AGGRESSIVE LOOP UNROLLING ANALYSIS: REPORT

INTRODUCTION AND PROBLEM

Loop unrolling reduces branch overhead and improves instruction-level parallelism but is constrained by four competing microarchitectural limits: register availability, memory bandwidth, instruction cache capacity, and arithmetic intensity. This report presents a constraint-based analysis framework that automatically determines optimal unroll factors and supports aggressive unrolling through configurable multipliers.

For the test program analyzed, Loop #1 can be unrolled by factor 2 (conservative) or 4 (aggressive), while Loop #2 remains at factor 1-2 due to high register pressure and memory bandwidth saturation.

TECHNICAL APPROACH

The system operates in four stages:

STAGE 1 - DISASSEMBLY PARSING Regex-based extraction of instructions from objdump output, classifying each as arithmetic, memory, or branch operation.

STAGE 2 - LOOP DETECTION Identification of natural loops via backward branch analysis. The branch target marks the loop header; the branch instruction marks the loop exit.

STAGE 3 - CHARACTERISTIC ANALYSIS For each loop, count live registers (unique registers used), memory accesses (separated by load/store type), and instruction types.

STAGE 4 - CONSTRAINT EVALUATION Four competing constraints determine the unroll factor as their minimum.

CONSTRAINTS

REGISTER PRESSURE Formula: 32 / live_registers × multiplier Interpretation: Indicates how many times live values can be duplicated across MIPS 32 GPRs. Higher multiplier = more aggressive register reuse.

MEMORY BANDWIDTH Formula: 16 / (mem_accesses × 2) × multiplier Assumption: 16 bytes/cycle L1 cache bandwidth; each load/store costs 2 bytes Interpretation: Scaling factor for how many times memory ops can repeat without saturating L1 bandwidth.

INSTRUCTION CACHE Formula: 16 / loop_size × multiplier Assumption: 16 instructions fit in I-cache hotspot (64-byte line, 4 bytes/instruction) Interpretation: How many times loop can expand before exceeding hotspot size.

ARITHMETIC INTENSITY Formula: 8 if (arith_instr / mem_access) > 0.5, else 4 Reasoning: Compute-bound loops tolerate more unrolling; memory-bound loops are limited by bandwidth.

CORE ALGORITHM

For each loop, evaluate all four constraints independently, then select the minimum as the bottleneck. Round the result down to the nearest power of 2 to align with hardware behavior (prefetcher strides, branch prediction, memory alignment).

KEY INNOVATION: All constraints are scaled by configurable multipliers, enabling fine-grained control over aggressiveness without hardcoding specific factors. Conservative mode uses 1.0x multipliers; aggressive mode uses 2.0-4.0x multipliers.

PROGRAM ANALYSIS

LOOP #1 (8 instructions, 5 live registers, 2 stores) ~

CONSERVATIVE MODE: • Register Pressure: 32 / 5 = 6 • Memory Bandwidth: 16 / (2*2) = 4 • Instruction Cache: 16 / 8 = 2 ← BOTTLENECK • Arithmetic Intensity: 5 arith / 2 mem = 2.5 → limit 8 • OPTIMAL UNROLL FACTOR: 2

AGGRESSIVE MODE (2x multipliers): • Register Pressure: 32 / 5 × 2 = 12 • Memory Bandwidth: 16 / 4 × 2 = 8 • Instruction Cache: 16 / 8 × 2 = 4 ← NEW BOTTLENECK • Arithmetic Intensity: 8 • OPTIMAL UNROLL FACTOR: 4

LOOP #2 (10 instructions, 7 live registers, 2 loads + 1 store) ~

CONSERVATIVE MODE: • Register Pressure: 32 / 7 ≈ 4 • Memory Bandwidth: 16 / (3*2) ≈ 2 • Instruction Cache: 16 / 10 ≈ 1 ← BOTTLENECK • Arithmetic Intensity: 6 arith / 3 mem = 2.0 → limit 8 • OPTIMAL UNROLL FACTOR: 1

AGGRESSIVE MODE (2x multipliers): • Register Pressure: 32 / 7 × 2 ≈ 8 • Memory Bandwidth: 16 / 6 × 2 ≈ 4 • Instruction Cache: 16 / 10 × 2 ≈ 2 ← NEW BOTTLENECK • Arithmetic Intensity: 8 • OPTIMAL UNROLL FACTOR: 2

4. PERFORMANCE AND RISK ANALYSIS

4.1 LOOP #1: FACTOR 2 → 4 (CONSERVATIVE → AGGRESSIVE)

BENEFITS: • Branch overhead reduced by 50% (half as many backward branches) • Larger instruction window allows CPU to discover more independent operations • Memory bandwidth NOT saturated (4 bytes stores vs. 16-byte limit)

RISKS: • Register pressure increases from 5 to ~10 (approaches 32-register limit) • Unrolled loop expands from 8 to 32 instructions (potential I-cache eviction of other code)

ESTIMATED SPEEDUP: 1.3-1.5x (realistic for compute-bound arithmetic)

RECOMMENDATION: Safe to apply aggressive unrolling to Loop #1.

4.2 LOOP #2: FACTOR 1 → 2 (CONSERVATIVE → AGGRESSIVE)

BENEFITS: • Minor branch reduction (negligible impact for factor 1→2)

RISKS: • Register pressure increases from 7 to 14 (DANGEROUS on 32-register ISA) • High risk of register spilling to stack, causing severe performance degradation • Loop expands from 10 to 20 instructions (significant I-cache contention) • Memory bandwidth constraint becomes tighter (3 accesses × 2 = 6 bytes vs. 16-byte limit is OK, but L1 cache contention is real)

RECOMMENDATION: Keep Loop #2 at factor 1 (conservative only). Actual speedup from aggressive unrolling likely 0.9-1.1x due to spill overhead, making it a net loss.

KEY FINDINGS

(1) BOTTLENECK IDENTIFICATION Loop #1 is instruction-cache limited; Loop #2 is instruction-cache and register limited. These constraints cannot be scaled independently
   - they represent real hardware limits.

(2) AGGRESSIVE MODE SAFETY Multiplier approach enables controllable risk scaling. Users can selectively bypass constraints using command-line flags, but should validate speedup empirically.

(3) ARCHITECTURE DEPENDENCY Constraint values (32 registers, 16-byte bandwidth) are MIPS-specific. Embedded systems and superscalar processors require tuned parameters.

CONCLUSION

The aggressive unrolling framework successfully demonstrates how microarchitectural constraints can be relaxed to explore higher unroll factors while maintaining visibility into performance tradeoffs. The multiplier-based approach scales across architectures and provides safe defaults with expert-controllable overrides.