individual objects, and its fragmentation score of zero in this context reflects an inability to reclaim memory rather than allocator quality.

3.3.3 Benchmark Execution Flow

All measured runs are executed on physical Nintendo Wii hardware. Dolphin (“Dolphin,” no date), a Wii emulator, is used only for unit tests during development as described in Section 3.8. Two warmup iterations are run before any measured results are collected. The warmup cycles allow for any CPU cache effects to settle before measurement begins. Each measured run then iterates over all four seeds. Each workload is run against all allocators in sequence within each seed. Each allocator is initialized from scratch at the start of its run and torn down immediately after, so its claimed memory is returned to the system for the next run. Results collected throughout the benchmark are allocated from a bump allocator in MEM2 so that MEM1 is fully available to each allocator being tested.

3.4 Metrics

Three metrics are collected per allocator.

Allocation latency is measured in nanoseconds per operation using a hardware counter read immediately before and after each call, so the measured window contains only the allocator’s execution. Elapsed ticks are converted as $t_{ns} = \frac{\text{ticks} \times 10^9}{\text{clock frequency}}$.

Internal and external fragmentation are each measured using the equations defined in Section 3.3.1.

Allocation failures are counted whenever an allocator refuses a request.

3.5 Data Collection Pipeline

After all iterations complete, each recorded allocation is serialized to CSV with fields for latency and fragmentation scores. At the very end there are summary fields for each seed that show total failure counts. Float formatting is unavailable in Zig's standard library on PowerPC, so fragmentation scores are encoded as integers scaled by 10^6 and formatted with a decimal point during serialization.

The CSV is sent over HTTP to a server running on the developer's machine, and gets saved to a timestamped file. The benchmark determines at runtime whether it is running on physical hardware or inside Dolphin and selects the appropriate host address, so the same binary works in both environments without needing to recompile. A launcher tool starts the CSV collection server before starting Dolphin (`zig build run`) or dispatching to real hardware (`zig build deploy`), providing a single command that drives the full benchmark and data collection cycle.

3.5.1 Tracy Profiling

The benchmark is also instrumented with Tracy (Taudul, no date), a frame profiler designed for game development. Porting Tracy to the Wii required big endian network serialization, 64-bit atomic emulation via interrupt masking, Threading primitives for the Wii, and reduced queue sizes to fit within the Wii's system memory (Collier, 2025b).

Each benchmark run is wrapped in a Tracy zone, so that results are updated live while the benchmark runs. External and internal fragmentation scores and heap usage are pushed as named Tracy plots after each allocation. The benchmark blocks at startup until a Tracy client connects, so no early frames are dropped.

This complements the CSV pipeline with a live view of allocator behavior. The CSV remains the canonical source for the quantitative results reported in Section 4. Tracy support is exposed as a first class feature of the toolchain by enabling `-Dtracy` at build time. All instrumentation compiles to no-ops when the flag is not set, so game code using the toolchain can adopt Tracy without any runtime cost in release builds.

3.6 Postmortem Crash Dump Pipeline

3.6.1 Exception Handler

On PowerPC, hardware exceptions are dispatched through fixed address exception vectors in low memory. `libogc` installs its own handlers at these vectors during startup, and passes all hardware exceptions to a function pointer called `PPCExcptCurPanicFn` which is set to this project's custom crash handler. Zig's panic function is also overridden, so software panics are also routed through the custom crash handler and sent to the developer's machine.

A recursion guard prevents a fault inside the handler from looping indefinitely. The stack pointer is recorded at program startup before any stack frames are pushed, so that the upper bound of the usable stack is recorded. At exception time, the number of stack bytes to capture is $\min(4096, \text{stack_top} - \text{stack_ptr})$, so that the size of the dump doesn't get too big. The network connection is established during program initialization rather than from inside the handler during a crash. That's because the handler runs with a very limited stack and can't safely perform network setup.

The crash dump buffer must be allocated as a static variable rather than on the stack. A local buffer of the required size, 4 KB of stack plus the magic header and full CPU state, would be unsafe in an exception context. Stack depth at the point of a crash is unpredictable. A crash deep in a callstack only leave a few hundred bytes of stack which would be nowhere near the required size for the buffer.

3.6.2 Crash Dump Format

The dump is serialized into the static buffer. A magic header comes first which provides a synchronization marker so the receiver can locate the start of the dump even if garbage bytes arrive before the connection is established (which often happens). A `CrashDump` struct follows, whose contents are laid out in Table 1, capturing the full CPU state at the point of exception with the raw stack bytes appended at the end.

Field	Size	Contents
exid	4 B	Exception identifier (Dsi, FPU, ZigPanic, etc.)
pc, lr	4 B each	Program counter and link register
ctr, cr, xer, msr	4 B each	Count, condition, fixed point exception, and machine state registers
gpr	32×4 B	General-purpose registers r0 to r31
fpr	32×8 B	Floating-point registers f0 to f31
fpscr	8 B	Floating-point status and control register
gqr	8×4 B	Graphics quantization registers
ps	32×8 B	Paired single registers (Wii SIMD extension)
stack	up to 4 KB	Raw stack bytes from the stack pointer upward

Table 1: CrashDump layout capturing CPU state. Many of these values aren't used in this project, but it's important that the entire CPU state is transmitted if possible.

3.6.3 Crash Dump Transmission

The dump can't be sent from inside the exception handler. When PowerPC enters exception mode it clears `MSR[EE]` (External Enable), a bitfield in the Machine State Register (MSR) that controls whether external interrupts are allowed. The Wii's networking stack relies on Starlet, the ARM co-processor responsible for hardware

I/O, through an interprocess communication (IPC) that requires a hardware interrupt request (IRQ) to signal completion. With MSR[EE] cleared, a TCP send would block forever waiting for a completion signal that never happens.

Simply re-enabling interrupts with `mtmsr` won't work either. SRR0 and SRR1 are the save-restore registers that hold the address and machine state to return to once the exception is handled. MSR[RI] (Recoverable Interrupt) is also cleared on exception. While it's cleared, a nested exception would overwrite SRR0 and SRR1 before they are saved anywhere else.

Using `rfi` (return from interrupt) to leave exception mode before doing any I/O avoids both problems. Once the crash data is fully serialized into the static buffer, the handler writes the address of the `send_crash` function into SRR0 and the MSR value captured at the time of the crash into SRR1. That saved MSR value has EE, IR (instruction address translation), DR (data address translation), and RI all set, providing a normal execution context. Executing `rfi` then restores that machine state and jumps to the `send` function, which sends and closes the socket in a normal execution environment.

```
// MSR[EE] and MSR[RI] are cleared by hardware
on_exception(frame):
    serialize frame.registers and frame.stack into crash_buffer
    SRR0 <- address of send_crash
    SRR1 <- frame.msr           // saved MSR with EE, RI and all
other bits set
    rfi                       // restores MSR from SRR1 and jumps
to SRR0

// Normal execution context. Interrupts are re-enabled by rfi
send_crash():
    tcp_send(socket, crash_buffer)
    tcp_close(socket)
```

Listing 1: Pseudocode for crash dump transmission using `rfi`

3.6.4 Symbolization Tool

A server runs on the developer's machine. It accepts the TCP connection, scans the byte stream for the magic header, reads the `CrashDump` struct and the stack that follows, and writes them to a timestamped binary file. The symbolization tool, a Rust program, run with the dump file and the project's ELF as arguments.

DWARF debug information is loaded from the ELF using the `addr2line` and `gimli` crates (Fitzgerald and Craig, no date a; b). For each address recovered from the callstack, the corresponding function name, source file, and line number are retrieved from the DWARF Line Number Table. A formatted crash report is produced which contains the exception type, all register values, and a numbered

stacktrace where each frame is annotated with its function name, source location, and a snippet of the surrounding source code.

This part of the toolchain fills the gap identified in Section 2.2.3 where no tool exists for postmortem debugging on embedded PowerPC targets.

3.7 Stack Unwinding: PowerPC ABI Versus DWARF

The callstack is recovered from the crash dump using two strategies. Outputs are compared to evaluate the reliability of DWARF unwinding on Zig PowerPC code.

The first strategy is DWARF Call Frame Information (CFI) unwinding. Zig and other ELF targeting compiler emit `.eh_frame` by default on ELF targets rather than the DWARF standard `.debug_frame` (Bastian, Kell and Zappa Nardelli, 2019). Inspection of the linked ELF confirms this because `.debug_frame` only covers a handful of libogc stubs while all Zig compiled functions appear in `.eh_frame`. The unwinder steps through frames one at a time using the CFI rules to locate each saved return address in the crash dump's stack. As noted in Section 2, DWARF unwinding has a documented history of unreliability. Whether the Zig compiler generates correct CFI for PowerPC is an open question.

The second strategy is PowerPC ABI chain following. The PowerPC ABI requires that every function stores the caller's stack pointer at the first word of the current frame, creating a linked list that starts with the current value of `r1` (Zucker and Karhi, 1995). The link register (which holds the return address) saved by each function is stored four bytes above that word. Walking the callstack is then just a matter of following the chain, with no DWARF tables required. This is the approach Torvalds advocated for (Torvalds, 2012). It's a hard platform requirement and not a loose annotation for a compiler to get wrong.

The experiment uses a single crash scenario. A deliberate null pointer dereference at the bottom of a call chain five layers deep which creates a DSI (data storage interrupt) hardware exception. Both strategies are applied to the resulting dump and the recovered callstacks are verified against the known call sequence from the source code. The panic handler also transmits a crash dump, but stack traces from panic dumps are unreliable. In a hardware exception, the CPU saves the full register state before any code runs. In a software panic, the handler is an ordinary function call. By the time it captures `r1` (stack pointer), the compiler has already set up the handler's own stack frame. `pc` (program counter) is set to the where the crash is, but `r1` points inside the handler, so both strategies will produce garbage frames since they use `r1`. Panic dumps are still useful in practice since the panic message and approximate call site are still helpful for finding the source of a bug, but they are not suitable for a controlled experiment. The experiment uses a hardware exception so that both strategies can be evaluated fairly. A strategy is

reliable if it recovers all frames in order with correct function names and source locations.

3.8 Test Infrastructure

A suite of unit tests is provided for each allocator using Zig’s builtin test blocks. `std.testing` is not used because it pulls in POSIX dependencies unavailable on the Wii. So instead, a test signals failure by returning a Zig error. Tests can run on real hardware or in Dolphin. Tests cover initialization, single and multiple allocations, alignment guarantees, free and reallocate sequences, and out of memory behavior. In CI, tests run in Dolphin with `zig build test`.

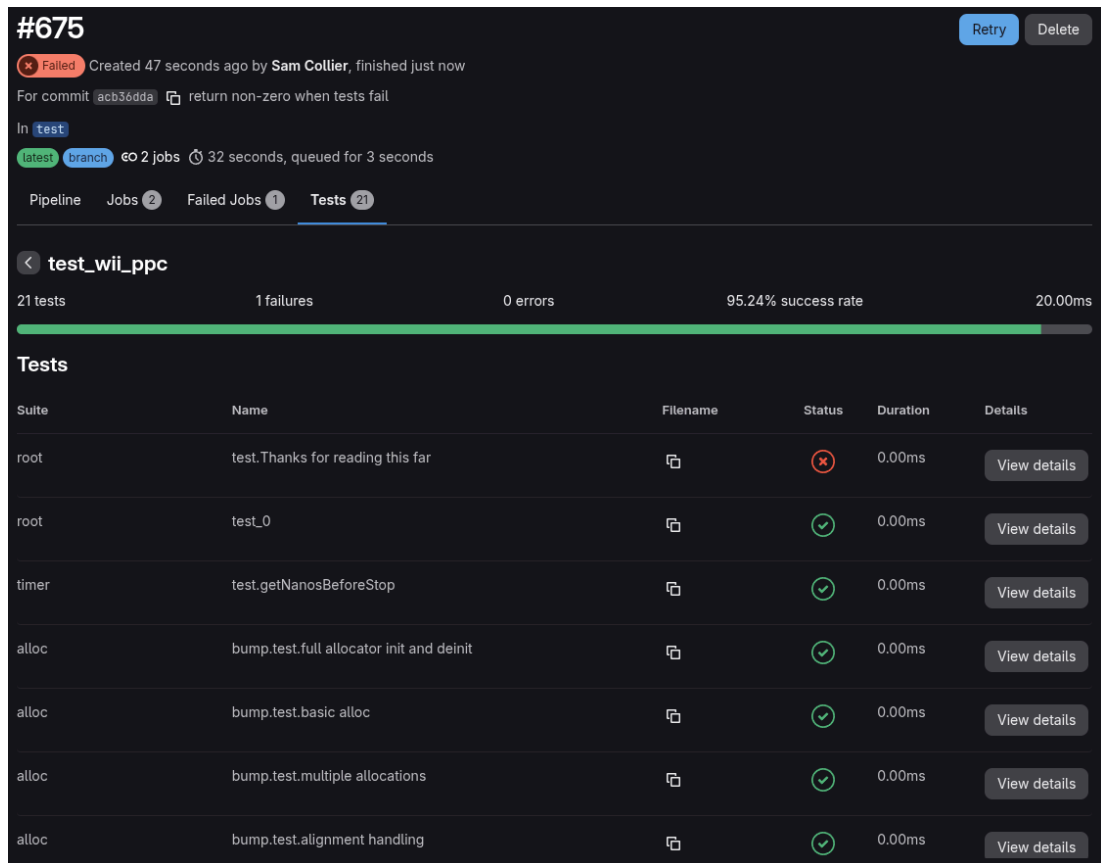


Figure 6: Screenshot of Gitlab’s CI Unit Test Dashboard

For the CI, Dolphin’s test output is collected and stored as a log file. A custom tool then parses the log into JUnit XML, allowing results to show up in CI dashboards that XML test output. This makes the test suite a first class participant in the project’s CI pipeline despite the unconventional execution environment. Modern developers use CI and unit testing to maintain quality and confidence when shipping titles (Soltani, 2018). So this addition to the toolchain also empowers developer efficiency.

4 Results

Results were collected by running the benchmark suite on a physical Wii. Each workload was executed once per seed per allocator across four different seeds.

4.1 Allocation Latency

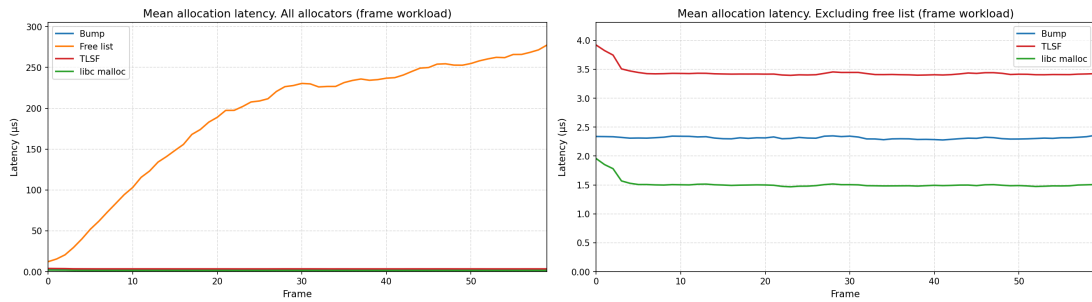


Figure 7: Mean allocation latency per frame (single lifetime workload).

On the single lifetime workload in Figure 7, the bump allocator recorded the lowest latency as expected, and remained flat throughout the run.

TLSF also maintained consistent latency throughout.

The free list allocator exhibited different behavior. Its mean latency grew linearly across the run.

libc malloc appears lower in the chart than even the bump allocator because it doesn't do any stat tracking. The libc line is included in the charts as a point of reference but shouldn't be used as a comparable measurement.

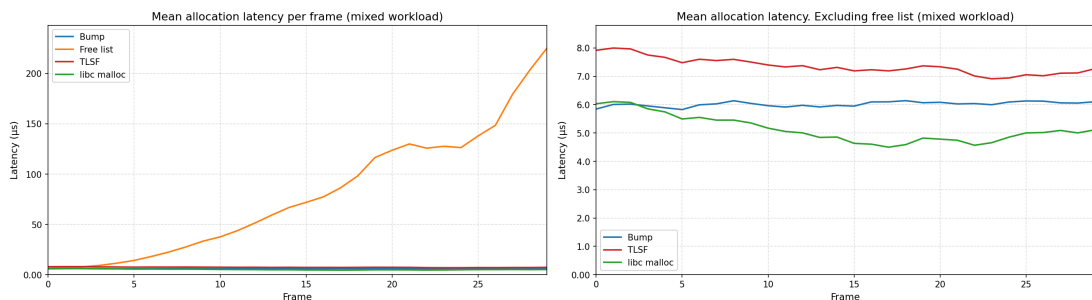


Figure 8: Mean allocation latency per frame (mixed workload).

On the mixed lifetime workload in Figure 8, latency was consistent with the single lifetime workload. Bump, libc, and TLSF all remained constant across the entire run. The free list grew linearly again.

4.2 External Fragmentation

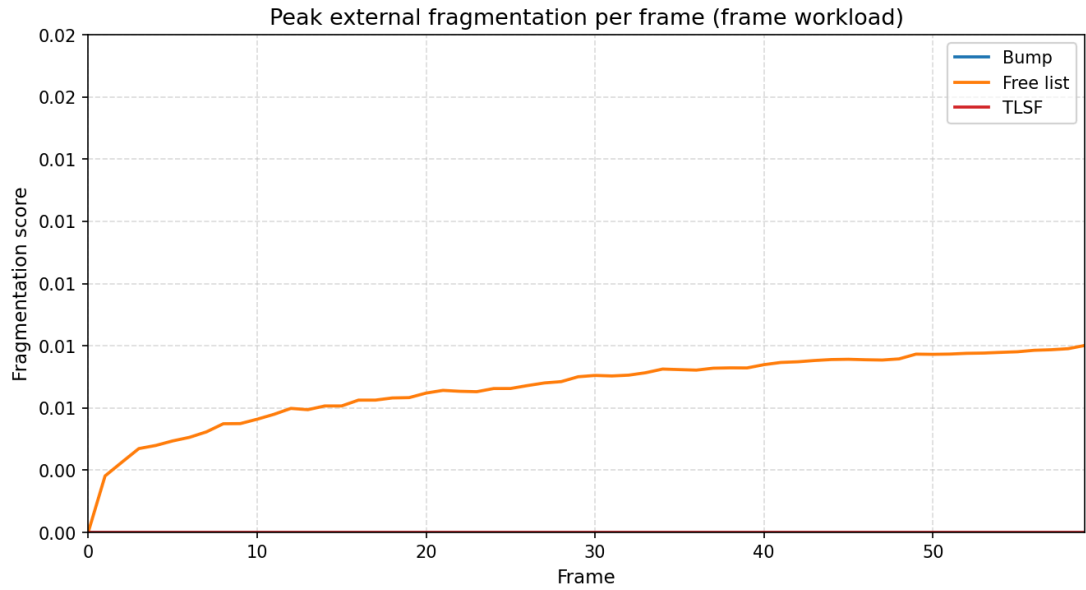


Figure 9: Peak external fragmentation per frame (single lifetime workload).

On the single lifetime workload in Figure 9, TLSF and the bump allocator both recorded zero external fragmentation across all frames. The free list recorded small scores.

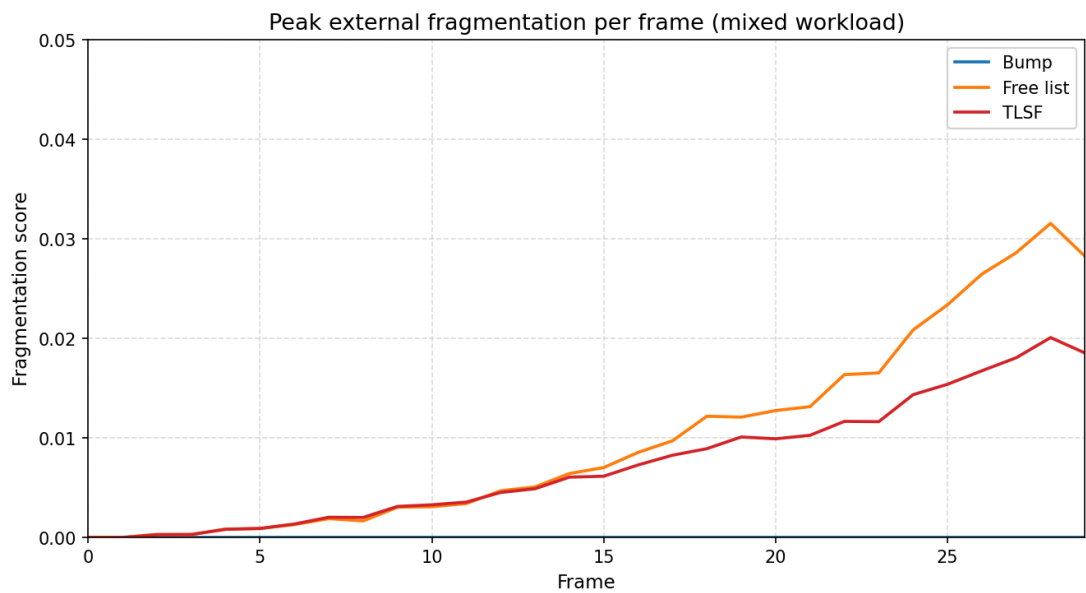


Figure 10: Peak external fragmentation per frame (mixed workload).

When introducing mixed lifetimes in Figure 10, fragmentation behavior changed. Both the free list and TLSF tracked closely initially, but then diverged with the free list pulling ahead. The bump allocator's score remained zero.

4.3 Internal Fragmentation

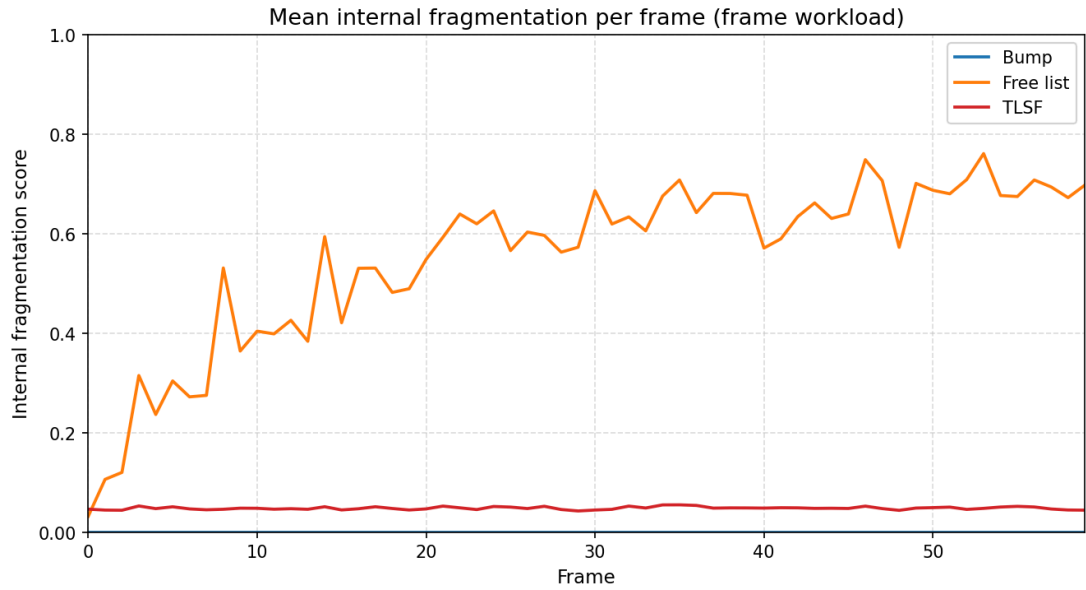


Figure 11: Mean internal fragmentation per frame (single lifetime workload).

On the single lifetime workload in Figure 11, the bump allocator recorded zero internal fragmentation as expected. The free list logged very high scores, but did not record any allocation failures. TLSF's score remained consistently low.

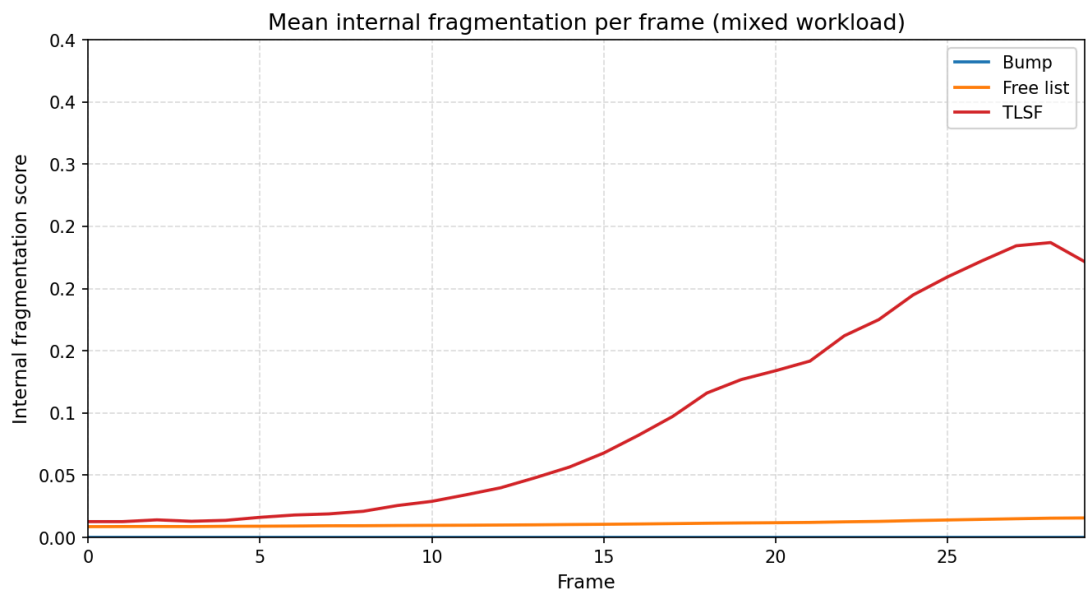


Figure 12: Mean internal fragmentation per frame (mixed workload).

However, on the mixed lifetime workload the tables turn. The free list's score remains near zero, and TLSF increased over the run and reported allocation failures shown in Figure 13.

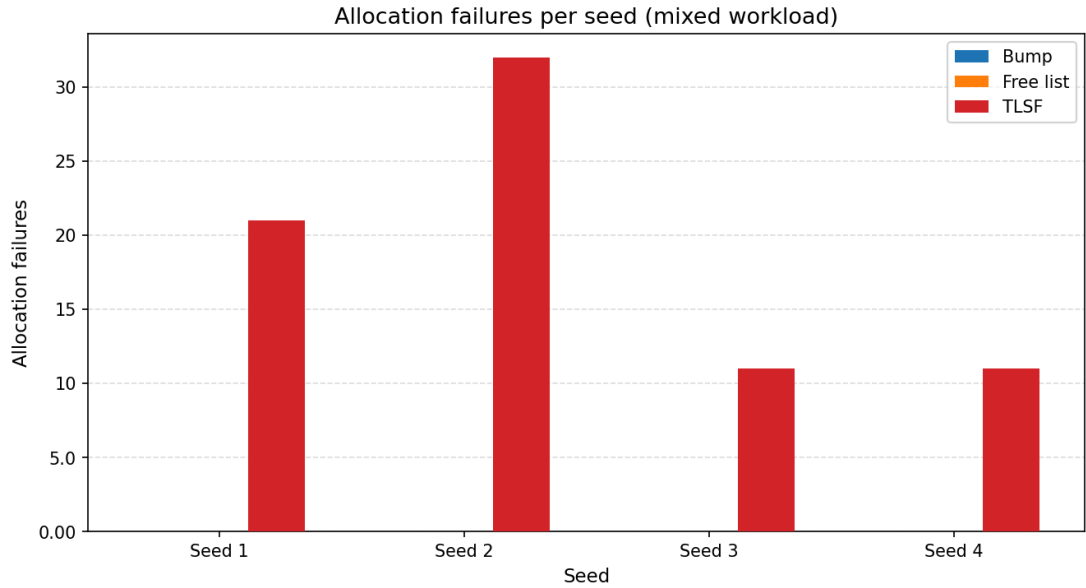


Figure 13: Allocation failures per seed (mixed workload).

4.4 Stack Unwinding Experiment

The experiment described in Section 3.7 was conducted. Both strategies recovered all call frames in the correct order.

5 Discussion

This chapter evaluates the results from Section 4 against the literature in Section 2. The allocator results are assessed across both workloads, and the stack unwinding experiment is evaluated against existing reliability concerns of DWARF unwinding.

5.1 Allocator Performance

5.1.1 Single Lifetime Workload

When working within one lifetime, this workload corroborates other literature which claims that general-purpose allocators achieve close to zero external fragmentation (Johnstone and Wilson, 1998). Both TLSF and the bump allocator recorded zero external fragmentation across all 60 frames shown in Figure 9. This is the expected outcome for all properly implemented allocators. When every object is freed at the end of each frame, all freed blocks should coalesce to a single free block between frames.

Unfortunately, the free list’s results are marred by a coalescing bug. During allocation, splitting places the next block header immediately after the payload and rounds it forward to the next 4 byte aligned address which leaves a tiny gap between 0 and 3 bytes. The block size only records the payload size, so the padding added after to align the next header is never accounted for. When the block is later freed, the coalescing check uses the recorded size to find where the next block should start and compares that address against the actual next free block. Because the recorded address is offset by the alignment gap, the two blocks do not appear

adjacent and are not merged. Each successive frame will leave behind small unmerged blocks which extend the $O(n)$ block scan.

TLSF's $O(1)$ latency guarantee (Masmano *et al.*, 2004) has been verified on the Wii. Its average allocation latency of roughly $3.5 \mu s$ remained consistent across its entire 60 frame run in Figure 7.

The bump allocator recorded zero fragmentation, but this reflects its inability to free individual objects, not allocator quality. TLSF was able to match bump's score while also keeping the flexibility to free individual objects for only an extra microsecond of latency.

The coalescing bug is also responsible for the free list's high internal fragmentation score on the frame workload displayed in Figure 11. Each unmerged free block contains a 12 byte header which is excluded from the allocator's free total, so as blocks accumulate across frames, those headers appear as used memory not serving a live allocation. TLSF's score remained consistently low since it coalesces correctly on frees.

5.1.2 Mixed Lifetime Workload

Introducing mixed lifetimes changes things because persistent objects prevent full coalescing between frames. In Figure 10, both TLSF and the free list began with similar external fragmentation scores but diverged by the end of the run.

The standout result from the mixed workload is TLSF's allocation failures. In Figure 13, TLSF had several allocation failures across all seeds, and the free list had none. This is due to TLSF's minimum split threshold. When splitting a block, any remainder under 152 bytes is absorbed into the allocated block instead of returned as a free block. Over 30 frames, these remainders accumulate inside live blocks and become unavailable for future requests.

The internal fragmentation results in Figure 12 are the inverse of the frame workload. The free list's score remains near zero even though the same coalescing bug being present. The persistent objects keep live requested bytes large, so the accumulated free block headers are a small fraction of committed memory. The bug is still a problem, but the larger amount of live payload bytes dampens it in the score.

The free list didn't fail because it has a much smaller minimum split threshold of 32 bytes, so it wastes fewer bytes per split. For workloads with persistent live objects and variable allocation sizes, this splitting policy may be preferable to TLSF even with its weaker latency guarantee.

Failure counts varied significantly across seeds, so no single seed is sufficient to understand the worst case behavior. This justifies the decision described in Section 3.3.2 to run all four seeds.

5.2 Stack Unwinding Experiment

Both DWARF and ABI unwinders recovered all nine call frames in the correct order from the crash dump.

The ABI chain approach is the simpler and safer of the two. It requires only that calling convention is followed and `r1` is maintained which are hard platform requirements (Zucker and Karhi, 1995). Torvalds advocated for this approach (Torvalds, 2012), and for this project’s toolchain it makes the most sense. DWARF’s advantage is portability. On other platforms without a reliable ABI back chain, it is the only option. However on PowerPC, the call chain is a hard requirement, so that advantage does not apply.

During development of the DWARF unwinder, an early implementation produced a trace of depth two rather than the correct nine. The DWARF table for each function contains how to locate the caller’s stack frame as a register plus an offset called the Canonical Frame Address (CFA). Zig enables frame pointers by default, and uses `r31` as the frame pointer register, so the rows covering code within a function’s body reference `r31` instead of `r1`. It only inspected the single row covering the current pc, which referenced `r31`, so it found no usable CFA and stopped. The trace terminated early with no error and looked like a reasonable result.

The correct approach shown in Listing 2 is to scan all FDE rows up to the current pc and use the last row where the CFA is represented as `r1 + offset`. Before `r31` is set up, each function emits a row using `r1 + offset`, and since `r31` equals `r1` at that point, that row’s CFA is correct.

```

while let Ok(Some(row)) = table.next_row() {
    if row.start_address() > current_pc as u64 {
        break;
    }
    let cfa = match row.cfa() {
        gimli::CfaRule::RegisterAndOffset { register, offset } => {
            let base = if *register == DWARF_REG_R1 {
                current_sp
            } else {
                continue; // skip r31 rows
            };
            (base as i64 + offset) as u32
        }
        _ => continue,
    };
}

```

Listing 2: FDE row scan in the DWARF unwinder (Collier, 2026)

The ABI unwinder proved its value beyond just providing an alternative trace. Because the ABI unwinder reads directly from the linked list of stack pointers maintained in r1 at runtime, it requires no debug tables and cannot be misled by the DWARF output. Running both and comparing them revealed that the DWARF unwinder had a bug. The DWARF output alone gave no indication that anything was wrong. This is the practical unreliability that the literature warns about (Torvalds, 2012; Bastian, Kell and Zappa Nardelli, 2019). In this case, it’s not that the encoded tables are wrong, but that an incorrect unwinder implementation fails silently in a way that looks plausible.

5.3 Limitations

The free list results are affected by a coalescing bug and should only judge the implemented allocator not the first-fit policy. The libc latency data has no measurement overhead, so it cannot be fairly compared with the other allocators. Both workloads are synthetic and do not reflect the full complexity of a real video game, and the results are limited to a single platform and four random seeds per workload.

6 Conclusion

The original project proposal intended to port Unreal Engine 5 to the Nintendo Wii (Collier, 2025a). The scope of this dissertation had to be smaller, and instead addressed two foundational questions to help achieve that goal. The first is finding which memory allocator is best suited to the Wii’s constrained physical memory. The second is whether postmortem debugging is practical on the Wii.

The allocator benchmark confirmed that TLSF’s $O(1)$ latency guarantee holds in practice on Wii hardware. Across both workloads, all three allocators achieved close to zero external fragmentation which is consistent with the existing literature

(Johnstone and Wilson, 1998). Internal fragmentation was also low except for the free list.

On the mixed lifetime workload, TLSF's minimum split threshold caused allocation failures across all seeds. TLSF's failures exclusively occurred under mixed lifetime conditions which might make it easier to miss in other tests. Mixed lifetime workloads are commonplace on Unreal Engine 5, so TLSF is not the right allocator for the job. The free list had no failures under the same workload. It is not production ready in its current form, but its smaller splitting threshold is the safest bet for now.

The stack unwinding experiment established that Zig does emit correct DWARF CFI, and that both ABI and DWARF unwinding are accurate for the Wii. And with the crash dump pipeline, a developer could recover a full call trace from a dump automatically sent from any user over the internet. This is how modern developers receive and act on crash reports. Without the pipeline, a crash would have to be reproduced manually which is very inefficient. The experiment also confirms that Zig is a viable option for Wii development.

The choice to use Zig contributes to developer efficiency. Every standard library function that allocates takes an explicit allocator, so the developer has easy control over memory. And Zig's toolchain did not require any additional configuration to output the correct DWARF debug information.

6.1 Future Work

This is not the quickest path to bringing Unreal Engine to the Wii, but each checkpoint will leave behind a meaningful open-source contribution to either the homebrew community or modern Unreal Engine development regardless of whether the final goal is ever reached. The author will also learn a lot along the way.

The first step is a native Zig replacement for libogc and devkitPro. It would give Zig programs direct access to the Wii's hardware without a dependency on C, and give the author a comprehensive understanding of the hardware and ABI before attempting to port Unreal.

The next checkpoint is rewriting Unreal Engine's build system in Zig. Starting with the Unreal Header Tool which generates C++ code mainly for reflection purposes. Then replacing the Unreal Build Tool which is the bespoke build system used by the engine. The author should gain a deep understanding of Unreal's toolchain after completing them. And each would be a useful open source contribution to modern Unreal Engine development on its own since they could speed up builds and CI jobs. As a stretch goal, the new build system could be extended to support Zig within Unreal Engine projects. The fork could also add additional desired

features such as support for C++ templated UObjects (“How to build a template/generic class derived from UObject,” 2014).

The final step after gaining all that knowledge is adding a Wii platform layer for Unreal Engine, and attempting to port an Unreal Engine project to the Wii.

List of References

Bastian, T., Kell, S. and Zappa Nardelli, F. (2019) “Reliable and fast DWARF-based stack unwinding,” *Proc. ACM Program. Lang.*, 3(OOPSLA), pp. 1–24. Available at: <https://doi.org/10.1145/3360572>.

Bhattacharai, A. (2024) “TLSF Allocator,” *ricefields.me*, 20 April. Available at: <https://ricefields.me/2024/04/20/tlsf-allocator.html> (Accessed: March 24, 2026).

Bryant, R.E. and O'Hallaron, D.R. (2015) *Computer Systems: A Programmer's Perspective*. 3rd ed. Pearson Education.

Collier, S. (2025a) *Porting Unreal Engine 5 to the Nintendo Wii*. Honors project proposal. Available at: <https://samcollier.dev/assets/HonorsProposal.pdf> (Accessed: April 27, 2026).

Collier, S. (2025b) “Profiling the Wii,” *samcollier.dev*, 1 December. Available at: <https://samcollier.dev/2025/12/01/TracyWii.html> (Accessed: April 19, 2026).

Collier, S. (2026) “wii-zig”. Available at: <https://github.com/goopey7/wii-zig> (Accessed: April 28, 2026).

Copetti, R. (2020) *Wii Architecture*. Available at: <https://classic.copetti.org/writings/consoles/wii/> (Accessed: April 28, 2026).

devkitPro (2020) “Getting Started”. Available at: https://devkitpro.org/wiki/Getting_Started (Accessed: March 22, 2026).

Durner, D. *et al.* (2019) “Experimental Study of Memory Allocation for High-Performance Query Processing,” *ADMS Workshop at VLDB 2019* [Preprint]. Available at: <https://arxiv.org/pdf/1905.01135>.

DWARF Debugging Information Format Committee (2017) *DWARF Debugging Information Format, Version 5*. Available at: <https://dwarfstd.org/doc/DWARF5.pdf> (Accessed: March 21, 2026).

Epic Games (2020) “Unreal Engine 5”. Available at: <https://www.unrealengine.com/> (Accessed: April 28, 2026).

Fitzgerald, N. and Craig, P. (no date a) “addr2line”. Available at: <https://crates.io/crates/addr2line> (Accessed: April 29, 2026).

- Fitzgerald, N. and Craig, P. (no date b) “gimli”. Available at: <https://crates.io/crates/gimli> (Accessed: April 29, 2026).
- Gay, D.E. (2001) *Memory Management with Explicit Regions*. Doctoral dissertation. Available at: <https://theory.stanford.edu/~aiken/publications/theses/gay.pdf> (Accessed: March 1, 2026).
- Green, A. (no date) “CrashDebug”. Available at: <https://github.com/adamgreen/CrashDebug> (Accessed: March 14, 2026).
- Gregory, J. (2018) *Game Engine Architecture*. 3rd ed. A K Peters/CRC Press.
- Hall, B. (2019) “Memory Allocation Strategies - Part 2: Linear/Arena Allocators,” *gingerBill*, 8 February. Available at: <https://www.gingerbill.org/article/2019/02/08/memory-allocation-strategies-002/> (Accessed: April 6, 2026).
- Hanson, D.R. (1988) *Fast Allocation and Deallocation of Memory Based on Object Lifetimes*. Technical report CS-TR-191-88. Available at: <https://www.cs.princeton.edu/techreports/1988/191.pdf> (Accessed: March 1, 2026).
- Johnstone, M.S. and Wilson, P.R. (1998) “The Memory Fragmentation Problem: Solved?,” *ACM SIGPLAN Notices*, 34. Available at: <https://doi.org/10.1145/286860.286864>.
- Linux Foundation (no date) *Exception Frames*. Available at: https://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Embedded/LSB-Embedded/ehframechpt.html (Accessed: April 27, 2026).
- Masmano, M. *et al.* (2004) “TLSF: A new dynamic memory allocator for real-time systems”. Available at: <https://doi.org/10.1109/EMRTS.2004.1311009>.
- Meer, A. (2008) “Epic: Unreal Engine 4 Console Exclusive,” *Rock Paper Shotgun*, 13 March. Available at: <https://www.rockpapershotgun.com/epic-unreal-engine-4-console-exclusive> (Accessed: April 28, 2026).
- Microsoft (2025) *Minidump Files*. Available at: <https://learn.microsoft.com/en-us/windows/win32/debug/minidump-files> (Accessed: March 20, 2026).
- Soltani, J. (2018) “Adopting Continuous Delivery (In Sea of Thieves)”. Available at: <https://youtu.be/cKfz2nEgaX8> (Accessed: April 29, 2026).
- Taudul, B. (no date) “Tracy Profiler”. Available at: <https://github.com/wolfpld/tracy> (Accessed: April 19, 2026).
- Torvalds, L. (2012) “Re: [RFC 0/5] kernel: backtrace unwind support,” *Linux Kernel Mailing List*. Available at: <https://lkml.org/lkml/2012/2/10/356> (Accessed: March 21, 2026).

Wilson, P.R. *et al.* (1995) “Dynamic storage allocation: A survey and critical review,” *Memory Management*. Edited by H.G. Baler. Berlin, Heidelberg: Springer Berlin Heidelberg. Available at: <https://www.cs.hmc.edu/~oneill/gc-library/Wilson-Alloc-Survey-1995.pdf> (Accessed: March 1, 2026).

Zucker, S. and Karhi, K. (1995) *System V Application Binary Interface: PowerPC Processor Supplement*. Revision A. 802–3334–10. Mountain View, CA: SunSoft. Available at: http://refspecs.linux-foundation.org/elf/elfspec_ppc.pdf (Accessed: March 22, 2026).

“devkitPPC” (no date). Available at: <https://wiibrew.org/wiki/DevkitPPC> (Accessed: April 15, 2026).

“Dolphin” (no date). Available at: <https://github.com/dolphin-emu/dolphin> (Accessed: October 12, 2025).

“How to build a template/generic class derived from UObject” (2014) *Unreal Engine Forums*. Available at: <https://forums.unrealengine.com/t/how-to-build-a-template-generic-class-derived-from-uobject/288125> (Accessed: April 28, 2026).

“sbrk.c” (no date). Available at: <https://github.com/devkitPro/libogc/blob/e206d4f337736d353e5e726242cd630dcd703e03/libogc/sbrk.c> (Accessed: April 28, 2026).

“Zig Language Reference” (2025). Available at: <https://ziglang.org/documentation/0.15.2/> (Accessed: March 22, 2026).