Improvements to Linear Scan register allocation

Alkis Evlogimenos (alkis)

April 1, 2004

1 Abstract

Linear scan register allocation is a fast global register allocation first presented in [PS99] as an alternative to the more widely used graph coloring approach. In this paper, I apply the linear scan register allocation algorithm in a system with SSA form and show how to improve the algorithm by taking advantage of lifetime holes and memory operands, and also eliminate the need for reserving registers for spill code.

2 Introduction

Linear scan register allocation was first introduced in [PS99] as a fast alternative to graph coloring register allocation. Later a slower variant of linear scan that produced better code, second-change bin backing was proposed in [THS98]. In another paper [MP02] a variation of linear scan was proposed that takes advantage of lifetime holes and handles preallocated registers.

In this paper, I further refine the linear scan algorithm. First, I introduce the concept of instruction slots to generalize the representation of live intervals in the context of lifetime holes and spill code and also provide an easy way to update this information incrementally. This new concept is described in more detail in Section 3.1. Then, starting from the lifetime hole implementation in [MP02], I add spill code fusion in memory operands when the target instruction set architecture supports it, as I further describe in Section 3.4. Finally, in Section 3.3, I show how to eliminate the need of reserved registers, increasing opportunities for register allocation and reducing spill code, especially in register poor ISAs.

Experimental evaluation of the above enhancements shows that they not only increase the quality of the generated code but also decrease the running time of the register allocator, as shown in more detail in Section 4.

3 Refinements to Linear Scan

3.1 Live Intervals

Central to linear scan register allocation is the notion of a live interval. In [PS99] a live interval is defined as in interval $[i, j]$ where $1 \leq i \leq j \leq N$ for which a variable is live in some ordering $1 \ldots N$ of the instructions. In this implementation I introduce the concept of instruction slots in order to handle spill code and reuse last *use* operands for allocating *def* operands in a generic way. Furthermore, live intervals are composed out of live ranges and thus can represent lifetime holes.

Each instruction has four slots. A *load* slot, a *use* slot, a *def* slot and a *store* slot. The *use* and *def* slots model the way machine instructions logically use their operands: all operands are first read, computation is performed and finally operands are written back in the register file. By modeling this in the live interval representation, adjacent live intervals that do not overlap, can reuse a register allocated to a last use for an instruction def. For example for this code:

```
0  A = ...
4  B = ...
8  C = A + B ;; last use of A
```

we will have the following intervals: $A = [3, 11)$, $B = [7, x)$, and $C = [11, y)$. Because of the different slots used for *use* and *def* the fact that the live ranges of A and C do not overlap can be naturally expressed in the interval representation without requiring further information. The original linear scan algorithm as presented in [PS99] does not have this problem since the definition is the start and last use is the end of the interval with no holes in between. Since this implementation takes advantage of lifetime holes, there is the need to distinguish between a last use and a break in a temporary's lifetime.

The *load* and *store* slots are used for handling spill code. Because this implementation does not reserve registers for spilling temporaries, there is the need to encode spill code information in the live interval. Adding the spill code incrementally will cause a renumbering of the instructions for the live intervals to be correct, which is an expensive operation. Instead, the live interval for the spilled temporary is updated to take into account the worst case spill code: a load before each *use* and a store after each *def*. After updating, the live interval can be used by the unmodified linear scan algorithm to allocate a spill register to the temporary. The need for the *load* and *store* slots is evident in the following example:

```
0  A = B + ...
4  C = A + D
```

Assuming all temporaries will be spilled we have the following live intervals: $A = [3, 4), [4, 5) = [3, 5)$, $B = [0, 2)$, $C = [7, x)$, and $D = [4, 6)$. Because the live intervals of A and D overlap, two spill registers are needed to allocate the code

fragment. Allocating B and C is trivial since their live intervals do not overlap with any other so we can use any of the two registers.

To build live intervals information, I use the live variable analysis provided by the LLVM compilation framework [LA04]. The pseudo-code for building live intervals using live variable information is provided in Figure 1.

```
1  for each basic block B in dfs order do
2      for each instruction I in B
3          let V <- register defined by I
4          let INT <- interval for register V
5          let I.n <- index of instruction
6
7          if (V is live through B)
8              add range [DEF-SLOT(I.n), B.end+1) to INT
9          else
10             let B.kill[V].n <- index of instruction that kills V
11             add range [DEF-SLOT(I.n), USE-SLOT(B.kill[V].n))
12
13         for all basic blocks B' that V is live through
14             add range [B'.start, B'.end+1) to INT
15
16         for all basic blocks B'' that V is killed
17             let I' <- instruction in B'' that kills V
18             add range [B''.start, USE-SLOT(I'.n)) to INT
19
```

Figure 1: COMPUTE-INTERVALS algorithm.

3.2 Register move coalescing

After live interval computation, register move coalescing is performed in a single pass over the instruction stream. The approach is similar to the one described in [MP02] in the sense that a union-find algorithm is used. The main difference is that joining live intervals is less aggressive since I only join on register moves whereas in [MP02] any two live intervals that do not overlap and are not preallocated are joined.

For every register move, the source and destination operands are observed. If the live intervals of the two operands do not overlap, and at least one of the operands is not preallocated, the live intervals are joined. When joining a union-find data structure is used, where both operands get a representative. In the case one of them is preallocated, the representative is that fixed register otherwise it is the temporary with the live interval starting earlier. I chose the temporary with the live interval starting earlier so that the ordering of the live intervals is preserved.

I opted to join only on register moves as this has the clear advantage of eliminating the move instruction. Joining any two intervals that do not overlap increases the number of overlapping intervals as the new joined interval overlaps the union of the overlapping intervals of the two joined intervals. Although I have not experimentally verified this, I expect the number of spills to increase with the more aggressive coalescing, as a result of increased register pressure. The pseudo-code for joining live intervals is shown in Figure 2.

```
1  for each basic block B in dfs order do
2      for each instruction I in B
3          if I is a move instruction
4              let DST be the destination register
5              let SRC be the source register
6              let DSTINT be the interval of DST
7              let SRCINT be the interval of SRC
8
9              if DSTINT does not overlap SRCINT
10                 join DSTINT with SRCINT
```

Figure 2: JOIN-INTERVALS algorithm.

3.3 Elimination of reserved registers

In both the original [PS99] and improved [MP02] linear scan algorithm there is no mention of how spill code is allocated. From their pseudo-code and algorithm description one can infer that registers are reserved for this purpose. Reserving 2 registers when allocating for a register poor ISA like the x86, causes an almost 30% loss of the available register file (2 out of 7 available registers). In order to maximize performance, especially for register poor architectures, I devised a way to reserve no registers for spill code. Instead when spilling, the algorithm backtracks, updates the spilled temporary's live interval and restarts. This makes the algorithm super-linear with a worst case quadratic complexity as it can potentially backtrack on every allocation attempt. Nevertheless, in practice the running time is still linear.

The relevant changes to the linear scan algorithm as refined in [MP02] are shown in the pseudo-code in Figure 4. When there are no more free registers (line 22), the temporary that overlaps with the current interval and has the minimum spill weight is spilled. It is worth noting that the current interval is a spill candidate as well. In the case the current interval is spilled no rollback is needed. Otherwise, all allocations need to be undone as shown in Figure 6. Intervals are popped out of the *handled* set and have their allocations undone until the start of the spilled interval. By performing the rollback, spilled temporaries are allocated in the context of linear scan and as such reserving registers for spill code is no longer required.

3.4 Fusing spill code into instructions

Some architectures, like the x86, allow memory instruction operands. In this implementation, I show how to take advantage of this feature. When updating the live interval of a spilled temporary, live ranges are only added when the memory reference cannot be fused in the instruction. For example in the following code sequence, assuming A will be spilled:

```
0  mov A, B
4  add C, B
8  add B, A
```

Before spilling A will have the following live interval: $[3, 11)$. After spilling without fusion the code sequence will remain the same but the new live interval will be: $[3, 5), [8, 10)$.

When spill code fusion is used and assuming only the first reference can be fused the code sequence will be modified to:

```
0  mov [stack slot], B
4  add C, B
8  add B, A
```

and A will have its live interval updated to: $[8, 10)$. If only the last reference can be fused, the code sequence will instead be modified to:

```
0  mov A, B
4  add C, B
8  add B, [stack slot]
```

and A's live interval will obviously be just $[3, 5)$. In the case both references are fused the resulting live interval will be empty and the code will have no references to the temporary A.

Although spill code fusion is more heavily used in the CISC ISAs, it is still useful for RISC ISAs as well. For example, register moves can have the source operand fused into a load or the destination operand fused into a store.

An interesting consequence of fusing memory operands into instructions is that it slows down backtracking. When a temporary is spilled, its updated live interval is pushed back into the *unhandled* set. Recall that the *unhandled* set contains the live intervals to be allocated in increasing start point. Since memory operand fusion can change the start point of a live interval (as is the case in the code sequence where only the move is fused), the spilled live interval cannot simply be inserted in the front of the *unhandled* set. In the current implementation the *unhandled* is scanned until a suitable point is found. This can be optimized to a binary search in the set, but it will still be slower than the constant time insertion needed when a live interval's start point never changes. Nevertheless, the reduced register pressure because of the use of memory operands on the x86, results in less spills and thus less rollbacks which in turn improves running time as shown in Figure 7.

4 Experimental results

For experimental results I used the SPEC suite of programs, compiled with the x86 back-end of the LLVM compiler infrastructure on an AMD Athlon XP 2400+ CPU. The register allocators used are:

- local: a local register allocator. It spills all temporaries on basic block boundaries and uses a simple last-use heuristic when spilling in the middle of a basic block.
- ls: a basic version of the linear scan register allocator, with lifetime holes and no reserved registers. This in not the original linear scan, since it uses backtracking to eliminate the need for reserved registers.
- ls+memop: same as the ls register allocator with the addition of spill code fusion in memory operands,
- ls+join: same as the ls register allocator with the addition of register move coalescing (join-intervals).
- ls+join+memop: same as the ls register allocator with the addition of both register move coalescing and spill code fusion in memory operands.

4.1 Register move coalescing

The internal representation of LLVM is in SSA form. After transforming the IR into machine code, SSA is preserved with the ϕ pseudo instructions. These pseudo instructions are eliminated before register allocation. Due to the simplicity of the algorithm used during ϕ-elimination, an excessive amount of register moves is generated. The $JOIN\text{-}INTERVALS$ pass as described in Section 3.2 attempts to rectify this by joining live intervals that do not overlap.

As shown in Table 1, the $JOIN\text{-}INTERVALS$ pass manages to successfully eliminate on average over 35% of the original live intervals, from programs in the SPEC2000 suite.

The reduction in the number of live intervals ends up speeding up compilation as shown in Figure 7. It also reduces the instruction count as a large number of register moves are eliminated as shown in Figure 8. On the other hand joining live ranges increases register pressure and as such may cause more spills. This is true for a few benchmarks, namely 255.vortex and 300.twolf in which the number of loads/stores added is greater when the ls+join is used in Figure 9. Even in this cases, the end result is an overall reduction in instruction counts.

4.2 Fusing spill code into instructions

By fusing spill code into instructions, register pressure is reduced and as a result better allocation is possible. Looking at Figure 10 we can see that using spill

Benchmark	Original	Joined	Ratio
177.mesa	82058	56039	0.68
179.art	2430	1637	0.67
183.equake	2597	1940	0.75
188.ammp	19460	14059	0.72
164.gzip	7939	4678	0.59
175.vpr	26138	15877	0.61
176.gcc	273574	151894	0.56
181.mcf	2065	1331	0.64
186.crafty	46567	27110	0.58
197.parser	21810	13100	0.60
252.eon	102570	80503	0.78
253.perlbmk	139692	89065	0.64
254.gap	115386	68589	0.59
255.vortex	65644	46026	0.70
256.bzip2	5211	3046	0.58
300.twolf	43959	29381	0.67
Total	957100	604275	0.63

Table 1: Live interval counts before and after register move coalescing.

code fusion often results in an increase in memory references. This is because code that used to be like the following before allocation:

```
0  add %R3, %R1
1  sub %R4, %R1
```

becomes:

```
0  mov %R1, [stack slot]
1  add %R3, %R1
2  sub %R4, %R1
```

when fusion is not used and:

```
0  add %R3, [stack slot]
1  sub %R4, [stack slot]
```

when fusion is used. The code using memory operands has more memory references but no explicit loads. Because the cache of modern x86 variants is almost as fast as registers and all subsequent references of the spilled temporary are guaranteed to be in the L1 cache, the performance of the fused code is expected to be similar to that of the non-fused code.

In Figure 9, we see a dramatic reduction of explicit loads/stores. This is both due to a reduction in register spills because of reduced register pressure and also a direct reduction of loads/stores because of spill code fusion.

Going back to Figure 7 we can see that compilation times are also improved even though spill code fusion slows down backtracking. This is mainly due to less backtracking because of decreased spilling. Also when the temporary corresponding to the current interval is spilled and all its references are fused, its live interval is removed and no backtracking is necessary.

4.3 Runtime Performance

The local register allocator is compared to ls+join+memop register allocator in both static and dynamic compilation in Table 2. For static compilation, the linear scan register allocator is clearly a better choice. Running time improvements range between 9% to 58%. For two benchmarks the linear scan allocator performs worse than the local allocator. I haven't yet identified the source of this slowdown.

For dynamic compilation, the running time performance is more varied. Because of the additional overhead of linear scan compared to the local allocator it is more often the case that linear scan performs worse than the local allocator. For programs that run for a non-trivial amount of time the cost of register allocation is amortized and thus linear scan outperforms the local register allocator. Thus, in the context of a JIT it may be a better choice to first allocate with a very fast register allocator like the local one, and identify hot functions which will later be reallocated by the slower but better linear scan register allocator.

5 Related Work

Linear scan register allocation was first introduced in [PEK97] and further documented in [PS99]. This initial incantation of the algorithm is extremely simple mainly because the quality of the code was good enough for the application and the authors concentrated more in compile time performance rather than run time performance. Consequently, lifetime holes are not exploited and there is no attempt to coalesce register moves. Also there is little information on how registers are allocated for spill code. The pseudo-code provided suggests that enough registers are reserved for use when spilling, so this makes it a relatively bad allocator for register poor architectures.

Second-chance bin-packing was proposed as an improvement to linear scan in [THS98]. As in this implementation second-chance bin packing takes advantages of lifetime holes, but it does not attempt to backtrack and reallocate after a spill. Instead, the live interval is split and the temporary is allowed to live in different registers in its lifetime. Because of this, their implementation needs to keep extra bookkeeping information on where a spilled temporary lives during its lifetime. In addition to the extra information, compensation code needs to be added on CFG edges so that allocation assumptions are maintained.

Benchmark	Static			JIT		
	local	ls	Ratio	local	ls	Ratio
177.mesa	2.776	2.744	1.01	6.121	6.023	1.01
179.art	5.663	4.863	1.16	5.926	5.441	1.09
183.equake	24.160	22.215	1.09	24.498	26.159	0.94
188.ammp	87.710	93.219	0.88	91.520	96.941	0.94
164.gzip	49.297	40.049	1.23	52.012	40.011	1.29
175.vpr	19.664	15.074	1.30	21.394	19.713	1.09
176.gcc	4.747	3.421	1.39	40.157	52.579	0.76
181.mcf	32.720	27.628	1.18	32.397	28.425	1.14
186.crafty	31.132	25.211	1.23	**	**	**
197.parser	8.456	6.261	1.35	10.591	9.383	1.12
252.eon	2.047	1.733	1.18	8.885	18.992	0.47
253.perlbmk	*	*	*	*	*	*
254.gap	6.745	4.727	1.41	12.526	13.725	0.91
255.vortex	9.394	10.025	0.94	17.845	25.053	0.71
256.bzip2	52.161	32.933	1.58	52.872	33.647	1.57
300.twolf	14.177	11.202	1.27	**	**	**

Table 2: Running time for static and JIT compilations comparing the local and ls+join+memop register allocators (* running times are too short to compare, ** benchmark run fails).

In a later study, a variant of linear scan that attempted to take advantage of SSA and work well with register constraints was proposed in [MP02]. This variant, like second-chance bin packing, takes advantage of lifetime interval holes. Furthermore it joins live intervals in a similar way as I presented in Section 3.2.

6 Conclusion and further work

In this paper, I presented several refinements to the linear scan register allocation algorithm. These refinements successfully improve the quality of code, while reducing compilation time. Improving code quality is always an advantage in any compiler framework. Reducing compilation complexity and computational resources makes this variant of linear scan a better candidate for use in a just in time compiler when compilation speed is of critical importance.

There are several directions to improving the algorithm even more. Given the relatively inexpensive way of updating live interval information, it will be interesting to see how a profile driven, region based linear scan register allocator can benefit a JIT compiler. Another improvement will be to introduce new temporaries for each def or use of a spilled temporary so that spill code is not constrained to use only one register.

An interesting study could be done comparing an aggressive graph coloring register allocator to this variant of linear scan. In previous studies, [PS99] and [THS98] it was shown that linear scan performs in the range of 10% of a good quality graph coloring register allocator. Given the proposed improvements, it is hoped that the gap will be further reduced but that has yet to be determined.

References

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[MP02] Hanspeter Mossenbock and Michael Pfeiffer. Linear scan register allocation in the context of ssa form and register constraints. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 229–246. Springer-Verlag, 2002.

[PEK97] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 109–121, 1997.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[THS98] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

```
UNHANDLED = intervals in increasing start point
FIXED = fixed intervals in increasing start point
ACTIVE = { }
INACTIVE = { }
HANDLED = { }
REGMAP = a map from INT.reg to machine registers

while (UNHANDLED not empty) do
    let INT <- interval with earliest start point

    foreach INT' in ACTIVE
        if (INT' expired at INT.start)
            remove INT' from ACTIVE
            mark REGMAP[INT'.reg] free
        else if (INT' not live at INT.start)
            move INT' to INACTIVE
            mark REGMAP[INT'.reg] free

    foreach INT' in INACTIVE
        if (INT' expired at INT.start)
            remove INT' from INACTIVE
        else if (INT' live at INT.start)
            move INT' to ACTIVE
            mark REGMAP[INT'.reg] used

    if (INT fixed)
        add INT in ACTIVE
        mark REGMAP[INT.reg] as used
    else
        ASSIGN-REG-OR-MEM(INT)
```

Figure 3: LINEAR-SCAN algorithm.

```
1  // INT <- the interval to allocate
2
3  save physical register free/use state
4
5  foreach INT' in ACTIVE
6      UPDATE-SPILL-WEIGHTS(INT')
7
8  foreach INT' in INACTIVE and FIXED
9      if (INT overlaps INT')
10         UPDATE-SPILL-WEIGHTS(INT')
11         mark REGMAP[INT'.reg] used
12
13 let R <- free register for INT.reg of NULL
14
15 restore physical register free/use state
16
17 if R != NULL
18     REGMAP[INT.reg] = R
19     mark R used
20     add INT to ACTIVE
21     add INT to HANDLED
22 else
23     let REG-OF-MIN-WEIGHT <- register with min spill weight
24     let MIN-WEIGHT <- weight of REG-OF-MIN-WEIGHT
25
26     if (INT.weight <= MIN-WEIGHT)
27         UPDATE-SPILLED-INTERVAL(INT)
28         if (INT is not empty)
29             insert INT at correct position in UNHANDLED
30     else
31         prepend INT to UNHANDLED
32         let EARLIEST-START <- INT.start
33
34         foreach INT' in ACTIVE and INACTIVE
35             if (regmap[INT'.reg] == REG-OF-MIN-WEIGHT &&
36                 INT' not fixed && INT' overlaps INT)
37                 EARLIEST-START = min(EARLIEST-START, INT'.start)
38                 UPDATE-SPILLED-INTERVAL(INT')
39
40         ROLLBACK(EARLIEST-START)
```

Figure 4: ASSIGN-REG-OR-MEM(INT) algorithm.

```
// INT <- interval to be updated

let INT' <- spilled interval for INT.reg

foreach instruction I in INT
    let I.n <- index of instruction
    let I.next.n <- index of next instruction

    fold as many references of INT.reg in I
    for each reference R of INT.reg
        let START <- R is use ? USE-SLOT(I.n) : DEF-SLOT(I.n)
        let END   <- R is def ? USE-SLOT(I.next.n) : USE-SLOT(I.n)
        add range [START, END)
```

Figure 5: UPDATE-SPILLED-INTERVAL(INT) algorithm.

```
// EARLIEST-START <- start of earliest interval affected

while (HANDLED not empty)
    let INT' <- last interval in HANDLED

    if (INT'.start < EARLIEST-START)
        break

    remove INT' from HANDLED
    if (INT' in ACTIVE)
        mark REGMAP[INT'.reg] as free

    remove INT' from ACTIVE or INACTIVE
    if (INT' is not empty)
        insert INT' at correct position in UNHANDLED

foreach INT' in HANDLED
    if (INT' expired after EARLIEST-START)
        add INT' to ACTIVE
        mark REGMAP[INT'.reg] used
```

Figure 6: ROLLBACK(EARLIEST-START) algorithm.

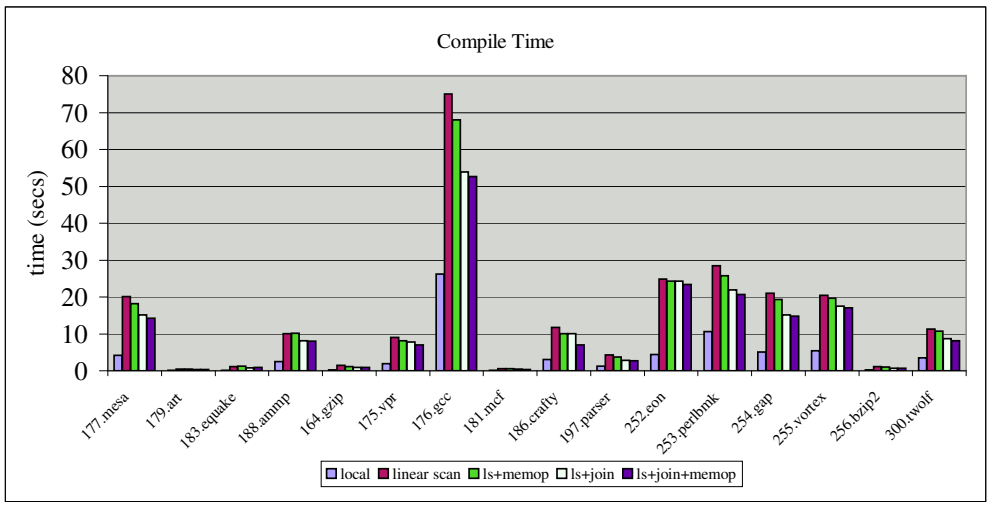

Figure 7: Compile Time.

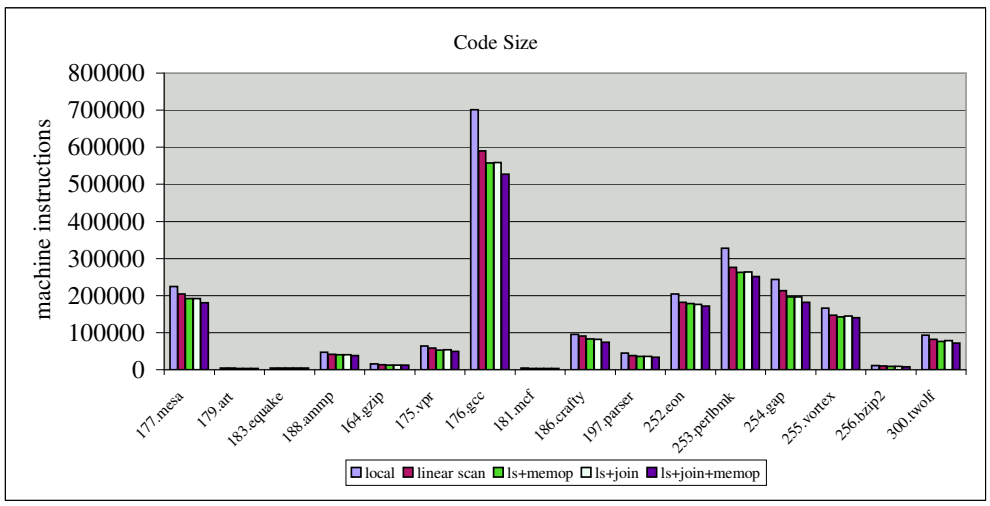

Figure 8: Code Size.

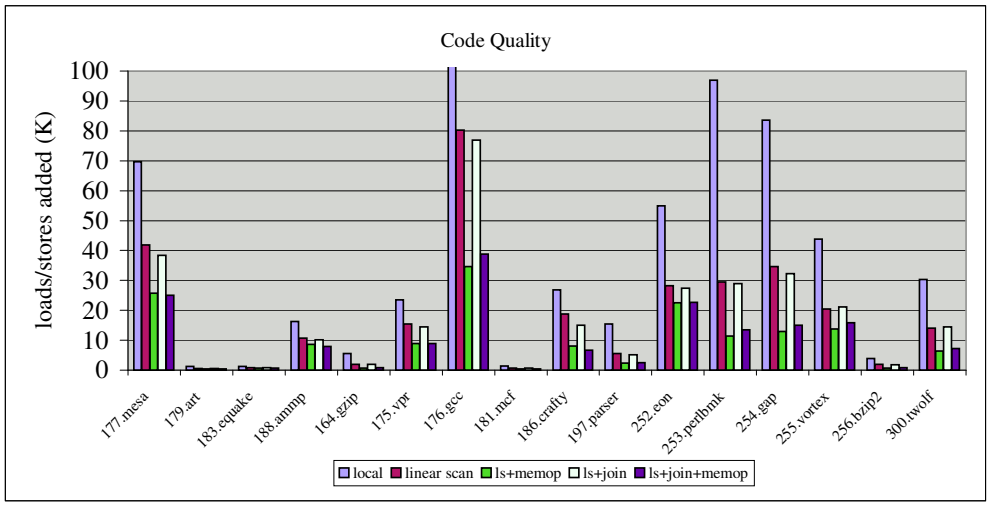

Figure 9: Code Quality - Loads/Stores added.

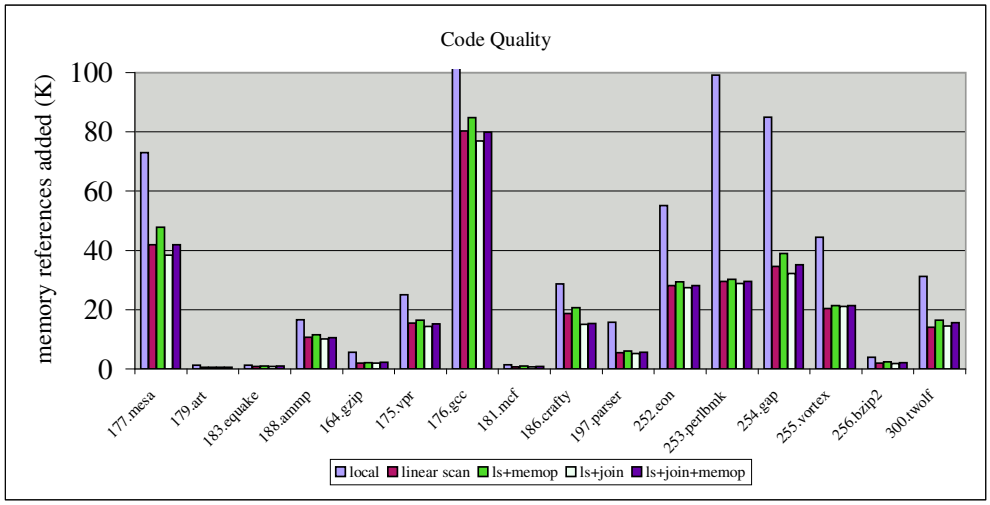

Figure 10: Code Quality - Added Memory References.