A Decoupled non-SSA Global Register Allocation using Bipartite Liveness Graphs

Rajkishore Barik, Intel Labs
Jisheng Zhao, Rice University
Vivek Sarkar, Rice University

Register allocation is an essential optimization for all compilers. A number of sophisticated register allocation algorithms have been developed over the years. The two fundamental classes of register allocation algorithms used in modern compilers are based on Graph Coloring (GC) and Linear Scan (LS). However, these two algorithms have fundamental limitations in terms of precision. For example, the key data structure used in GC-based algorithms, the interference graph, lacks information on the program points at which two variables may interfere. The LS-based algorithms make local decisions regarding spilling, and thereby trades off global optimization for reduced compile-time and space overheads. Recently, researchers have proposed SSA-based decoupled register allocation algorithms that exploits the live-range split-points of the SSA representation to optimally solve the spilling problem. However, SSA-based register allocation often requires extra complexity in repairing register assignments during with SSA elimination and in addressing architectural constraints such as aliasing and ABI-encoding; this extra overhead can be prohibitively expensive in dynamic compilation contexts.

This paper proposes a decoupled non-SSA-based global register allocation algorithm for dynamic compilation. It addresses the limitations in current algorithms by introducing a Bipartite Liveness Graph (BLG) based register allocation algorithm that models the spilling phase as an optimization problem on the BLG itself and the assignment phase as a separate optimization problem. Advanced register allocation optimizations such as move coalescing, live-range splitting, and register class handling are also performed along with the spilling and assignment phases. In the presence of register classes, we propose a bucket-based greedy heuristic for assignment that strikes a balance between spill-cost and register class constraints. We present experimental evaluation of our BLG-based register allocation algorithm and compare it with production quality register allocators in JikesRVM and LLVM.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors
General Terms: Algorithms, Performance
Additional Key Words and Phrases: Decoupled register allocation, Bipartite graph, Coalesce graph, Basic and compound intervals

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Register allocation is an essential compiler optimization that has received much attention from the research community during the last five decades. Its relevance continues to increase with current trends towards energy-efficient processors in which some of
the burden of memory hierarchy management is shifting back from hardware to software. Two fundamental classes of register allocation algorithms have emerged over the years, Graph Coloring (GC) and Linear Scan (LS). Register allocation algorithms based on Graph Coloring (GC) [Chaitin et al. 1981; Briggs et al. 1994; George and Appel 1996; Park and Moon 1998; Smith et al. 2004], including more recent variants based on Static Single Assignment (SSA) form [Hack and Goos 2006], all use the Interference Graph (IG) as a primary data structure. Although the IG captures interferences among live ranges precisely, its lack of program point specific information can lead to imprecise result, especially for scenarios where insertion of additional move and exchange instructions can avoid spilling. On the other hand, register allocation algorithms based on Linear Scan (LS) e.g., [Traub et al. 1998; Poletto and Sarkar 1999; Wimmer and Mössenböck 2005; Sarkar and Barik 2007; Wimmer and Franz 2010] overcome the compile-time and compile-space overheads of GC algorithms, but do so at the expense of achieving poorer execution times than GC. The key reason for this is due to the lack of global information while making the spilling decisions. The primary goal of this paper is to address these limitations using a program point specific data structure called Bipartite Liveness Graphs (BLG).

A secondary goal of this paper is to simplify the implementation of the register allocator by decoupling the register spilling and register assignment phases in an optimizing back-end. This will allow the spilling phase to focus on spilling decisions and the assignment phase to focus on coalescing and physical register assignment decisions. While this form of decoupling has been performed for other register allocation algorithms in the past including [Appel and George 2001] and SSA-based register allocation algorithms [Hack and Goos 2006; Brisk 2006; Pereira and Palsberg 2009; Colombet et al. 2011], our approach is unique in its use of the Bipartite Liveness Graph (BLG) for the spilling phase and the Coalesce Graph (CG) for the assignment phase. The CG consists of both IR move instructions and register-to-register moves that arise from our BLG based allocation phase. In GC algorithms, the coupling between these phases is manifest in the integration of coloring and coalescing decisions, which can further compromise the effectiveness of the final solution and complicate the implementation of the allocator. These complications arise from non-trivial problems that must be addressed by the implementer in dealing with coalescing in traditional GC allocators. Further, register allocation for today’s architectures includes new challenges due to hardware features such as register classes, register aliases, pre-coloring, and register pairs. To produce high quality machine code, a register allocator must consider these hardware features in both the allocation and assignment phases.

Even though recent trends in register allocation is shifting towards decoupled SSA-based algorithms, there are known complexities in SSA elimination after register allocation [Brisk 2006; Pereira and Palsberg 2009] and in addressing architecture-level register aliasing and encoding constraints [Colombet et al. 2011] that suggest that the decoupled SSA approach will be challenging to use for dynamic compilation.

This paper addresses the register allocation challenges listed above by starting with a clean separation between the register spilling and register assignment phases. The spilling phase is modeled as an optimization problem on a new data structure called the Bipartite Liveness Graph (BLG). As we will see, the BLG is a more precise data structure than the IG. Assignment is modeled as a separate optimization problem that incorporates register-to-register moves and exchanges as alternatives to spilling, and handles move coalescing and register class constraints.

Specifically, we make the following contributions towards the above goals:

(1) We introduce a novel Bipartite Liveness Graph (BLG) representation as an alternative to the interference graph (IG) representation.
2. BIPARTITE LIVENESS GRAPH (BLG)

We start this section with some definitions. A program point can be split into two program points based on the values read and written at that program point [Sarkar and Barik 2007]:
Definition 2.1. Each program point \( p \) is split into \( p^- \) and \( p^+ \), where \( p^- \) consists of the variables that are read at \( p \) and \( p^+ \) consists of the variables that are written at \( p \).

\([x, y]\) is called a basic interval for variable \( v \) (denoted as \( BI(v) \)), for a basic interval of \( v \) if and only if for every program point, \( p \), such that \( p \geq x \) and \( p \leq y \) imply \( v \) is live at \( p \). Note that \( BI(v) \) does not include any hole. \( x \) and \( y \) denote the start and end points of \( BI(v) \) respectively. A compound interval for a variable \( v \) (denoted as \( CI(v) \)) consists of a set of basic intervals for \( v \). \( CI(v) \) can have holes. Let \( B \) denote the set of all basic intervals and \( C \) denote the set of all compound intervals in the program. Let \( L \) denote the set of start points and \( H \) denote the set of end points of all the basic intervals.

The number of simultaneously live symbolic registers at a program point \( p \) is denoted by \( numlive(p) \). \( MAXLIVE \) represents the maximum number of simultaneously live symbolic registers in any program point. A program point \( p \) is said to be constrained if \( numlive(p) > k \), where \( k \) is the total number of machine registers. In the presence of register classes, we call a program point \( p \) constrained if it violates any of the register requirements of any of the register classes of the symbolic registers that are live at \( p \).

Now we present a new representation, known as Bipartite Liveness Graph (BLG), that captures program point specific liveness information as an alternative to the interference graph. Formally,

**Definition 2.2. Bipartite Liveness Graph:** A bipartite liveness graph (BLG) is an undirected weighted bipartite\(^1\) graph \( G = (U \cup V, E) \), where \( V \) denotes all the basic interval end points\(^2\) in \( \mathcal{H} \), \( U \) denotes all the compound intervals in \( C \) and an edge \( e = (u, v) \in E \) indicates that the compound interval \( u \in U \) is live at the interval end point \( v \in V \). Each \( u \in U \) has an associated non-negative weight \( SPILL(u) \) that denotes the spill cost of \( u \). Similarly, each \( v \in V \) has an associated non-negative weight \( FREQ(v) \) that denotes the execution frequency of the IR instruction associated with basic interval end point \( v \).

It is obviously a waste of space to capture liveness information at every program point in \( V \) of \( BLG \). From a register allocation perspective, it suffices to consider only constrained program points corresponding to either the basic interval start points alone or end points alone but not both in \( V \). This is because spilling/assignment decisions only need to be taken at those points. Additional optimizations are also possible, e.g., if two interval end points have the same liveness information (i.e., same set of variables live), only one of them (but not both) needs to be added to the BLG for spilling decisions.

Figure 1 presents an example code fragment with its basic and compound intervals in Figure 1a) and the interference graph (IG) in Figure 1b). We observe that IG has a clique of size 3 due to the cycle comprising nodes \( c, d, \) and \( e \). Now consider a Graph Coloring register allocator that performs coalescing along with register allocation. Both aggressive [Chaitin et al. 1981] and conservative [Briggs et al. 1994] coalescing will be able to eliminate the move edges \( (a, c), (b, d), \) and \( (e, f) \) without increasing the colorability of the original interference graph. If we have two physical registers, we have to spill one of the coalesced nodes \( ac, bd, \) and \( ef \). The un-coalescing approach used in an optimistic coalescing technique [Park and Moon 1998] will be able to just spill one of the nodes involved in the cycle as it tries all possible combinations of assigning colors to individual nodes of a potentially spilled coalesced node. The points to note here

---

\(^1\)A bipartite graph is a graph whose vertices can be divided into two disjoint sets \( U \) and \( V \) such that each edge connects a vertex in \( U \) to one in \( V \).

\(^2\)The choice of interval end points is arbitrary. We could have used interval start points instead.
are that we can not color the IG using 2 physical registers and that opportunities for coalescing can be missed due to the inability to color certain nodes.

A closer look at the code reveals the fact that none of the program points have more than two variables live simultaneously. If this is the case, two questions come to mind:
1) Can we generate spill-free code with two physical registers that does not give up any coalescing of symbolic registers?
2) If the answer to the first question is yes, then why did Graph Coloring generate spill code and also miss the coalescing opportunity?

The answer to the first question is yes. The BLG with unconstrained interval end points for the example code is shown in Figure 1(c). This captures the fact that every basic interval end point in V has degree less than or equal to 2 indicating no more than two compound intervals are simultaneously live. (The BLG with constrained interval end points is empty in this case.) Let us name the two physical registers as \( r_1 \) and \( r_2 \). The following register assignment is possible:

\[
\text{reg}(1, 3) = r_1, \quad \text{reg}(2, 4) = r_2, \quad \text{reg}(4, 5) = r_2, \quad \text{reg}(3, 7) = r_1, \quad \text{reg}(6, 9) = r_2, \quad \text{reg}(9, 10) = r_2,
\]

\[
\text{reg}(8, 13) = r_1, \quad \text{reg}(11, 14) = r_2.
\]

This register assignment requires an additional register exchange operation since the register assignment for the basic intervals of both \( CI(c) \) and \( CI(d) \) were exchanged when the code after the \( if \) condition was executed. We need to insert an \text{exchg} \( r_1, r_2 \) instruction on the control flow edge between 4 and 13. As a result none of the coalescing opportunities in lines 3, 4, and 9 were given up during such an assignment.

Now let us try to answer the second question. Looking at the code fragment, we observe that at the program point 13, \( d \) interferes with two values of \( c \) that are assigned on lines 3 and 11. Similarly, \( c \) interferes with two values of \( d \) that are assigned on lines 4 and 8. During runtime, if the \( if \) branch is taken then assignments on lines 8 and 11 will be visible to the code following the \( if \) condition, otherwise assignments on lines 3 and 4 will be visible. This notion can not be precisely captured using the definition of \textit{live-ranges} in an interference graph unless we convert the program to SSA form or perform live-range splitting [Appel and George 2001]. Each of these approaches require additional complexities, e.g., the SSA-based approach needs to handle out-of-SSA translation by inserting extra copy statements.

The above example raises a question about the general approach of stating the global register allocation problem as the graph coloring problem on the IG. Even though the interference graph using live-ranges provides a global view of the program, it is less precise than a BLG with intervals.

Similar to GC, a Linear Scan (LS) register allocation algorithm (e.g., the LS algorithm implemented in Jikes RVM) when applied to Figure 1 will first spill one of \( CI(c) \), \( CI(d) \), or \( CI(e) \) compound intervals decided based on spill cost. If it decides to spill \( CI(e) \), then later on it will force another spill to one of \( CI(c) \), \( CI(d) \), or \( CI(f) \). This scenario is even worse than GC as it may spill more than one compound intervals. This problem arises in LS primarily due to the local decisions taken during the combined spilling and assignment phase.

3. OVERALL APPROACH
The overall register allocator presented in this paper is depicted in Figure . The first step in the allocator is to build data structures for basic intervals, compound intervals, and the Bipartite Liveness Graph (BLG). Then, the spilling is performed on the BLG to determine a set of compound intervals that need to be spilled as shown in the blocks for \textit{potential spill} and \textit{actual spill}. A combined phase of assignment and coalescing is then performed until all the symbolic registers are assigned physical registers or spilled. Next, register move and exchange instructions are added to the IR to produce correct code. Finally, spill code is added to the IR.
ALGORITHM 1: Greedy heuristic to perform spilling.

function GreedyAlloc()
    Input : Weighted Bipartite Liveness Graph $G = (U \cup V, E)$ and $k$ uniform physical registers
    Output: Set $T \subseteq U$ which needs to be spilled to ensure all interval end points $v \in V$ be unconstrained i.e., \( \forall b \in T, \text{spilled}(b) = \text{true} \)
    Stack $S := \emptyset$;
    //Potential spill selection
    $n := $ Choose a constrained node $n \in V$ with largest $FREQ(n)$;
    while $n \neq \text{null}$ do
        $s := $ Choose a compound interval $s \in U$ having an edge to $n$ and has smallest $SPILL(s)$;
        Push $s$ on to $S$; Delete edge $(s, n)$;
        $n := $ Choose a constrained node $n \in V$ having an edge to $s$ and has largest $FREQ(n)$;
        if $n == \text{null}$ then
            $n := $ Choose a constrained node $n \in V$ with largest $FREQ(n)$;
        end
        Delete all edges incident on $s$;
        Remove $s$ from $G$;
    end
    //Actual spill selection
    while $S$ is not empty do
        $s := \text{pop}(S)$;
        if $\forall n \in V, n$ becomes constrained by reverting $s$ and its edges in $G$ then
            for each basic interval $b$, in $s$ do
                spilled$(b) := \text{true}$; $T := T \cup \{b\}$;
            end
        end
    end
    return $T$

4. SPILLING USING BIPARTITE LIVENESS GRAPHS

In this section, we first describe an all-or-nothing approach for spills, that is, if a symbolic register is selected for spilling, every access of the symbolic register in the program will be replaced by a load or store instruction. Extending the $BLG$ to partial spills is described later in Section.

Definition 4.1.

Spill Optimization Problem: Given a $BLG$ with constrained end-points, $G$, and $k$ uniform physical registers, find a spill set $S \subseteq U$ and $G' \subseteq G$ induced by $S$ such that: (1) $\forall v \in V, v$ is unconstrained, i.e., $\text{DEGREE}(v) \leq k$; and (2) $\sum_{s \in S} \text{SPILL}(s)$ is minimized. For each compound interval $s \in S$ and basic interval $b \in s$, set $\text{spilled}(b) := \text{true}$. □
Given a BLG, the spill decision problem now reduces to an optimization problem whose solution ensures that no more than $k$ physical registers are needed at every interval end point, and at the same time, spills as few compound intervals as possible. Algorithm 4 provides a greedy heuristic that solves the spill optimization problem. Steps 3-11 choose Potential Spill candidates (as shown in Figure ) using a max-min heuristic. Each iteration of the loop alternates between largest frequency interval end point and smallest spill cost symbolic register. The alternating approach allows the option of completely unconstraining a high pressure region of program points before moving onto another. Steps 12-15 unspill some of the potential spill candidates resulting Actual Spill (as shown in Figure ) candidates. The unspilling step reverts a potential spill candidate and its edges back onto the BLG and verifies if the BLG becomes constrained after adding the potential spill candidate. If the BLG does not get constrained, then the symbolic register can be unspilled. Depending on the quality of potential spill candidate selection, the unspilling of spill candidates provides a way of rectifying the obvious spilling mistakes (akin to unspilling in Graph Coloring). The examination order of unspilling can have impact on final spilling decisions – currently we use a stack data structure that orders the potential spill symbolic registers in non-increasing spill cost.

One of the advantages of Algorithm 4 is that if a spill-free allocation exists, the algorithm is guaranteed to find an allocation without spills. On the other hand, if one works with an allocator based on graph coloring, it is an NP-hard problem to determine if a spill-free allocation exists. This seeming contradiction arises because BLG may require the insertion of register-copy instructions (described in Section ), whereas the standard graph coloring algorithm does not allow for this possibility. Prior work on SSA-based register allocation [Hack and Goos 2006; Brisk et al. 2005; Bouchez 2009] and on Extended Linear Scan [Sarkar and Barik 2007] independently established that the existence of a spill-free allocation can be determined in polynomial time, provided that extra register-copy instructions can be inserted. In the case of SSA-based register allocation, the extra copies arise from $\phi$-functions; in the case of Extended Linear Scan, they arise from the need to map from the register assignment for a symbolic register to another on a control flow edge. In both cases, the task of optimizing the additional copy instructions is a non-trivial problem.

**Theorem 4.2.**

Algorithm 4 ensures that every program point has $k$ or fewer symbolic registers simultaneously live.

**Proof:** First, we need to prove that Algorithm 4 makes every node $v \in V$ unconstrained. This is trivial as the algorithm continues to execute the while loop in Steps 4-11 until there are constrained nodes $v \in V$ in the BLG. This is guaranteed by steps 3, 7, and 9 in Algorithm 4. Second, we need to show that if no $v \in V$ is constrained, then every program point is unconstrained. We can prove this by contradiction. That is, if a program point, say $p^+$, is still constrained after all $v \in V$ are unconstrained, then we prove that such $p^+$ does not exist. Obviously, $p^+$ can not be an end point. Let $n^-$ represents the immediate next interval end point in the linear order of instructions from $p^+$. Thus, all program points from $[p^+, n^-]$ are constrained. This implies that $n^-$ must be constrained. This is a contradiction since $n^-$ is an interval end point and is constrained. Hence proved. □

**Theorem 4.3.** Given the bipartite liveness graph, Algorithm 4 requires $\mathcal{O}(|H| + \max(0, (MAXLIVE - k)) + |C|)$ time.
Proof: Every interval end point in $H$ is traversed at most $\text{MAXLIVE} - k$ number of times to make it unconstrained. To make an interval end point unconstrained, we need to visit all its neighbor and choose a minimum spill cost compound interval. This requires, at most, $|C|$ edge visits. □

4.1. Bipartite Liveness Graphs with Partial Spills

We now extend the register allocation problem for the BLG to allow for partial spills i.e., for splitting a symbolic register so that it can be assigned to registers at some program points, and accessed from memory at other program points. Live-range splitting has also been considered quite extensively in past work, though often with inconclusive results on the benefits of splitting. We consider a special case of partial spills, namely that of identifying one basic interval of a symbolic register for spilling. More general splitting of live ranges (as in [Bergner et al. 1997] say) is a subject for future work.

For partial spills, we define $\text{SPILLBI}$ for a basic interval that captures the spill cost of a basic interval including the cost for additional loads and stores for partial spilling. The problem statement can be summarized as follows.

Definition 4.4. Bipartite Liveness Graph with Partial Spills: A bipartite liveness graph with partial spills (BLGP) is a undirected weighted bipartite graph $G = (U \cup V, E)$, where $V$ denotes all the basic interval end points in $H$, $U$ denotes all the basic intervals in $B$ and an edge $e = (u, v) \in E$ indicates that the basic interval $u \in U$ is live at the interval end point $v \in V$. Each $u \in U$ has an associated non-negative weight $\text{SPILLBI}(u)$ that denotes the spill cost of $u$. Similarly, each $v \in V$ has an associated non-negative weight $\text{FREQ}(v)$ that denotes the execution frequency of the IR instruction associated with basic interval end point $v$. □

Definition 4.5. Register Allocation Optimization Problem with Partial Spills: Given a BLG with constrained end-points, $G$, and $k$ uniform physical registers, find a spill basic interval set $S \subseteq U$ and $G' \subseteq G$ induced by $S$ such that: (1) $\forall v \in V$, $v$ is unconstrained, i.e., $\text{DEGREE}(v) \leq k$; and (2) $\sum_{s \in S} \text{SPILLBI}(s)$ is minimized. For each basic interval $s \in S$, set $\text{spilled}(s) := true$. □

Algorithm 4 can be extended easily to support partial spills. That is, steps 3-11 can be modified to choose potential spill basic intervals instead of compound intervals using the original max-min heuristic. Similarly, the unspilling in steps 12-15 rectifies the spilling decisions by resurrecting basic intervals instead of compound intervals.

5. ASSIGNMENT USING REGISTER MOVES AND EXCHANGES

The spilling phase ensures that every program point needs $k$ or fewer physical registers. In this section, we first describe how assignment for basic intervals can be performed by possibly adding extra register moves/exchanges to the IR without spilling any symbolic registers.

5.1. Spill-Free Assignment

Definition 5.1. Spill-free Assignment: Given a set of basic intervals $b \in B$ with $\text{spilled}(b) = false$, and $k$ uniform physical registers, find register assignment $\text{reg}(b)$ for every basic interval, $b \in B$, including any register-to-register copy or exchange instructions that need to be inserted in the IR. □

The algorithm to perform register assignment for basic intervals is provided in Algorithm . The algorithm sorts the basic intervals in increasing start points. Steps 4-11
perform assignment to basic intervals using an *avail* list of physical registers. The assignment to a basic interval first prefers getting the physical register that was previously assigned to another basic interval of the same compound interval (as shown in Step 7). This avoids the need for additional move/exchange instructions. However, in cases where the already assigned physical register is unavailable, we assign a new available physical register (as shown in Step 10). Assigning such a new physical register may produce incorrect code without additional move/exchange instructions on certain control flow paths.

Steps 12-20 of Algorithm create a list of move instructions that need to be inserted on a control flow edge. These move instructions form the nodes of a directed anti-dependence graph $D$ in Algorithm. The edges in $D$ represent the anti-dependence between a pair of move instructions. Steps 5-10 of Algorithm add the anti-dependence edges to $D$. A strongly connected component (SCC) search is performed on $D$ to generate efficient code using *exchange* instructions for SCC’s of size 2 or more (as shown in steps 11-18). The nodes in a SCC are collapsed to a single node with exchange instructions. Finally, a topological sort order of $D$ produces the correct code for a control flow edge $e$.

ALGORITHM 2: Assignment using register moves and exchange instructions

```plaintext
function RegMoveAssignment()
    Input : IR, Set of basic intervals $b \in B$ with spilled($b$) = false and $k$ uniform physical registers
    Output: $\forall b \in B$, return the register assignment $reg(b)$ and any register moves and exchange instructions
    $M := \phi$;
    $avail$ := set of physical registers;
    for each basic interval $b := [x, y]$, in increasing start points i.e., $\mathcal{L}$ do
        for each basic interval $b' := [x', y']$ such that $y' < x$ do
            $avail := avail \cup reg(b')$;
            end
        end
        r := find a physical register $p \in avail$ that was assigned to another basic interval of the same compound interval;
        if $r == null$ then
            Assert avail is not empty;
            r := find a physical register $p \in avail$;
        end
        $reg(b) := r$; $avail := avail - \{r\}$;
    end
    for each control flow edge, $e$ do
        for each compound interval $c \in C$ that is live at both end points of $e$ do
            $b_1 :=$ basic interval of $c$ at the source of $e$;
            $b_2 :=$ basic interval of $c$ at the destination of $e$;
            if $b_1 \neq null$ and $b_2 \neq null$ then
                $r_1 := reg(b_1)$; $r_2 := reg(b_2)$;
                if $r_1 \neq r_2$ then
                    m := generate a new move instruction that moves $r_1$ to $r_2$ i.e., mov $r_2, r_1$;
                    $M := M \cup \{m\}$;
                end
            end
        end
        GenerateMoves(IR, M,e);
    end
    return $T$ and IR
```

**ALGORITHM 3:** Insertion of move and exchange operations on a control flow edge

```text
function GenerateMoves()
  Input : IR, Set of move instructions M and a control flow edge e
  Output: Modified IR with register move and exchange instructions added
  D := φ /
  for m₁ ∈ M do
    Add a node for m₁ in D;
  end
  for m₁ ∈ D do
    for m₂ ∈ D and m₂ != m₁ do
      s₁ := source of the move instruction in m₁;
      d₂ := destination of the move instruction in m₂;
      if s₁ == d₂ then
        Add a a directed edge (m₁, m₂) to D;
      end
    end
  end
  S := Find strongly connected components in D;
  for each s ∈ S do
    Collapse all the nodes in s to a single node n in D;
    while number of move instructions in s > 1 do
      m₁ := Remove first move instruction from s;
      m₂ := Next move instruction in s;
      x := Generate an exchange instruction between the destinations of m₁ and m₂;
      Append x to the instructions of n;
    end
  end
  for each node n in D in topological sort order do
    Add the move or exchange instructions of the node n to the IR along the control flow edge e;
  end
  return Modified IR
```

**Lemma 5.2.** The assertion on line 9 of Algorithm never fails.

**Proof:** Follows from the fact that every interval end point has no more than k symbolic registers simultaneously live. □

**Theorem 5.3.**
Spill-free assignment takes $O(|E| \times (|C| + |K|^2))$ space where $E$ represents the control flow edges in a program and $K$ represents the available physical registers.

**Proof:** Additional space requirement in assignment phase is due to the anti-dependence graph $D$. For every control flow edge $e \in E$, in the worst case we need to insert $|C|$ number of register-to-register move instructions. These are the number of nodes in $D$. The number of edges in $D$ are bounded by the square of physical registers $K$, i.e., it represents all possible anti-dependences between all possible pairs of physical registers. Hence the overall space complexity is $O(|E| \times (|C| + |K|^2))$. □

**Theorem 5.4.**
Spill-free assignment takes $O(|B| + (|E| \times (|C| + |K|^2)))$ time.

**Proof:** Similar in nature to the proof for Theorem 4.7. □
5.2. Assignment with Move Coalescing and Register Moves

Move coalescing is an important optimization in register allocation algorithms that assigns the same physical registers to the source and destination of an IR move instruction when possible to do so. The register assignment phase must try to coalesce as many moves as possible so as to get rid of the move instructions from the IR. As we saw in the preceding section, additional register moves may be inserted in the assignment phase instead of spilling. Note that move coalescing approaches using aggressive [Chaitin et al. 1981], conservative [Briggs et al. 1994], and optimistic [Park and Moon 1998] techniques are shown to be NP-complete by [Bouchez et al. 2007]. In this section, we first present a coalesce graph that models both the IR move instructions and register-to-register moves. Then, the register assignment phase on the coalesce graph is formulated as an optimization problem that tries to maximize the number of move instructions removed after assignment. We provide a greedy heuristic to solve it.

Definition 5.5.
A Coalesce Graph (CG) is an undirected weighted graph $G = (V, E)$ where $V$ represents the basic intervals in $B$ and an edge $e \in V \times V$ corresponds to the following two types of move instructions between a pair of basic intervals:

1. $E_m$: the move instructions already present in the IR. The weight of such an edge $W(e)$ is the estimated frequency of the corresponding move instruction.
2. $E_r$: the move instructions that need to be added on control flow edges for which the two interval end points have different register assignments for the same compound interval. The weight of such an edge $W(e)$ is the estimated frequency of the control-flow edge on which the move instruction is added.

Definition 5.6.
Assignment Optimization Problem: Given a set of basic intervals $b \in B$ with $spilled(b) = false$, $CG = (V, E = \{E_m \cup E_r\})$, IR, and $k$ uniform physical registers, find register assignment $reg(b)$ for every basic interval $b$ such that the following objective function is minimized:

$$\sum_{e \in E, e = (b_1, b_2) \land reg(b_1) \neq reg(b_2)} W(e)$$

The assignment guides which additional register-to-register copy or exchange instructions need to be inserted in the IR.

Algorithm presents a greedy heuristic to select a physical register for a basic interval $b$ given the coalesce graph and the available set of physical register $avail$. $avail$ is updated as basic intervals expire. $Map$ is a data structure that maps a physical register to a cost. Steps 3-7 find the physical registers and their associated costs that are already assigned to the neighbors of $b$ in the coalesce graph (similar to the idea of biased coloring [Briggs et al. 1992]). Our approach takes into account the edges in $E$, due to register-to-register moves. The greedy heuristics chooses a physical register $reg(b)$ with maximum cost, i.e., the benefit of assigning the physical register to basic interval $b$.

Theorem 5.7.
Register assignment using Algorithm requires $O(|B| + |IR| + (|C| \times max_c))$ space where $max_c$ denotes the maximum number of basic intervals in a compound interval.

Proof: The additional space requirement is due to the coalesce graph $CG$ containing $|B|$ number of nodes. $E_m$, in the worst case ends up creating $|IR|$ edges. $E_r$ adds edges
ALGORITHM 4: Greedy heuristic to choose a physical register that maximizes copy removal

function GetPreferredPhysical()
    Input: A basic interval \( b \in B \), coalesce graph \( G = (V, E = \{E_m \cup E_r\}) \) and a set avail currently available uniform physical registers
    Output: Find the assignment \( \text{reg}(b) \)
    Map := \( \phi \);
    //Maximize the IR moves that can be removed
    for each edge \( e = (b_1, b) \in E_m \cup E_r \) do
        if \( b_1 \) and \( b \) do not intersect then
            \( p := \text{reg}(b_1) \);
            if \( p \neq \text{null} \) and \( p \in \text{avail} \) then
                Map\((p) := Map\((p) + W(e)\);
            end
        end
    end

    ret := Find \( p \) with maximum cost in Map;
    if \( \text{ret} == \text{null} \) then
        \( \text{ret} := \text{Find any free physical register from } R \);
    end

    Remove \( \text{ret} \) from \( \text{avail} \);
    \( \text{reg}(b) := \text{ret} \);
    return \( \text{reg}(b) \);

between basic intervals of the same compound interval and hence needs \( |C| \cdot \operatorname{max}_c \) number of edges. \( \Box \)

THEOREM 5.8.
Register assignment using Algorithm takes \( O((|B| \cdot \max_c) + |IR|) + (|E| \cdot (|C| + |K|^2)) \) time.

Proof: In addition to Theorem 4.8, before deciding a physical register for each basic interval \( b \) it is required to traverse each of the neighbors in \( CG \). For all basic intervals, this adds over all \( 2 \cdot |IR| \) time complexity for \( IR \) move instructions and \( |B| \cdot \max_c \) time complexity for \( E_r \) edges in \( CG \). \( \Box \)

6. SPILLING AND ASSIGNMENT WITH REGISTER CLASSES

In the preceding sections, we have described register spilling and assignment for \( k \) physical registers that are uniform, i.e., they are independent and interchangeable [Smith et al. 2004]. However, modern systems such as x86, HP RA-RISC, Sun SPARC, and MIPS come with physical registers which may not necessarily be interchangeable. For example, the Intel 32-bit x86 architecture provides eight integer physical registers, of which six are typically exposed for register allocation. These six physical registers are further divided into four high level overlapping register classes based on calling conventions and 8-bit operand accesses. Since the register classes may not necessarily be disjoint, a register allocator must take into account register classes during spilling and assignment to produce high quality machine code. In this section, we describe how spilling and assignment can be performed in the presence of register classes. We assume calling conventions related constraints are also expressed in additional register classes with infinite spill cost.

6.1. Constrained Spilling using BLG

Allocation in the presence of register classes can be achieved using the following two approaches:
(1) Build BLG for each register class and apply the algorithm in Figure 4 to each BLG in a particular order starting with the most constrained register class that has fewer physical registers in a class. For example, in the 32-bit x86 architecture, we need to build four BLGs for four integer register classes and apply the algorithm in Figure 4 in the order 8 bit non-volatile (EBX), non-volatile (EBX, EBP, and EDI), 8 bit volatile (EAX, EBX, ECX, and EDX), and then for the complete integer register class. If a compound interval is spilled in a BLG for a register class, that decision needs to be propagated to the other BLGs of other classes.

(2) An alternative approach is to build a single BLG. During every visit of an interval end point in Figure 4, we make it unconstrained with respect to all other register classes before another end point is visited. This approach is space-efficient as it builds only one BLG but can eagerly generate more spills than (1).

Our experimental results in Section were obtained using Approach (1).

**ALGORITHM 5:** Bucket-based greedy heuristic to perform assignment in the presence of register classes.

```
function ConstrainedAssignment ()
    Input : Set of basic intervals \( b \in B \), \( \forall b \in B \text{ regclass}(b) \), a set of physical register classes \( K \), a compile-time constant \( \text{num\_bucket} \)
    Output: Find the assignment \( \text{reg}(b) \) and spill decision \( \text{spilled}(b) \)
    //Find total number of elements per regclass
    for \( b \in B \) do
        \( \text{cid} := \text{getClassId}(\text{regclass}(b)) \);
        \( \text{perClass}[\text{cid}] ++ \);
    end
    //Decide per bucket number of elements
    for \( i := 0; i < |K|; i ++ \) do
        \( \text{perBucket}[i] := \left\lfloor \frac{|\text{perClass}[i]|}{|K|} \right\rfloor + 1 \);
        \( \text{availBucket}[i] := 0 \);
    end
    //assignOrder is a 2-d array of basic intervals;
    //Determine the bucket for \( b \);
    for \( b \in B \) in decreasing order of \( \text{SPILL}(b) \) do
        \( \text{cid} := \text{getClassId}(\text{regclass}(b)); \)
        \( \text{bucket} := \text{availBucket}[\text{cid}] ; \)
        Append \( b \) to \( \text{assignOrder}[\text{bucket}][\text{cid}] \);
        if \( |\text{assignOrder}[\text{bucket}][\text{cid}]| > \text{perBucket}[\text{cid}] \) then
            \( \text{availBucket}[\text{cid}] ++ \);
        end
    end
    //Assign physical registers
    for \( i := 0; i < |K|; i ++ \) do
        for \( j := 0; j < \text{num\_bucket}; j ++ \) do
            for \( b \in \text{assignOrder}[i][j] \) do
                findAssignment(b);
            end
        end
    end
end
```
6.2. Constrained Assignment and Move Coalescing

Given a coalesce graph (as defined in Section ), when we try to find an assignment for a basic interval \( b \), the register classes of the neighbors of \( b \) in the coalesce graph along with the register class of \( b \), play a key role in selecting a physical register for \( b \). An IR move instruction can be coalesced if source and destination basic intervals have a non-null intersection in their register classes.

Another key point in register assignment is that we no longer can rely on the increasing start point order for assignment of basic intervals since an early decision of physical register assignment of a register class may result in more symbolic registers being spilled later on or giving up other opportunities for coalescing. We define the register assignment problem in the presence of register classes as an optimization problem that may incur additional spills.

**Definition 6.1.**

**Constrained Assignment Optimization Problem:** Given a set of basic intervals \( b \in B \) with \( \text{spilled}(b) = \text{false} \), \( \text{regclass}(b) \) indicating physical registers that can be assigned to each \( b \), \( CG = (V, E = \{E_m \cup E_r\}) \), and IR, find a register assignment \( \text{reg}(b) \) for a subset of basic intervals \( S \subseteq B \) such that the following objective function is minimized:

\[
\sum_{b \in B - S} \text{SPILL}(b) + \sum_{e \in E, e = (b_1, b_2) \land \text{reg}(b_1) \neq \text{reg}(b_2)} W(e)
\]

Insert additional register-to-register copy or exchange instructions in the IR. □

**ALGORITHM 6:** Greedy heuristic to choose a physical register that maximizes copy removal in the presence of register classes

```plaintext
function findAssignment()
    Input : A basic interval \( b \in B \), \( \forall b \in B \ \text{regclass}(b) \), coalesce graph \( G = (V, E = \{E_m \cup E_r\}) \), a set of available physical registers \( \text{avail} \)
    Output: Find the assignment \( \text{reg}(b) \)
    Compute Map using Steps 3-7 of Algorithm ;
    \( RMap := \text{Map} ; \)
    for each edge \( e = (b_1, b_2) \in E_m \cup E_r \) do
        if \( b_1 \) and \( b_2 \) intersect then
            for each \( p \) in Map do
                if \( p \) can be assigned to \( b_1 \), i.e., \( p \in \text{regclass}(b_1) \) then
                    \( RMap(p) := RMap(p) + W(e) ; \)
                end
            end
        end
    end
    \( ret := \text{Find} \ p \ \text{with maximum cost in} \ RMap ; \)
    Follow Steps 7-11 of Algorithm ;
```

Algorithm presents a bucket-based approach to register assignment that tries to strike a balance between register classes and spill cost. The assignOrder data structure holds sorted basic intervals according to register classes in a two dimensional array. Each register class is represented as a unique integer id. Steps 2-4 compute the total number of basic intervals per register class. Steps 5-7 compute the number of elements per bucket. Steps 8-13 decide the appropriate bucket in assignOrder where
a basic interval should reside (based on next availability). Steps 14-17 find an assignment for basic intervals by traversing the assignOrder array in a row major order. The heuristic for assigning a physical register to a basic interval follows a similar approach described in Section except additional care must be taken to account for register class constraints. The details are provided in Algorithm.

7. EXPERIMENTAL RESULTS

We present an experimental evaluation of the BLG register spilling and assignment algorithms presented in this paper. The experimental setup consists of two compiler infrastructures, a static compiler evaluation using LLVM 2.7 [llv 2009] and a dynamic compiler evaluation JikesRVM 3.1.1 [jik 2011]. In the static compilation evaluation, we perform both compile-time and run-time comparisons of our BLG allocator compared to an existing Graph Coloring implementation [Cooper and Dasgupta 2006] and the LLVM Linear Scan [llv 2009]. In the dynamic compilation evaluation, we compare our BLG allocator performance compared to JikesRVM Linear Scan algorithm.

7.1. LLVM 2.7 evaluation

The LLVM evaluations were performed on an Intel Xeon 2.66GHz system with 8GB of memory and running RedHat Linux (RHEL 5).

Benchmarks: We used ten benchmarks from the SPECCPU 2006 benchmark suite [Corporation 2006]. The integer benchmarks used are 401.bzip2, 429.mcf, 458.sjeng, 464.h264ref, and 473.astar. The floating-point benchmarks used are 410.bwaves, 434.zeusmp, 435.gromacs, 444.namd, and 470.lbm. All the benchmarks were executed under the optimization level -O2 of LLVM. Since we invoked LLVM in static compilation mode, we ran each benchmark five times and reported the best of the 5 runs as the runtime performance measurement.

Comparison approaches: Experimental results are reported for the following cases:

(1) LLVMLS – Baseline measurement using the default Linear Scan register allocator in LLVM; This allocator implements aggressive live-range splitting and differs from the standard linear scan algorithm [Poletto and Sarkar 1999] by introducing backtracking. These extensions are described in Wimmer et al. [Wimmer and Mössenböck 2005]. This algorithm also performs aggressive coalescing prior to register allocation.

(2) GC – the Chaitin-Briggs [Chaitin et al. 1981; Briggs et al. 1994] register allocator. This implementation uses the same code base of Chaitin-Briggs allocator with aggressive coalescing that was used in [Cooper and Dasgupta 2006]. Details of the Chaitin-Briggs allocator can be found in [Briggs et al. 1994].

(3) BLG+LS – the register spilling and assignment algorithm presented in Section with the spill code generation algorithm from 1) above i.e., after the allocation and assignment passes are completed using BLG, the IR is rewritten using the physical registers for the non-spilled variables and move code is inserted. The IR is then passed to the Linear Scan register allocator of LLVM to generate spill code\(^3\).

(4) BLG+GS – the register spilling and assignment algorithm presented in Section with the spill code generation algorithm from 2) above i.e., after allocation and assignment are completed using BLG, the IR is rewritten using the physical registers for the non-spilled variables and move code is inserted. The IR is then passed to the

---

\(^3\)We do not devise any new spill code generation technique – our focus is on spilling and assignment, and thus, we use existing spill code generation techniques such as basic block based spill code generation in LLVM Linear Scan and global spill code generation in Graph coloring.
Chaitin-Briggs register allocator to generate spill code). For the BLG allocator, we set the compile-time constant $num\_bucket$ to 5. Note that, this approach does not yet implement partial spills.

Table I. Comparison of compile-time statistics between BLG+LS and GC for SPECCPU 2006 benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>max function</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$IR$</td>
<td>IG</td>
<td>IG</td>
<td>BLG</td>
<td>BLG</td>
<td>BLG</td>
</tr>
<tr>
<td></td>
<td>#nodes</td>
<td>#edges</td>
<td>#nodes</td>
<td>#edges</td>
<td>Space Usage Ratio</td>
<td>nodes</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>sendMTFValues</td>
<td>3545</td>
<td>2693</td>
<td>53562</td>
<td>1844</td>
<td>9819</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>bi_cgstab_block_</td>
<td>2083</td>
<td>1025</td>
<td>5430</td>
<td>134</td>
<td>269</td>
</tr>
<tr>
<td>429.mcf</td>
<td>read_min</td>
<td>440</td>
<td>279</td>
<td>3376</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>setup</td>
<td>5147</td>
<td>3030</td>
<td>33138</td>
<td>387</td>
<td>1750</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>do_inputrec</td>
<td>3519</td>
<td>1941</td>
<td>36606</td>
<td>64</td>
<td>142</td>
</tr>
<tr>
<td>444.namd</td>
<td>_ZN20ComputeNonbondedUtil-50calc_self_energy_fullelect_fep-EP9nonbonded</td>
<td>2244</td>
<td>907</td>
<td>6156</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>std_eval</td>
<td>1316</td>
<td>812</td>
<td>7908</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>SubPelBlock-SearchBiPred</td>
<td>5787</td>
<td>4757</td>
<td>86092</td>
<td>356</td>
<td>921</td>
</tr>
<tr>
<td>470.lbm</td>
<td>LBM_handleIn-OutFlow</td>
<td>1162</td>
<td>643</td>
<td>5380</td>
<td>189</td>
<td>270</td>
</tr>
<tr>
<td>473.astar</td>
<td>_ZN6wayobj18-makeobstacle-bound2EPiiS0</td>
<td>382</td>
<td>295</td>
<td>438</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Compile-time Comparison: Table compares the compile-time overheads of BLG vs. GC. The measurements were obtained for functions with the largest interference graphs (in term of number of nodes) in the SPECCPU 2006 benchmarks. Column 3 reports the total number of LLVM IR instructions for the max function. Column 4 and 5 report the total number of nodes and edges in the IG respectively. (We only report these numbers for the first iteration of the Chaitin-Briggs allocator – subsequent iterations require additional smaller interference graphs.) Column 5 and 6 report the total number of nodes and edges in BLG that only considers constrained interval end points (i.e., those end points with MAXLIVE > $k$; unconstrained interval end points are not necessary, as described in Section 4). We define Space Usage Ratio metric as the ratio of the following two quantities: (1) sum of columns 3-5 ($|IG|$); (2) sum of columns 3, 6, and 7 ($|BLG|$). This metric varies from $2.9 \times$ to $13.7 \times$ in our case, indicating the lower space usage of BLG compared to GC. While theoretically both IG and BLG can be quadratic, in practice, we observe BLG to be much smaller than IG.

Runtime comparison: Figure reports the relative performance improvement of the register allocation algorithm presented in this paper along with Chaitin-Briggs spill code generator, BLG+GS, compared to the original Chaitin-Briggs allocator, i.e., GC on the Intel Xeon system. We observe a performance improvement of up to 7.87% in 464.h264ref benchmark and we do not observe any degradation in any of the benchmarks. While comparing our BLG allocator with Linear Scan spill code generator, i.e., BLG+LS, with that of LLVM’s default register allocator LLVM+LS (as shown in Table), we did not observe any noticeable performance difference. The reason is that the default LLVM register allocator implements other register allocation techniques such
Fig. 3. Percentage Improvement of execution times obtained by BLG+GS, (i.e., BLG+Chaitin-Briggs spiller) compared to GC in the LLVM static compiler infrastructure for SPECCPU 2006 benchmarks on the Intel Xeon system.

Table II. Comparison of runtime performance of BLG+LS vs. LLVM Linear Scan register allocation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BLG+LS execution time (in sec)</th>
<th>LLVM+LS execution time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.lzip</td>
<td>9.9</td>
<td>10.0</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>2856.4</td>
<td>2853.1</td>
</tr>
<tr>
<td>429.mcf</td>
<td>6.7</td>
<td>6.8</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>40.4</td>
<td>40.5</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>2079.1</td>
<td>2076.7</td>
</tr>
<tr>
<td>444.namd</td>
<td>38.1</td>
<td>38.1</td>
</tr>
<tr>
<td>455.sjeng</td>
<td>11.0</td>
<td>11.1</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>1806.4</td>
<td>1806.4</td>
</tr>
<tr>
<td>470.lbm</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>473.astar</td>
<td>23.5</td>
<td>23.5</td>
</tr>
</tbody>
</table>

Execution Time: Comparison of execution times obtained by BLG+LS, (i.e., BLG+Linear Scan Spiller) compared to the default LLVM Linear Scan for SPEC CPU 2006 benchmarks using LLVM static compiler on the Intel system. Note that LLVM+LS performs additional optimizations, such as live-range splitting and backtracking compared to BLG+LS.

as aggressive live-range splitting and backtracking in order to help moderate register pressure during spilling and assignment phases. The adhoc heuristic via backtracking in LLVM performs unspilling recursively in order to avoid reserved spill registers and, this pass has a quadratic complexity as described in [Evlogimenos 2004]. Additionally, our scheme without any sophisticated live-range splitting mechanism is able to match the performance of state-of-the-art LLVM. In future, we would like to devise live-range splitting heuristics for BLG that exploit the structure of the program [Lueh et al. 2000; Appel and George 2001].
7.2. JikesRVM 3.1.1 evaluation

The JikesRVM evaluations were performed on two systems: (1) Intel Xeon 2.66GHz system with 8GB of memory and running RedHat Linux (RHEL 5); (2) PowerPC 7 2.66GHz system with 8GB memory, running SUSE Linux.

**Benchmarks:** We used the serial benchmarks in v2.0 of the Java Grande Forum (JGF) benchmark suite [jgf 2001] and Dacapo 2006 benchmark suite [Blackburn et al. 2006] to evaluate the performance of our register allocator. We choose the five large benchmarks from Section 3 ([raytracer, moldyn, montecarlo, euler, and search])

4 For Dacapo benchmark suite, we report performance evaluation of ten benchmarks out of total eleven benchmarks. These include antlr, bloat, fop, hsqldb, jython, luindex, pmd, xalan, lusearch, and eclipse6. Further, for PowerPC 7 evaluation, we could not compile lusearch and luindex benchmarks in Jikes RVM 3.1.1.

**Compiler:** The boot image for JikesRVM used a production configuration. Since the JikesRVM release did not support generation of Intel exchange instruction, we modified its assembler to add this support. JikesRVM uses SSE registers for storing double/floating point values. However, to the best of our knowledge, there does not exist a direct exchange instruction to swap values in SSE registers, so we generate three xor instructions to exchange a pair of float/double values. The exchange instructions are generated judiciously, i.e., if there is a free physical register available for swapping the values, an exchange instruction is not generated [Boissinot et al. 2009]. For all Java runs, the execution times are reported for dynamic compilation (both runtime and compile-time) and use the methodology described in [Georges et al. 2007], i.e., we report the average runtime performance of 30-runs within a single VM invocation along with the execution variance that uses a 95% confidence interval.

**Comparison approaches:** Experimental results in JikesRVM evaluation are reported for the following cases: 1) LS – Baseline measurement with Linear Scan register allocator in JikesRVM that uses the algorithm from [Poletto and Sarkar 1999] with extensions for live-range “holes”; 2) ELS – the Extended Linear Scan algorithm from [Sarkar and Barik 2007]; 3) BLG – the BLG register allocation algorithm presented in Section ; 4) BLG+PARTIAL – the BLG register allocation algorithm with partial spills presented in Section . The compile-time constant num_bucket in Figure is set to 5 for all runs. Increasing this number to a higher value does not impact the runtime performance obviously.

**Runtime comparison:** Figure reports the relative performance improvements for ELS, BLG, and BLG + PARTIAL allocators compared to the default LS allocator of JikesRVM on the Intel Xeon system. The BLG register allocator resulted in a performance improvement in the range of -0.04% to 11.37% (for moldyn). The BLG+PARTIAL register allocator resulted in a performance improvement in the range of -0.69% to 8.81% (for moldyn). For moldyn benchmark, the most-frequently executed function is force. MAXLIVE for this function is >7. (Jikes RVM uses 8 SSE registers for storing double/float values, and one out of them, XMM7, is used for scratch register.) Spilling decisions for this method impact the performance of the benchmark significantly. BLG for this method coalesces more moves than LS and is able to spill 14 symbolic registers compared to 16 symbolic registers in LS. The BLG + PARTIAL allocator improves performance for bloat, eclipse, montecarlo, and euler benchmarks when compared to BLG. The runtime performance benefits for both BLG and BLG + PARTIAL are not surprising as they perform global spill decisions on a bipartite liveness graph compared to the local spill decisions made by LS and ELS. We observed a slow-down of

---

4Results for Section 1 and 2 benchmarks have been omitted since they are smaller benchmarks.

5chart is omitted as it requires special AWT library to compile and existing Jike RVM is unable to compile it.
A Decoupled non-SSA Global Register Allocation using Bipartite Liveness Graphs

Fig. 4. Percentage improvement of ELS, BLG, and BLG + PARTIAL compared to LS in JikesRVM dynamic compiler on a x86_64 Intel Xeon system.

0.69%, 0.41%, and 0.12% for luindex, psd, and raytracer for BLG + PARTIAL: our current heuristic splits live-ranges only at basic interval granularity which may not be optimal. More sophisticated live-range splitting is left for future work.

On the PowerPC 7 system, we observe performance improvements for BLG allocator compared to LS in the range of 0.23% to 7.34% (for xalan) as shown in Figure. The BLG + PARTIAL allocator is able to improve performance for most of the benchmarks.

**Compile-time comparison:** Table reports compile-time comparison of BLG vs. LS. As described in previous sections, Linear Scan is best for compile-time efficiency as it performs both spilling and assignment in just one pass over the basic intervals. BLG adds extra new passes for spilling and unspilling via bipartite graph, move-code generation, and move coalescing optimization. Thus, BLG is expected to perform slower than LS. We observe an increase in compile-time from 2.02x to 5.37x for BLG vs. LS. This increase in compile-time is insignificant compared to the total execution time of a benchmark since BLG outperforms LS for all benchmarks except euler on the Xeon system and for all benchmarks on the PowerPC 7 system. Interestingly, in our current implementation we observe that the move-code generation component consumes maximum time. This is because it may require the construction of a move-graph and performs strongly connected component search in this graph for a control flow edge. In future, we would like to optimize the compile-time of this phase.

**Static Spill-cost savings:** Figure reports the percentage improvement in static spill cost for BLG compared to LS. The static frequency estimates are computed using standard technique where a spill instruction inside a loop is estimated as $10^d$, where $d$ denotes loop depth. We observe reduction in static spill cost for all workloads. For
Percentage improvement of \textit{ELS}, \textit{BLG}, and \textit{BLG+PARTIAL} compared to \textit{LS} in JikesRVM dynamic compiler on a PowerPC 7 system.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BLG Comp. time in ms</th>
<th>LS Comp. time in ms</th>
<th>Relative increase in Comp. time (BLG/LS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>0.35</td>
<td>0.07</td>
<td>5.37</td>
</tr>
<tr>
<td>bloat</td>
<td>1.43</td>
<td>0.3</td>
<td>4.76</td>
</tr>
<tr>
<td>fop</td>
<td>0.08</td>
<td>0.03</td>
<td>2.44</td>
</tr>
<tr>
<td>hsqldb</td>
<td>0.87</td>
<td>0.28</td>
<td>3.08</td>
</tr>
<tr>
<td>jython</td>
<td>0.6</td>
<td>0.11</td>
<td>5.27</td>
</tr>
<tr>
<td>luindex</td>
<td>1.08</td>
<td>0.22</td>
<td>4.8</td>
</tr>
<tr>
<td>pmd</td>
<td>0.78</td>
<td>0.21</td>
<td>3.62</td>
</tr>
<tr>
<td>xalan</td>
<td>0.9</td>
<td>0.31</td>
<td>2.89</td>
</tr>
<tr>
<td>eclipse</td>
<td>1.58</td>
<td>0.78</td>
<td>2.02</td>
</tr>
<tr>
<td>montecarlo</td>
<td>0.09</td>
<td>0.02</td>
<td>3.8</td>
</tr>
<tr>
<td>raytracer</td>
<td>0.05</td>
<td>0.02</td>
<td>3.23</td>
</tr>
<tr>
<td>moldyn</td>
<td>0.02</td>
<td>0.01</td>
<td>2.83</td>
</tr>
<tr>
<td>euler</td>
<td>0.26</td>
<td>0.1</td>
<td>2.5</td>
</tr>
<tr>
<td>search</td>
<td>0.05</td>
<td>0.015</td>
<td>3.2</td>
</tr>
</tbody>
</table>

eclipse, we reduce the spill cost by 93% which is significant. Please keep in mind that these static measures may not directly correlate to runtime performances due to pipelining and caching effects.
8. RELATED WORK

Spill-free register allocation of general programs is NP-complete [Chaitin et al. 1981]. There exist a plethora of past works in using graph coloring-based approaches to spill-free register allocation [Chaitin et al. 1981; Briggs et al. 1989; Briggs et al. 1994; Park and Moon 1998; George and Appel 1996; Budimlic et al. 2002; Callahan and Koblenz 1991; Gupta et al. 1994; Smith et al. 2004; Cooper and Dasgupta 2006]. The key data structures of a Graph Coloring based algorithm are live-ranges and the interference graph. Allocation phase is performed on the interference graph by removing the live-ranges of degree fewer than \( k \). In cases where every live-range has degree more than or equal to \( k \), a live-range having lowest spill cost is chosen for spilling. The live-ranges that are removed from the interference graph, are assigned physical registers based on the reverse order in which the live-ranges were removed from the interference graph. One of the key limitations of graph coloring based register allocation is that the live-ranges introduce imprecision that may lead to making the interference graph uncolorable (like the one seen in Figure 3). In contrast, our approach builds on the simple foundations of Linear Scan register allocation like intervals and precisely captures liveness information using a novel BLG data structure, which is used for spill-free register allocation [Sarkar and Barik 2007].

Recently, the focus in graph coloring-based register allocation has shifted to SSA-based register allocation [Hack and Goos 2006; Brisk et al. 2005; Brisk 2006; Colombet et al. 2011; Bouchez 2009; Pereira and Palsberg 2005; 2009; Braun et al. 2010]. In SSA representation, the interference graph is chordal and can be colored optimally in linear time. Like our approach and others in the literature [Appel and George 2001], current approaches to SSA register allocation separate between allocation and assignment phases in register allocation. However, an SSA register allocation incurs additional complexity of dealing with parallel-copy statements during out-of-ssa translation [Hack and Goos 2008; Brisk 2006] and also of dealing with repairing [Colombet et al. 2011]. Our BLG allocator does not need an interference graph for allocation and efficiently inserts a few register-to-register moves and exchange operations during assignment as opposed to expensive approaches to eliminate a large number of parallel-copy instructions in SSA-based register allocation.
Linear Scan [Poletto and Sarkar 1999; Traub et al. 1998; Wimmer and Mössenböck 2005; Thammanur and Pande 2004; Wimmer and Franz 2010; Sarkar and Barik 2007] register allocation algorithms have been preferred for JIT-compilers such as Jikes [jik 2011], HotSpot [Kotzmann et al. 2008], and LLVM [llv 2009] due to their low compilation-time and space complexity. Compared to existing linear scan algorithms, our approach separates allocation and assignment phases. This leads to a much better global spilling decision using a novel bipartite graph. Traditional linear scan algorithms often combine allocation and assignment for efficiency reasons and hence end up making local spill decisions that lead to performance lag. The spill-free register allocation algorithm presented in Extended Linear Scan (ELS) algorithm [Sarkar and Barik 2007] the spill decisions are taken locally at every program point (i.e., each interval end point is eagerly made completely unconstrained before moving onto another). This is the reason why they had observed a slowdown in SPEC benchmark 181.mcf.

In contrast, the BLG based allocation algorithm described in this paper makes global decisions using the BLG data structure that decides the symbolic registers that need to be spilled to keep the overall spill cost minimized. Additionally, this paper describes move coalescing optimizations (in Section ), register allocation in the presence of register classes (in Section ), and partial spills (in Section ). More recently, a tree-based register allocation algorithm has been proposed in [Rong 2009] that imposes a partial ordering among the basic blocks during coloring and assignment phases unlike the total order imposed in linear scan.

The graph coloring-based register allocation algorithm was first extended to handle register classes and aliasing by Smith et al [Smith et al. 2004]. The problem of spill-free register allocation is NP-complete even in the presence of register classes and aliasing [Lee et al. 2007]. The approach taken by Smith et al is to handle register classes and aliasing by exploiting the coloring constraints on each node of the interference graph. This approach is elegant and can be easily integrated into any graph coloring register allocation algorithm. More recently, a new Linear Scan register allocation algorithm based on puzzle solving was introduced by Pereira and Palsberg [Pereira and Palsberg 2008; 2010] to handle precoloring and aliasing issues in register allocation. Their approach views the register file as a puzzle and the program variables as puzzle pieces. For many common architectures, the register allocation using puzzles can be solved in polynomial time. Our BLG register allocator handles these architectural constraints without building the interference graph. For allocation phase, we construct BLG for each register class and propagate spill information across BLG’s of other register classes. For assignment phase, we use a bucket-based approach that strikes a balance between spill cost and move code optimization.

A bipartite graph-based register assignment phase was proposed by Zhang et al. [Zhang et al. 2004] that is performed on hot paths of an already register allocated code, i.e., as a post register allocation pass. The spilled variables on the hot path form one set of vertices of the bipartite graph where as the other set of vertices consists of the set of dead physical registers. An edge is added to their bipartite graph if both the spilled variable and dead physical register are alive in the same basic block. The weight of such an edge is the spill cost of the spilled variable in the basic block. Dead register assignment is then performed using weighted bipartite graph matching. This approach differs from our BLG allocator in many ways: 1) the nodes, edges, and weights of the bipartite graph are all different; 2) our bipartite liveness graph represents liveness information and solves the allocation phase of register allocation.

The meeting graph model for loop cyclic register allocation described in [Eisenbeis et al. 1995] is different from the BLG model. The meeting graph captures information about non-overlapping intervals i.e., an edge is added when one interval ends and another starts. This information is useful for obtaining bounds for optimal coloring.
inside loops. In contrast, BLG captures liveness information at high pressure program points which is used to perform global register allocation.

9. CONCLUSIONS

In this paper, we addressed the problem of developing a register allocation algorithm that builds on the simplicity of Linear Scan while improving its runtime performance. It does so by separating the spilling and assignment phases. The spilling phase is modeled as an optimization problem on Bipartite Liveness Graphs (BLG’s), a new data structure introduced in this paper. In the spilling and assignment phase, we focus on reducing the number of spill instructions by using register-to-register move and exchange instructions wherever possible to maximize the use of registers. We model register assignment as a second optimization problem that includes move coalescing, as well as register class constraints, and provide a heuristic solution to this problem as well. Our implementation of BLG-based register allocation phase combined with the constrained assignment in JikesRVM demonstrates runtime performances improvements in the range of -0.04% to 11.37% and in the range of 0.23% to 7.34% on Intel Xeon and PowerPC 7 systems respectively. Additionally, we observe a performance improvement of up to 7.87% for SPECTCPU 2006 benchmarks using our BLG register allocator that uses a graph coloring based spill code generator when compared to Chaitin-Briggs register allocator on the Intel Xeon system.

These results show that BLG register allocation algorithm is a promising alternate to the large body of register allocators existing today. Possible directions for future work include support for more aggressive live-range splitting, backtracking, and studying the impact of move and exchange instructions on code size compared to spill load/store instructions. Further, we would like to study the combined effect of BLG with instruction scheduling.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and suggestions on the past submissions related to this article.

REFERENCES


Keith D. Cooper and Anshuman Dasgupta. 2006. Tailoring Graph-coloring Register Allocation For Runtime Compilation. In CGO ’06. IEEE Computer Society, Washington, DC, USA, 39–49. DOI: http://dx.doi.org/10.1109/CGO.2006.35


Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. ACM Transactions on Programming Languages and Systems 18, 3 (May 1996), 300–324.


A Decoupled non-SSA Global Register Allocation using Bipartite Liveness Graphs


Received June 2013; revised September 2013; accepted November 2013