Extended Linear Scan: An Alternate Foundation for Global Register Allocation

Vivek Sarkar[1] and Rajkishore Barik[2]

[1] IBM T.J. Watson Research Center
vsarkar@us.ibm.com
[2] IBM India Research Laboratory
rajbarik@in.ibm.com

Abstract. In this paper, we extend past work on Linear Scan register allocation, and propose two *Extended Linear Scan* (*ELS*) algorithms that retain the compile-time efficiency of past Linear Scan algorithms while delivering performance that can match or surpass that of Graph Coloring. Specifically, this paper makes the following contributions:

- We highlight three fundamental *theoretical limitations* in using Graph Coloring as a foundation for global register allocation, and introduce a *basic* Extended Linear Scan algorithm, ELS_0, which addresses all three limitations for the problem of Spill-Free Register Allocation.
- We introduce the ELS_1 algorithm which extends ELS_0 to obtain a greedy algorithm for the problem of Register Allocation with Total Spills.
- Finally, we present experimental results to compare the Graph Coloring and Extended Linear Scan algorithms. Our results show that the compile-time speedups for ELS_1 relative to GC were significant, and varied from 15× to 68×. In addition, the resulting execution time improved by up to 5.8%, with an average improvement of 2.3%.

Together, these results show that Extended Linear Scan is promising as an alternate foundation for global register allocation, compared to Graph Coloring, due to its compile-time scalability without loss of execution time performance.

1 Introduction

Register allocation is the process of determining which variables (symbolic registers) should be held in physical machine registers at different program points and which should be *spilled*. *Register assignment* is the sub-process of identifying which specific machine registers should be used at different program points to hold which variables. The scope of register allocation may be *local* (restricted to a small region of a procedure, such as an innermost loop or an extended basic block), *global* (performed on an entire procedure) or *interprocedural* (performed across multiple procedures). Ever since its inclusion in the first compiler for FORTRAN five decades ago, register allocation has retained its role as one of the most important optimizations performed by compilers for high-level programming languages, and the algorithms used for register allocation have matured accordingly.

Starting with the seminal paper by Chaitin [5], the dominant approaches for global register allocation have been based on the idea of building an *Interference Graph* (IG)

S. Krishnamurthi and M. Odersky (Eds.): CC 2007, LNCS 4420, pp. 141–155, 2007.

for variables in a procedure, and employing *Graph Coloring* (GC) heuristics to perform the allocation. Significant advances have been achieved over these years through the introduction of new coloring, spilling, and coalescing heuristics based on the IG *e.g.,* [2, 3, 4, 6, 7, 12]. However, a key limitation that underlies all register allocation algorithms based on Graph Coloring is that the number of variables that can be processed by the register allocation phase in an optimizing compiler is limited by the size of the IG. The number of edges in the IG can be quadratic in the number of nodes in the worst case, and is usually observed to be super-linear in practice. The results in Section 4 show that the IG size is typically $O(n^{1.5})$ for n nodes. This non-linear complexity in space and time limits the code size that can be optimized and thereby has a damping effect on aggressive use of code transformations that can potentially increase opportunities for register allocation, such as *variable renaming*, *loop unrolling* and *procedure inlining*, but which also have the side effect of increasing the size of the IG. Finally, the non-linear complexity makes it prohibitive to use Graph Coloring for register allocation in just-in-time and dynamic compilers, where compile-time overhead contributes directly to run-time.

Recent work on *Linear Scan* algorithms [14, 16] has led to more efficient algorithms for global register allocation that use data structures with size that is linear in the number of variables. The results reported thus far suggest that Linear Scan should be used when compile-time space and time overhead is at a premium (as in dynamic compilation), but an algorithm based on Graph Coloring should be used when the best runtime performance is desired.

This paper extends past work on Linear Scan register allocation, and proposes two *Extended Linear Scan* (*ELS*) algorithms that retain the compile-time efficiency of past Linear Scan algorithms while delivering performance that can match or surpass that of Graph Coloring. The focus of this paper is on revisiting the premise that Graph Coloring is the most suitable foundation for global register allocation, and on evaluating *ELS* as an alternate foundation. Specifically, this paper makes the following contributions:

1. It highlights three fundamental *theoretical limitations* in using Graph Coloring as a foundation for global register allocation (Section 2.2).
2. It introduces the *basic* Extended Linear Scan algorithm, ELS_0 (Section 2.3), which addresses all three limitations for the problem of Spill-Free Register Allocation.
3. It introduces the ELS_1 algorithm (Section 3), which extends ELS_0 to obtain a greedy algorithm for the problem of Register Allocation with Total Spills (RATS).
4. It includes experimental results for eight SPECint2000 benchmarks to compare the Graph Coloring and Extended Linear Scan algorithms (Section 4). The results show that the space and time used by ELS_1 is significantly smaller than those used by GC – the compile-time speedups for ELS_1 relative to GC varied from 15× to 68×. In addition, the runtime performance improved by up to 5.8% for ELS_1 relative to GC, with an average improvement of 2.3%. This is a significant improvement over past Linear Scan algorithms which delivered compile-time efficiency but lagged behind Graph Coloring in runtime performance.

Together, these results show that Extended Linear Scan is promising as an alternate foundation for global register allocation, compared to Graph Coloring, due to its compile-time scalability without loss of execution time performance. Our expectation is that the coloring, spilling, and coalescing heuristics that have been developed over the

past decades as refinements to Graph Coloring, will be equally amenable to adaptation in an Extended Linear Scan foundation.

2 Spill-Free Register Allocation

This section introduces the Spill-Free Register Allocation (SFRA) problem as a theoretical foundation for comparing the fundamental differences between the Graph Coloring and Extended Linear Scan algorithms.

Spill-Free Register Allocation (SFRA): Given a set of symbolic registers, $\Re$, and k physical registers, determine if it is possible to assign each symbolic register $s \in \Re$ to a physical register, $reg(s,P)$ at each program point P where s is live. If so, report the register assignments, including any register-to-register copy statements that need to be inserted. If not, report that no feasible solution exists. □

Two key assumptions in the specification of the SFRA problem are as follows. First, two "program points" are defined for each instruction, i_k. i_k^- denotes the point at which the input operands of instruction i_k are read, and i_k^+ denotes the point at which the output operands of instruction i_k are written. Second, we assume that register allocation is performed as a separate pass from instruction scheduling — instruction scheduling considerations for register allocation [1, 9, 11, 13] are beyond the scope of this paper.

2.1 Basic Graph Coloring Solution to the SFRA Problem

Figure 1 summarizes the basic Graph Coloring algorithm for Spill-Free Register Allocation as described by Chaitin [5]. The correctness of this algorithm has also been established earlier in [5]. It is easy to see that the algorithm requires $O(|\Re|^2)$ space, since the interference graph can be quadratic in the number of symbolic registers. The major overhead in execution time occurs in constructing the interference graph in step 2, which takes $O(|\Re|^2)$ time. (This assumes that the liveness information in input 3 has been precomputed in a way such that each instance of the *simultaneously live* condition in step 2 can be computed in constant time. Otherwise, the execution time for step 2 could be larger than $O(|\Re|^2)$.)

2.2 Theoretical Limitations of Graph Coloring Solution

In this section, we summarize three fundamental theoretical limitations in using Graph Coloring as a foundation for global register allocation.

First, Graph Coloring is a *more limited problem than Register Allocation*. Transforming Register Allocation to Graph Coloring ensures that finding a k-coloring of an Interference Graph will lead to a feasible solution to the SFRA problem, but the converse is not true *i.e.,* it is *not necessary that an SFRA problem instance for which a solution exists can be transformed into a Graph Coloring problem for which a solution exists*. Consider the two examples in Figure 2, assuming that there are two physical registers available. In each case, a spill-free solution exists for the SFRA problem instance, but not for the Graph Coloring instance. In Example #2, the solution to the SFRA problem includes a register move instruction in the loop, but a solution based on Graph Coloring

Inputs:

1. *IR*, intermediate representation for program to be optimized.
2. $\Re$, set of *symbolic registers* that are candidates for allocation.
3. Liveness information that can be used to query if a symbolic register $s \in \Re$ is live at program point P
4. k, number of registers available for allocation.

Outputs:

1. *Success*, a boolean value that indicates whether or not a spill-free allocation was found for all symbolic registers in $\Re$.
2. If *Success* = *true*, then $reg(s)$ specifies the physical register assigned to symbolic register s, for all $s \in \Re$. (For basic Graph Coloring, no register moves are necessary because the register assignment will be the same for s at all program points.)

Algorithm:

1. Initialize an empty undirected *Interference Graph* (IG) with one node for each symbolic register.
2. **for each** pair of distinct symbolic registers, s_i and s_j, such that there exists a program point P where both s_i and s_j are simultaneously live **do**
   (a) Insert an edge in IG between node s_i and s_j
   end for
3. /* "Simplify" step in graph coloring heuristic */
   Initialize T := an empty stack;
4. Initialize IG' := copy of IG;
5. **while** $\exists$ a node s_i in IG' with degree $< k$ **do**
   (a) Delete node s_i from IG'
   (b) Push s_i on T
   end while
6. **if** IG' is now an empty graph **then** *Success* := *true*;
   else *Success* := *false*; **return**;
   end if
7. /* "Assignment" step in graph coloring heuristic */
   Initialize $reg(s_i)$:= *null* for each node s_i in IG;
   while T is non-empty **do**
   (a) s_i := pop(T);
   (b) $reg(s_i)$:= any register in $1 \ldots k$ that is distinct from $reg(s_j)$ for all nodes s_j that are adjacent to s_i in IG;
   end while

Fig. 1. Overview of Graph Coloring algorithm for Spill-Free Register Allocation

instead inserts a spill instruction in the loop. It is of course well known (*e.g.,* [2]) that *renaming* of variables or *live-range splitting* can be performed to obtain spill-free solutions with Graph Coloring for the examples in Figure 2. The observation being made here is that these transformations are orthogonal to Graph Coloring and are equally

SFRA problem instance #1: Find a spill-free register allocation for symbolic registers s_A, s_B, s_C in the program shown below, assuming that there are $k = 2$ physical registers available.

```
switch ( ... ) {
                case 0:                 case 1:                 case 2:
    i1:  sA := ...          i4:  sB := ...          i7:  sA := ...
    i2:  sB := ...          i5:  sC := ...          i8:  sC := ...
    i3:  ... := sA op sB    i6:  ... := sB op sC    i9:  ... := sA op sC
                break;                  break;                  break;
}
```

Graph Coloring problem instance: the Interference Graph is a complete clique for the three nodes s_A, s_B, s_C, and is therefore not 2-colorable.

SFRA solution: A simple solution exists to the above SFRA problem instance as follows, assuming that the two physical registers available are r_1 and r_2. No register moves are necessary for this solution:
$reg(s_A, [i_1^+, i_3^-]) = r_1$, $reg(s_B, [i_1^+, i_3^-]) = r_2$, $reg(s_B, [i_4^+, i_6^-]) = r_1$, $reg(s_C, [i_4^+, i_6^-]) = r_2$, $reg(s_A, [i_7^+, i_9^-]) = r_1$, $reg(s_C, [i_7^+, i_9^-]) = r_2$.

SFRA problem instance #2: Find a spill-free register allocation for symbolic registers s_A, s_B, s_C in the program shown below, assuming $k = 2$ physical registers.

```
i1:   sC := ...
      /* Start of loop */
i2:   sA := ...
i3:   ... := sC op ...
i4:   sB := ...
i5:   ... := sA op ...
i6:   sC := ...
i7:   ... := sB op ...
i8:   if sC <= 0 goto i10
i9:   goto i2
      /* End of loop */
i10:  ...
```

Graph Coloring problem instance: the Interference Graph is again a complete clique for the three nodes s_A, s_B, s_C, and is therefore not 2-colorable.
SFRA solution: The following solution exists to the above SFRA problem instance assuming that there are two physical registers available, r_1 and r_2. It also requires the insertion of a register-move instruction $r_1 := r_2$ between instructions i_8 and i_9.

$reg(s_C, [i_1^+, i_3^-]) = r1$, $reg(s_A, [i_2^+, i_5^-]) = r2$, $reg(s_B, [i_4^+, i_7^-]) = r1$, $reg(s_C, [i_6^+, i_8^+]) = r2$.

Fig. 2. Examples #1 and #2 for which a solution exists to the SFRA problem instance, but no solution exists to the corresponding Graph Coloring instance

applicable to Extended Linear Scan (*ELS*). Also, these transformations come at the cost of increasing the number of nodes and edges in IG, thereby further exacerbating the time and space complexity of register allocation based on Graph Coloring.

Second, the $O(|\Re|^2)$ space requirement for constructing the interference graph is a *scalability limitation* because the overhead of any register allocation algorithm based on Graph Coloring becomes prohibitively large when compiling procedures with a large number of symbolic registers (especially after transformations such as procedure inlining and loop unrolling are performed), or in scenarios where compiler space and time overhead is at a premium (as in dynamic compilation).

Third, Graph Coloring is an *NP-hard* optimization problem (without even the guarantee of a constant performance bound), whereas an exact solution can be obtained for SFRA in time that is linear in the number of live intervals for all symbolic registers as shown below in Section 2.3.

Together these limitations suggest that the Graph Coloring formulation may have made the global register allocation algorithm harder to solve than necessary, and thereby provide the motivation for our work on Extended Linear Scan.

2.3 Basic Extended Linear Scan Algorithm, ELS_0

In this section, we introduce the basic Extended Linear Scan algorithm, ELS_0, for the SFRA problem. ELS_0 addresses the three limitations of Graph Coloring outlined in the previous section, and also serves as the foundation for the ELS_1 algorithm. A summary of the ELS_0 algorithm can be found in Figures 3 and 4. The *live range* of a symbolic register s is represented by an *Interval Set*, $\mathscr{I}(s)$. Each interval, $[P,Q]$ in $\mathscr{I}(s)$ represents a range of program points at which s is live. The interval set is a precise representation

Inputs: Same as in Figure 1 (*IR*, $\Re$, liveness information, k).
Outputs:

1. *Success*, a boolean value as in Figure 1.
2. If *Success* = *true*, then $reg(s,[P,Q])$ specifies the physical register assigned to symbolic register s, for all program points in interval $[P,Q] \in \mathscr{I}(s)$. This *reg* mapping can be used to easily compute $reg(s,x)$ for any program point x where s is live, by identifying the interval in $\mathscr{I}(s)$ that contains x.
3. Modified *IR* with insertion of register-move instructions to handle cases when different physical registers may be assigned to the same symbolic register in different intervals.

Data structure initialization:

1. Interval Set $\mathscr{I}(s)$ for each symbolic register s
2. $\mathscr{I} = \cup_{s \in \Re} \mathscr{I}(s)$, the set of all intervals in the program (each interval is labeled with its symbolic register)
3. *IEP*, the set of interval endpoints in $\mathscr{I}$
4. *numlive* := 0
5. *count*[P] := 0, for each point in *IEP*

Fig. 3. Inputs, Outputs, Initialization for Extended Linear Scan algorithm ELS_0 for Spill-Free Register Allocation (see Figure 4)

Algorithm:

1. **for each** program point P in *IEP*, in increasing order **do**
   (a) **for each** interval $[O,P] \in \mathscr{I}$ **do** *numlive*−− **end for**
   (b) **for each** interval $[P,Q] \in \mathscr{I}$ **do** *numlive*++ **end for**
   (c) $count[P] := numlive$
   end for
2. **if** (∃ a program point P in *IEP* with $count[P] > k$) **then**
   (a) $Success := false$; **return**;
   end if
3. /* A feasible solution exists. Compute *reg* mapping and register-moves. */
   $Success := true$;
4. Initialize *avail* := set of all physical registers, $1 \ldots k$
5. **for each** program point P in *IEP*, in increasing order **do**
   (a) **for each** interval $[O,P] \in \mathscr{I}$ **do**
      i. $avail := avail \cup \{ r_j \}$, where r_j is the physical register that had been previously assigned to interval $[O,P]$
      end for
   (b) **for each** interval $[P,Q] \in \mathscr{I}$ **do**
      i. Let $s :=$ symbolic register corresponding to $[P,Q]$
      ii. Select a physical register r_j from *avail*, using the following heuristics:
         – If s is live at P, then prefer selecting r_j previously assigned to s, and
         – If program point P corresponds to a register-to-register copy statement of the form, $s := t$, then prefer selecting r_j reviously assigned to t.
      iii. $reg(s,[P,Q]) := r_j$ /* s is assigned r_j for all points in $[P,Q]$ */
      iv. $avail := avail$ - $\{ r_j \}$;
   end for
6. /* Insert register move instructions as needed. */
   for each program point P **do**
   for each program point Q that is a control flow successor to P **do**
   (a) Initialize M to be an empty set of move instructions
   (b) **for each** symbolic reg. s such that s is live at P and Q **do**
      i. **if** $(reg(s,P) \neq reg(s,Q))$ **then**
      insert a move instruction "$reg(s,Q) := reg(s,P)$" into set M **end if**
      end for
   (c) Treat the move instructions in M as a directed graph G in which there is an edge from move instruction m_1 to move instruction m_2 if m_1 reads the register written by m_2
   (d) Compute the strongly connected components (SCC's) of directed graph G
   (e) For each SCC, create a sequence of move and xor instructions to implement its register moves without the use of a temporary register, and insert these instructions on the control flow edge from P to Q (as part of Output 3 in Figure 3)
   end for
   end for

Fig. 4. Overview of Extended Linear Scan algorithm ELS_0 for Spill-Free Register Allocation (see Figure 3)

of liveness — as in [16], there may be "holes" in the interval set corresponding to program points where s is not live. We also define $\mathscr{I} = \cup_{s \in \mathfrak{R}} \mathscr{I}(s)$ to be the set of all intervals in the program, and *IEP* to be the set of *interval endpoints i.e.,* program points

that correspond to endpoints of intervals in $\mathscr{I}$. In the worst case theoretically, the size of $\mathscr{I}$ can be quadratic ($|\Re| \times |IR|$), where $\Re$ is the set of symbolic registers and *IR* is the intermediate representation of the procedure. The worst case can be achieved (for example) when each symbolic register is live at every other instruction in *IR* and therefore has $|IR|/2$ intervals. However, as shown in Section 4, in practice the average number of intervals per symbolic register is bounded by a small constant (≈ 2).

The outputs listed for the ELS_0 algorithm in Figure 4 are an extension of the outputs for the Graph Coloring algorithm in Figure 1. The boolean value, *Success*, indicates if a feasible SFRA solution can be found. The register map, *reg* is finer-grained for ELS_0 than for *GC* since it is capable of assigning different physical registers to different intervals in the Interval Set of a given symbolic register. The third output of the ELS_0 algorithm is a set of register-move instructions needed to support the register map. We assume that it is preferable to generate register-register moves than spill loads and stores on current and future systems, even for loads and stores that results in cache hits. This is because many processors incur a coherence overhead for loads and stores, compared to register accesses. Further, register-register moves can be optimized by efficient copy coalescing algorithms such as the one presented in [3].

We now outline how the ELS_0 algorithm addresses the three limitations for Graph Coloring discussed in Section 2.2:

1. The ELS_0 algorithm is guaranteed to find a feasible solution to an SFRA problem instance if and only if a feasible solution exists (Theorem 1).
2. The ELS_0 algorithm has a space requirement that is linear in the size of the input SFRA problem instance (Theorem 2).
3. The ELS_0 algorithm also has a time complexity that is linear in the size of the input SFRA problem instance (Theorem 2).

Theorem 1. *The* ELS_0 *algorithm always computes a correct solution for the SFRA problem.*

Proof: [Sketch] The ELS_0 algorithm returns *Success* = *false* only if there exists a program point *P* with $count[P] > k$ *i.e.,* with more than *k* symbolic registers that are live at *P* (which means that a spill-free register allocation is not possible). If the ELS_0 algorithm returns *Success* = *true* then $count[P] \leq k$ must be true at all program points $P \in IEP$. Therefore, there must be a physical register available in the *avail* set for each symbolic register at each program point. The register-move instructions inserted by step 6 ensure that a symbolic register's value is correctly carried across different physical registers that may be assigned to the same symbolic register. □

Theorem 2. *The* ELS_0 *algorithm takes* $O(|\mathrm{IR}| + |\mathscr{I}|)$ *space and* $O(|\mathrm{IR}| + |\mathscr{I}|)$ *time.*

Proof: [Sketch] It is easy to see that steps 1–5b take $O(|IR| + |\mathscr{I}|)$ space and time, assuming that all liveness information is precomputed (as in the Graph Coloring algorithm in Figure 1). Note that the size of the *avail* set is bounded by a constant, *k* (= number of physical registers). For step 6, the key observation is that there can be at most *k* register move instructions inserted on any control flow edge. □

3 Register Allocation with Total Spills

In this section, we extend the SFRA problem statement to allow for *total spills i.e.,* for identifying a subset of symbolic registers for which all accesses will be performed through memory instead of registers, with the goal of finding a solution with the smallest spill cost. Since the GCC compiler used to obtain our experimental results lacks support for pseudo-register live range splitting [8], an investigation of *live range splitting* and *partial spills* in the ELS framework is a subject for future work.

Register Allocation with Total Spills (RATS): Given a set of symbolic registers, $\Re$, k physical registers, and estimated execution frequency $freq[P]$ for each program point P, a register allocation with total spills consists of

1. a boolean function, $spilled(s)$, which indicates if s is to be spilled, and
2. for each symbolic register with $spilled(s) = false$, a register assignment, $reg(s,P)$ at each program point P where s is live.

There are two versions of the RATS problem, depending on whether or not insertion of register-move instructions is permitted:

- *regMoves* = *false*. In this version, no register-move instructions are allowed to be inserted, and the optimization problem is to find a register allocation with lowest *spill cost i.e.,* the lowest number of dynamic load and store instructions for the spilled symbolic registers, as determined by the $freq[P]$ values.
- *regMoves* = *true*. In this version, register-move instructions are permitted as in the SFRA problem statement, and the optimization goal is to minimize the combined overhead of *spill cost* and *register moves*. The relative weightage to be given to spill costs and register moves is architecture-specific. □

The SFRA problem in Section 2.3 is a *decision problem* which indicates whether a feasible spill-free register allocation can be obtained or not. In contrast, the RATS problem is an *optimization problem*, with the goal of minimizing spill costs (for the *regMoves* = *false* version) and a combination of spill costs and register-move cost (for the *regMoves* = *true* version). Note that it is trivial to obtain a feasible solution to the RATS problem by marking all symbolic registers as spilled — the challenge is to find a least-cost solution. It is well known that both versions of the RATS problem outlined above (with *regMoves* = *false* or *true*) are NP-hard.

The original algorithm by Chaitin addressed the *regMoves* = *false* version of the RATS problem by extending the algorithm in Figure 1 with a *priority function* that favored spilling symbolic register s with the smallest value of $totalSpillCost(s)/iDegree(s)$, where

$$totalSpillCost(s) = \sum_{\text{point } P \text{ w/ read of } s} freq[P] \; + \sum_{\text{point } Q \text{ w/ write of } s} freq[Q]$$

is the frequency-weighted sum of all read and write accesses to s, and $iDegree(s)$ is the degree of s in the simplified Interference Graph. There has been a very substantial amount of past work on augmenting and refining this priority function, starting with [6]. As mentioned earlier, we expect that these advanced spill heuristics designed for GC will be equally applicable to an ELS foundation.

Inputs:

1. *IR*, $\Re$, k, as in Figure 1.
2. $freq[P]$, estimated frequency for program point $P \in IEP$.
3. *regMoves*, version of the RATS problem to be solved.

Outputs:

1. *spill(s)*, indicates if symbolic register s was spilled.
2. If *spill(s)* = *false*, then $reg(s,P)$ specifies the physical register assigned to s at each program point P where s is live.
3. If *regMoves* = *true*, the *IR* is modified with insertion of register-move instructions as in Figure 4.

Data structure initialization:
Initialize $\mathscr{I}(s)$, $\mathscr{I}$, *IEP*, and *count* as in Figure 3, an empty stack T, and $spill(s)$:= *false* and $totalSpillCost(s)$ as defined in Section 3.

Fig. 5. Inputs, Outputs, and Initialization for Extended Linear Scan algorithm ELS_1 for Register Allocation with Total Spills (see Figure 6)

Figures 5 and 6 summarize our Extended Linear Scan algorithm for the RATS problem, ELS_1. This algorithm uses an input parameter, *regMoves*, to address both versions of the RATS problem. Figure 5 includes initialization steps from the ELS_0 algorithm, and also initializes $spill(s)$ and $totalSpillCost(s)$. Figure 6 contains the main ELS_1 algorithm. Step 1 in Figure 6 is the *Spill Identification* pass. It uses the observation from the SFRA problem that the only program points P for which spill decisions need to be made are those for which $count[P] > K$. The heuristic used in step 1a is to process these program points in decreasing order of $freq[P]$. As in Chaitin's Graph Coloring algorithm, Step 1b selects the symbolic register with the smallest value of $totalSpillCost(s)/iDegree(s,P)$ for spilling. A key difference with graph coloring is that this decision is driven by the choice of program point P, and allows for assigning different physical registers to the same symbolic register at different program points, when *regMoves* = *true*. We define $iDegree(s,P) = count[P] - 1$ to be the number of symbolic registers that interfere with s at some program point P with $count[P] > k$, when computed in step 1b of ELS_1 algorithm. After Step 1 has completed, a feasible register allocation is obtained with $count[P] \leq k$ at each program point P. The set of registers selected to be spilled are identified by $spill(s)$ = *true*, and are also pushed on to stack T. Step 2 is the *Spill Resurrection* pass. It examines the symbolic registers pushed on the stack to see if any of them can be "unspilled". Opportunities for resurrection arise when a later spill decision causes an earlier spill decision to become redundant.

Step 3 is the *Register Assignment* pass. If *regMoves* is *true*, the algorithm uses steps 4, 5, 6 of the ELS_0 algorithm in Figure 4. If *regMoves* is *false*, then we use a different register assignment algorithm that does not insert any register-move instructions. As indicated in step 3, the *regMoves* = *false* case can result in additional symbolic registers being spilled.

Algorithm:

1. /* Spill Identification. */
   while ($\exists$ a program point $Q \in IEP$ with $count[Q] > k$) **do**
   (a) $P :=$ program point in IEP with $count[P] > k$ and *largest* estimated frequency, $freq[P]$;
   (b) $s :=$ symbolic register s that is live at P, has $spill(s) = false$, and has the *smallest* value of $totalSpillCost(s)/iDegree(s,P)$;
   (c) Set $spill(s) := true$ and push s on stack T ;
   (d) **for each** program point $X \in IEP$ where s is live **do**
   $count[X] := count[X] - 1$;
   end for
   end while
2. /* Spill Resurrection. */
   while (stack T is non-empty) **do**
   (a) $s := pop(T)$;
   (b) **if** ($count[Q] < k$ at each point Q where s is live) **then**
   /* Resurrect symbolic register with *largest* spill cost. */
   i. Set $spill(s) := false$
   ii. **for each** program point X where s is live **do**
   $count[X] := count[X] + 1$; **end for**
   end if
   end while
3. /* Register Assignment. */
   if (*regMoves*) **then**
   Run steps 4, 5, 6 of the ELS_0 algorithm, restricted to symbolic registers s with $spill(s) = false$
   else /* Modified version of steps 4 and 5 in Figure 4. */
   for each program point P in IEP, in decreasing order of $freq[P]$ **do**
   (a) $avail :=$ set of physical registers that have not been assigned to a symbolic register that is live at P
   (b) **for each** symbolic register s that is live at P and does not have an assigned physical register, in decreasing order of $totalSpillCost(s)$ **do**
   i. Select a physical register r_j from $avail$ using the copy heuristic from step 5(b)ii in Figure 4.
   ii. **if** no register r_j was found **then** $spill(s) :=$ *true*;
   else $reg(s,*) := r_j$; $avail := avail - \{r_j\}$;
   end if
   end for
   end for
   end if

Fig. 6. Overview of Extended Linear Scan algorithm ELS_1 for Register Allocation with Total Spills (see Figure 5)

Theorem 3. *The ELS_1 algorithm always computes a correct solution to the RATS problem.*

Proof: [Sketch] The solution obtained after step 2 in the ELS_1 algorithm (Figure 6) is guaranteed to have $count[P] \leq k$ at each program point P. If *regMoves* = *true*, then the

RATS problem degenerates to SFRA in Step 3, and the correctness result from Theorem 1 holds. If *regMoves* = *false*, then steps 3(b)i and 3(b)ii ensure that each non-spilled symbolic register is assigned a physical register, or is spilled if no physical register is available, for each program point in *IEP*. □

Theorem 4. *The* ELS_1 *algorithm takes* $O(|\text{IR}| + |\mathscr{I}|)$ *space and* $O(|\text{IR}| + |\mathscr{I}|(log(count_{max}) + log|\text{IR}|))$ *time, where* $count_{max}$ *is the maximum value of* $count[P]$ *at any program point P.*

Proof: It is easy to see that the initialization in Figure 5 will take $O(|IR| + |\mathscr{I}|)$ space and $O(|IR| + |\mathscr{I}|)$ time. Note that the computation of $totalSpillCost(s)$ just takes $O(|IR|)$ time because each instruction in the intermediate representation can result in the increment of $totalSpillCost(s)$ for at most a constant number of symbolic registers.

For Step 1 (Spill Identification), the selection in step 1a of program point P with $count[P] > k$ and largest estimated frequency, $freq[P]$, contributes $O(|\mathscr{I}|log|IR|)$ time and step 1b contributes $O(|\mathscr{I}|log(count_{max}))$ time, assuming that a heap data structure (or equivalent) is used in both cases. Finally, step 2 (Spill Resurrection) and step 3 (Register Assignment) contribute at most $O(|\mathscr{I}|)$ time. □

4 Experimental Results

In this section, we report on experimental results obtained from a prototype implementation of Graph Coloring (as described in [10]) and ELS_1 in version 4.1 of the *gcc* compiler using the `-O3` option. Compile-time and execution time were measured on a POWER5 processor running at 1.9GHz with 31.7GB of real memory running AIX 5.3.

Experimental results are presented for eight out of twelve programs from v2 of the SPECint2000 benchmark suite. Results were not obtained for 252.eon because it is a C++ benchmark, and for the three other benchmarks — 176.gcc, 253.perlbmk, and 255.vortex — because of known issues [15] that require benchmark modification or installation of v3 of the CPU2000 benchmarks.

Table 1 summarizes compile-time overheads of the Graph Coloring and Extended Linear Scan algorithms. The measurements were obtained for functions with the largest interference graphs in the eight SPECint2000 benchmarks, using the `-O3 -finline-limit=3000 -ftime-report` options in gcc. It is interesting to note that the Interference Graph size, $|IG|$, typically grows as $O(|S|^{1.5})$, where as the number of intervals, $|\mathscr{I}|$ is always $\leq 2|S|$. This is one of the important reasons behind the compile-time efficiency of the Linear Scan and Extended Linear Scan algorithms. While it is theoretically possible for the number of intervals for a symbolic register to be as high as half the total number of instructions in the program (e.g., if every alternate instruction is a "hole" – which could lead to a non-linear complexity for *ELS*), we see that in practice the average number of intervals per symbolic register is bounded by a small constant (≈ 2). We see that the Space Compression Factor (SCF) = $|\mathscr{I}|/|IG|$ varies from 4.5% to 22.7%, indicating the extent to which we expect the interval set, $\mathscr{I}$, to be smaller than the interference graph, IG. Finally, the last two columns contain the compile-time spent in global register allocation for these two algorithms. For improved measurement accuracy, the register allocation phase was repeated 100 times, and the timing (in ms)

Table 1. Compile-time overheads for functions with the largest interference graphs in SPECint2000 benchmarks. $|S|$ = # symbolic registers, $|IG|$ = # nodes and edges in Interference Graph , $|\mathscr{I}|$ = # intervals in interval set, Space Compression Factor (SCF) = $|\mathscr{I}|/|IG|$, GC = graph coloring compile-time, ELS_1 = ELS_1 compile-time with *regMoves* = *true*.

Function	$\|S\|$	$\|IG\|$	$\|\mathscr{I}\|$	SCF	GC	ELS_1
164.gzip.build_tree	161	2301	261	11.3%	141.4ms	9.4ms
175.vpr.try_route	254	2380	445	18.7%	208.7ms	9.5ms
181.mcf.sort_basket	138	949	226	22.7%	6.8ms	0.1ms
186.crafty.InputMove	122	1004	219	21.8%	150.2ms	7.8s
197.parser.list_links	352	9090	414	4.5%	114.4ms	7.4ms
254.gap.SyFgets	547	7661	922	12.0%	118.8ms	8.0ms
256.bzip2.sendMTFValues	256	2426	430	17.7%	133.0ms	7.4ms
300.twolf.closepins	227	5105	503	9.8%	212.8ms	9.1ms

reported in Table 1 is the average over the 100 runs. While compile-time measurements depend significantly on the engineering of the algorithm implementations, the early indications are there is a marked reduction in compile-time when moving from GC to ELS_1 for all benchmarks. The compile-time speedups for ELS_1 relative to GC varied from 15× to 68×, with an overall speedup of 18.5× when adding all the compile-times.

Figure 7 shows the *SPEC rates* obtained for the Graph Coloring and ELS_1 algorithms, using the `-O3` option in gcc. Recall that a larger SPEC rate indicates better

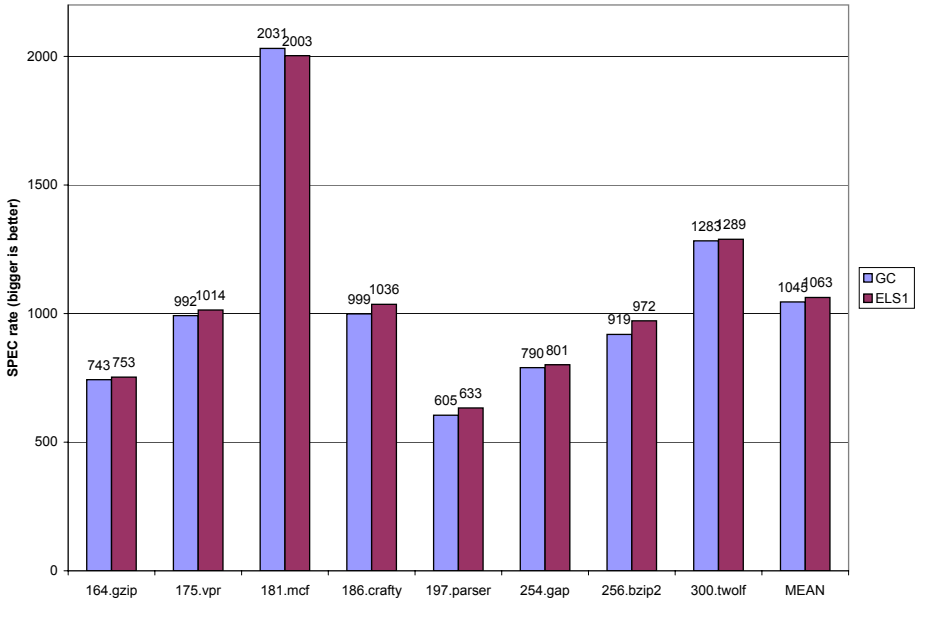

Fig. 7. SPEC rates for Graph Coloring and ELS_1 with *regMoves* = *true*

performance. In summary, the runtime performance improved by up to 5.8% for ELS_1 relative to GC (for `197.parser`), with an average improvement of 2.3%. There was only one case in which a small performance degradation was observed for ELS_1, relative to GC – a slowdown of 1.4% for `181.mcf`. These results clearly show that the compile-time benefits for Extended Linear Scan can be obtained without sacrificing runtime performance — in fact, ELS_1 delivers a net improvement in runtime performance relative to GC. Further, these measurements were obtained with *regMoves* = *true*, indicating that the extra register moves did not contribute a significant performance degradation. Runtime results were not obtained for the original Linear Scan algorithms, because it has already been established in prior work that their performance is inferior to that of Graph Coloring [14, 16].

5 Conclusions

This paper makes the case for using Extended Linear Scan as an alternate foundation to Graph Coloring for global register allocation. It highlighted three fundamental theoretical limitations with Graph Coloring as a foundation (Section 2.2). It introduced the basic Extended Linear Scan algorithm, ELS_0 (Section 2.3), which addressed all three limitations for the problem of Spill-Free Register Allocation (SFRA). It also introduced the ELS_1 algorithm (Section 3), which extended ELS_0 to obtain a greedy algorithm for the problem of Register Allocation with Total Spills (RATS). Finally, it included experimental results for eight SPECint2000 benchmarks to compare the Graph Coloring and Extended Linear Scan algorithms (Section 4).

The results show that the space and time used by ELS_1 is significantly smaller than those used by GC. The Space Compression Factor (SCF) = $|\mathscr{I}|/|IG|$ varied from 4.5% to 22.7%, and the compile-time speedups for ELS_1 relative to GC varied from 15× to 68×. In addition, the runtime performance improved by up to 5.8% for ELS_1 relative to GC, with an average improvement of 2.3%. This is a significant improvement over past Linear Scan algorithms which delivered compile-time efficiency but lagged behind Graph Coloring in runtime performance. Together, these results show that Extended Linear Scan is promising as an alternate foundation for global register allocation, compared to Graph Coloring, due to its compile-time scalability without loss of execution time performance. Directions for future work include further study of the trade-off between register-move instructions and spill load/store instructions, and support for region-based live range splitting.

References

[1] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for riscs. In *ASPLOS-IV*, pages 122–131, 1991.

[2] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[3] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 25–32, 2002.

[4] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 192–203, 1991.

[5] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[6] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 222–232, 1984.

[7] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[8] Vladimir N. Makarov. Yet another gcc register allocator. In *Proceedings of the GCC Developers Summit*, pages 148–157, May 2005.

[9] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. In *Technical Report STAN-CS-TN-95-22, Department of Computer Science, Stanford University*, 1995.

[10] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[11] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *MICRO 28*, pages 169–179, 1995.

[12] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 196–204, October 1998.

[13] Shlomit S. Pinter. Register allocation with instruction scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 248–257, 1993.

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

[15] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/cpu2000/issues/, 2006.

[16] 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.