Split Register Allocation: Linear Complexity Without the Performance Penalty

Boubacar Diouf,[1] Albert Cohen,[1] Fabrice Rastello,[2] John Cavazos[3]

[1] ALCHEMY Group, INRIA Saclay and Paris-Sud University,
[2] LIP, École Normale Supérieure de Lyon,[3] University of Delaware

Abstract. Just-in-time compilers are becoming ubiquitous, spurring the design of more efficient algorithms and more elaborate intermediate representations. They rely on continuous, feedback-directed (re-)compilation frameworks to adaptively select a limited set of hot functions for aggressive optimization. To date, (quasi-)linear complexity has remained a driving force in the design of just-in-time optimizers.

This paper describes a *split register allocator* showing that linear complexity does not imply reduced code quality. We present a *split compiler* design, where more expensive ahead-of-time analyses guide lightweight just-in-time optimizations. A split register allocator can be very aggressive in its offline stage, producing a semantic summary through bytecode annotations that can be processed by a lightweight online stage. The challenges are fourfold: (sub-)linear-size annotation, linear-time online processing, minimal loss of code quality, and portability of the annotation.

We propose a split register allocator meeting these challenges. A compact annotation derived from an optimal integer linear program (ILP) formulation of register allocation drives a linear-time algorithm near optimality. We study the robustness of this algorithm to variations in the number of physical registers. Our method is implemented in JikesRVM and evaluated on standard benchmarks.

1 Introduction

Just-In-Time (JIT) compilers rely on continuous, feedback-directed (re-)compilation frameworks to select hot functions (frequently executed) for online optimizations. These online optimizations must make important trade-offs in terms of reducing compilation time for decreased generated code performance. Reducing compilation overhead has two main benefits, low-complexity algorithms simultaneously increase the amount of code being optimized while reducing the compilation time for hot functions. In practice, (quasi-)linear complexity is the rule for JIT compilation. This severely impacts what kind of optimizations are admissible and how aggressive they may be.

1.1 A Case for Split Compilation

Traditional bytecode language tool chains distribute the roles among offline and online compilers. Verification and code compaction are typically assigned to of-

fline compilation, while target-specific optimizations are performed by online compilation. *Split compilation* reconsiders this notion: it allows *a single optimization algorithm* to be split into *an offline and an online stage*, transferring the semantic information between those stages through carefully designed bytecode annotations.

Split compilation has the potential to combine the advantages of offline and online compilation: running expensive analyses offline to prune the optimization space, defering a more educated optimization decision to the online stage, when the precise execution context is known. Many JIT compilation efforts tried to leverage the accuracy of dynamic analysis to outperform native compilers; but split compilation is a concrete path to get the best of both worlds.

To make a concrete case for split compilation, we selected the (spill-everywhere) register allocation problem [1, 2]. Register allocation is an ideal candidate to demonstrate how split compilation impacts the design of future bytecode languages and compilers, and how it differs from plain annotation-enhanced JIT compilation [3]. Indeed:

– the principles of register allocation are reasonably well understood;
– it is one of the most important components of all JIT compilers;
– it is challenging to design an offline analysis that would improve online register allocation, while ignoring the exact register count of the target.

1.2 Outline of the Paper

This paper makes two important contributions.

1. We design bytecode annotations enabling a linear-time online algorithm to achieve high-quality register allocation, with negligible impact on the size of the bytecode.
2. We demonstrate how such annotations are robust to variations in the number of registers. With additional provisions in the offline stage, it is even possible to accommodate radical changes in the instruction set target architecture.

Our method is implemented in the JikesRVM open source JIT compiler for Java [4], and evaluated on x86. We do believe that it would be easy to port it to multi-language JIT frameworks like the ECMA-335 CLI standard.[1]

The paper is organized as follows. Section 2 presents the split register allocation flow and algorithms. Section 3 evaluates split register allocation, with coverage of performance improvements as well as annotation compaction and portability. Section 4 explores more complex compilation scenarios. Finally, Section 5 discusses related work on annotation-enhanced just-in-time compilation

2 Split Register Allocation

We first introduce some terminology. An interval characterizing the entire lifetime of a local variable or temporary may contain some *idle holes*. The live range

---

[1] http://www.ecma-international.org/publications/standards/Ecma-335.htm

of a variable x is the set of program points where x is live; it corresponds to a union of basic intervals. When linearising the control flow (e.g., when generating code), the basic interval of a given live range are interleaved with holes. Those holes correspond either to program points dominated by a redefinition of the variable (the variable is effectively dead at those points), or to a hole resulting from the order in which the basic blocks are numbered (a control-flow artifact). *Register pressure* refers to the amount of locally living variables. Considering that a variable is not alive during its idle holes can help in reducing the register pressure. JikesRVM takes advantage of this.

2.1 Optimization Problem and Baseline Algorithm

Since our primary focus is to illustrate the split compilation concept, we limit ourselves to the most basic register allocation and assignment problem:

– Spill everywhere allocation: spill the whole live range.
– Single-color assignment: when such a live range is allocated, all its basic intervals must be assigned to the same register. Some live ranges may be preassigned due to function call conventions and operand restrictions of some target instructions;

Throughout the paper, we handle register allocation in different register classes separately (e.g., general purpose, floating point), and call R the number of registers in the current class of interest.

Algorithm 1 recalls the main steps of the linear scan algorithm, as implemented in JikesRVM. Every time a basic interval i becomes active, Algorithm 1 calls the function ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$), where $V(i)$ is the live range corresponding to i. According to the allocation that has been performed up to this point, function ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$) returns, either a live range or $\perp$ (bottom): if it returns a live range, it is the one to be spilled in order to continue allocation; if it returns $\perp$ it was possible to assign $V(i)$ without spilling. These algorithms are the basic framework upon which the offline and online phases of our split register allocation are constructed.

Algorithm 1 LINEARSCAN

Input: *list*: the list of basic intervals ordered by increasing start point
1: **foreach:** $i \in list$ **do**
2:    $toSpill \leftarrow$ ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$)
3:    **if** $toSpill \neq \perp$ **then**
4:       **if** $toSpill \neq V(i)$ **then**
5:          Assign $V(i)$ to the register freed by $toSpill$
6:       **end if**
7:       Spill $toSpill$
8:    **end if**
9: **end for**
Return: sets of spilled live ranges and register assignments

Algorithm 2 ASSIGNORSUGGESTSPILLCANDIDATE

Input: v: a live range
1: **if** v was previously assigned to a register r **then**
2: **if** r is free **then**
3:    Continue with this assignment
4:    Return $\perp$
5: **else if** v can be assigned to another register r' **then**
6:    Assign v to r'
7:    Return $\perp$
8: **else**
9:    Let v' be the live range assigned to r
10:    Return the live range with the minimum cost among v and v'
11: **end if**
12: **else if** v can be assigned to a free register r **then**
13: Assign v to r
14: Return $\perp$
15: **else**
16: Return v' with the lowest cost among v and the other live ranges at the current point
17: **end if**
Return: a live range to spill or $\perp$

2.2 The ILP Model

Here, we discuss our formulation of spilling in register allocation as an ILP problem. We obtain spilling decisions offline and pass this information to the online compilation phase using annotations. Considering a set S of live ranges, a *spill set* of S is any subset S' of S such that $S \setminus S'$ can be allocated over the R registers (without spilling). We also consider a function which assigns to each live range in S the cost of spilling it. The cost of a spill set is the sum of the costs of live ranges within that set. An optimal register allocation is associated with a spill set with the minimal cost.

We build an ILP model that is optimal among spill-everywhere, single-color allocations, for a given cost model.

We model register allocation as a $\{0, 1\}$ linear program, the objective function being the cost of the spill set. We support multiple classes of registers, each register class is further decomposed into 2 subclasses: caller-saved (scratch register) and callee-saved (non-scratch register). Live ranges are partitioned according to register classes, and can be of the *volatile*, *non-volatile* or *preassigned* kinds: a *non*-volatile live range can only be assigned to some callee-saved register, and a preassigned live range can only be assigned to a specific physical register.

We create a $\{0, 1\}$ variable l_r for each live range l and register r that l may be assigned to (considering class and volatility constraints):

$$l_r = 1 \text{ if and only if } l \text{ is assigned to } r.$$

These variables are constrained by 3 kinds of (in)equalities.

1. At most one register per live range (single color assignment):

$$\sum_{1 \leq r \leq R} l_r \leq 1$$

.

2. Interfering live ranges cannot be assigned to the same register: $l_r + l'_r \leq 1$.
3. The third constraint states that if a live range l interferes with a live range l' preassigned to r, then $l_r = 0$.

2.3 Annotation Semantics

The offline stage generates annotations that can be used by an online stage to characterize important properties of some live ranges. The online stage may run on a target that may not match what was used to generate the annotations in the offline stage. This triggers portability problems: we address register count variations in this section, and defer the discussion of other problems to Section 4.

In the context of register allocation, the most specific portability issue is related to variations in the number of registers. To define portable annotations, it would be ideal to prove a general result about the inclusion of an optimal spill set for a given number of physical registers into one of the optimal spill sets for a lower number of registers. Unfortunately, this is not true in general. Figure 1 shows a counter example on the allocation of 5 live ranges — the horizontal bars. Every number on top of a horizontal bar denotes the cost of spilling the corresponding live range. Dashed black lines correspond to spilled live ranges. For the left graph, we assume $R = 2$ registers. For the graph on the right, $R = 1$ register only. When $R = 2$, we may optimally spill i_3 to assign i_1 and i_4 to one register and to assign i_2 and i_5 to the another one. When $R = 1$, the single optimal allocation is to spill i_2 and i_4 and to assign i_1, i_3 and i_5 to the single register. In this example we see clearly that an optimal spill set for two registers is not included in the optimal spill set for one register.
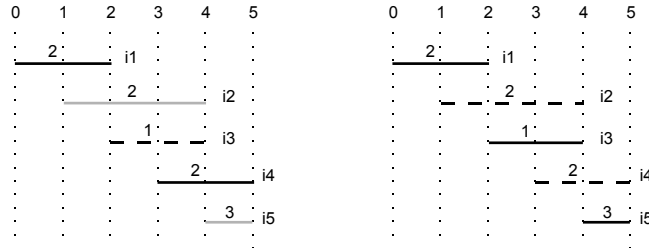

Fig. 1. Counter example to spill set inclusion

Although such an inclusion property does not always hold, we experimentally validated that only few live ranges should be spilled for $R+1$ registers but allocated for R registers. For example, considering the x86 instruction-set architecture, when moving incrementally by one register from the minimum number of registers, to a spill-free[2] number of registers for each method, inclusion property was violated for only 0.13% of the live ranges over the whole SPEC JVM

---

[2] until we reach a number of register for which allocation can be done without spilling

suite. This validates the intuition that the semantics of an allocate/spill-oriented annotation is portable across variations in the register count.

2.4 The Offline Procedure

Our split register allocation procedure derives from three key observations.

1. First, once the ILP solver finds an optimal spill set, it would be possible to directly annotate the code with the best spill set. This can lead to annotation bloat (although linear), with total annotation size potentially larger than the bytecode itself. Jones and Kamin do not address the problem [5].
2. Second, the more detailed the annotation, the more sensitive it is to low-level decisions on instruction selection and scheduling that may happen after register allocation. To make the annotation portable, it is important to focus it on semantic properties that preserve the essence of the offline optimization while maximizing independence w.r.t. post-pass optimizations in the online compilation stage. The idea here is to focus the annotation on long live ranges whose interferences do not vary much w.r.t. post-register allocation instruction selection and scheduling. Indeed, short live ranges are likely to be allocated due to their limited interferences and high-rate register usage.
3. Third, notice that a greedy allocation algorithm is typically too conservative, allocating a live range that should have been spilled or assigning an inappropriate register/color. This means that annotations should only pertain to "must-spill" information.

With those three observations in mind, we devised Algorithm 3. The intuition behind this algorithm is natural: why store annotations for live ranges on which a greedy, linear procedure can readily make the right decision?

The algorithm uses an oracle-driven version of the linear scan. Every time the greedy heuristic wishes to spill a live range which does not belong to the annotations, the algorithm forces it to spill a live range which is currently active and which belongs to the annotations. By doing so, we discover live ranges in the optimal spill set that the linear scan *cannot* find on its own.

Considering Algorithm 3, at a step where live range $V(i)$ is active (according to the allocation performed since the beginning of the method being allocated), function ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$) returns, either a live range or $\perp$ (bottom): if it returns a live range, it is the one to be spilled in order to continue allocation; if it returns $\perp$ it was possible to assign i without spilling. Function FINDACTIVELIVERANGE(*optimalSpills*) returns a currently active live range that is in the set *optimalSpills*, and set *annotation* records live ranges that will not be found by the linear scan.

The algorithm returns live ranges that will not be optimally allocated by the linear scan and keeps those as the constituents for the compressed annotations.

The final step consists of pairing the live ranges returned by Algorithm 3 with a "must-spill" tag. This pairing should be as economical as possible to represent, but it should also make sense across different targets and carry relevant allocation

---

Algorithm 3 COMPRESSANNOTATION

Input: *list*: the list of basic intervals ordered by increasing start point
Input: *optimalSpills*: the set of live ranges to be spilled as decided by the optimal allocator
1: *annotation* ← ⊥
2: **foreach:** $i \in list$ **do**
3:     *toSpill* ← ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$)
4:     **if** *toSpill* ≠ ⊥ **then**
5:        **if** *toSpill* ∉ *optimalSpills* **then**
6:           *toSpill* ← FINDACTIVELIVERANGE(*optimalSpills*)
7:           *annotation* ← *annotation* ∪ *toSpill*
8:        **end if**
9:        **if** *toSpill* ≠ $V(i)$ **then**
10:           Assign $V(i)$ to the register freed by *toSpill*
11:        **end if**
12:        Spill live range *toSpill*
13:     **end if**
14: **end for**
Return: *annotation*: the compressed annotations

---

information. For each live range l, we compute the maximal value of R for which l must be spilled, denoting it as $R_{\max}(l)$. We do not care much about offline compilation time in this study: the computation thus boils down to iterating the ILP model over decreasing values of R, pre-spilling live ranges spilled at the previous step (for $R + 1$ register) to guarantee inclusion.

Finally, annotated live ranges need to be stored in a compact persistent format, together with the bytecode program. Rather than storing every pair $(i, R_{\max}(l))$, we cluster live ranges with the same value of $R_{\max}(l)$, sort those clusters, and serialize the list of live ranges in every cluster, prepending each cluster's list with the corresponding value of $R_{\max}(l)$. We end up with separate strings, one for each size s of the register set, listing the live ranges that must be spilled for s registers and that were *not* already listed in a string associated with size s' greater than s. This way, most of the space is used to store live range names, for which we conservatively count up to 4 bytes per live range.

2.5   The Online Procedure

The online stage performs allocation based on a compact spill set collected by the offline stage, and carried as bytecode annotations.

Our online algorithm follows the steps of Algorithm 1. In addition, at every basic interval beginning, it checks whether the corresponding live range is present in the annotation. *If so, then spill it* (if the live range was not previously spilled).

This algorithm takes its roots in the decoupled allocation/assignment approach. As our experiments will confirm, the annotation-enhanced linear scan algorithm results in a much better quality allocation. Yet it does not optimally preserve the information available in the annotation and may yield spurious spill code. The reason is simple: register *assignment* on a colorable (spill-free) graph is equivalent to a graph coloring decision problem, which is NP-complete on live ranges [1]. It is *not* NP-complete with sufficient live-range splitting: linear complexity can be achieved on SSA form following a perfect elimination order — a greedy reverse post-order traversal of the SSA graph [6]. It is clearly the way to

---

Algorithm 4 ONLINEALLOCATION

Input: *list*: the list of basic intervals sorted in increasing start point
Input: *annotation*: a set of annotated live ranges
1: **foreach:** $i \in list$ **do**
2:    **if** $V(i)$ is not spilled **then**
3:       **if** $V(i) \in annotation$ **then**
4:          Spill $V(i)$
5:       **else**
6:          ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$)
7:       **end if**
8:    **end if**
9: **end for**
Return: sets of spilled live ranges and register assignments

---

go for optimality preservation, but it also implies a major engineering endeavor that has not yet been undertaken in a full-scale JIT compiler. Fortunately, the interference graphs that arise in non-SSA code are "mostly" chordal [7], which guarantees the existence of a perfect elimination order in most cases; this motivates the decoupled approach and explains the observed quality of our online algorithm.

3   Experimental Evaluation

We implemented split register allocation in JikesRVM version 3.0.1 [4], relying on CPLEX[3] for the offline resolution of optimal allocation problems.

3.1   Methodology

To assess the cost of a spill, we need to define the optimal solution we are aiming for. The cost model of the spill-everywhere problem is implemented in JikesRVM; it combines dynamic edge profiling, static use count and instruction type.

We illustrate split register allocation on SPEC JVM benchmarks. Experiments on the DaCapo benchmarks [8] could not be included at the time of the submission, but we are working hard on it. We target a 2.67GHz Intel Core 2 Quad, running in 32-bit mode, in a PC platform. This configuration is favorable to register allocation experiments due to the low number of registers, although the cost of spilling is often marginal due to out-of-order execution and to the sophisticated memory hierarchy.

Each figure was obtained from 100 individual runs of the benchmark, eliminating the 10% best and 10% worst performing points. We did not conduct a systematic statistical study of the performance distribution. Instead, we eliminated the largest source of variation by selecting a non-adaptive, aggressive (maximal optimization), profile-directed strategy (with embedded replay), using the following compilation flags:

```
-Xmx1024M -Xms1024M -X:irc:O3 -X:aos:enable_recompilation=false -X:aos:initial_compiler=opt
-X:aos:enable_replay_compile=true -X:vm:edgeCounterFile=my_edge_counter_file
```

---

[3] http://www.ilog.com/products/cplex

Split compilation is of course compatible with adaptive optimization. This methodology differs from the standard practices in that we do not run an adaptive compilation scheme [8, 9]. We claim our methodology is relevant in the context of split compilation:

– it eliminates the instability triggered by monitoring-based decisions, allowing to focus on the effect of the register allocation itself;
– an adaptive execution methodology is needed to compare the relative contributions of JIT-compilation, monitoring, garbage collection, and the effect of the optimizations themselves [9]; our methodology allows for a fair comparison nonetheless, since the online stage of the split allocation does not introduce significant overhead w.r.t. the original linear scan implementation.

Thanks to its Java API, it was easy to connect CPLEX to our framework. The total resolution time for the optimal register allocation of all SPEC JVM benchmarks — running with aggressive optimization including inlining and unrolling — takes less than 4 minutes on a Core 2 Quad processor at 2.67GHz with 4GB of RAM.

3.2 Performance Results

Benchmark	check	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack
Live ranges	86672	86870	181396	122993	93055	406348	127847	122755	220871
Annotations	77	105	214	191	98	685	315	195	236
Compression %	0.09%	0.12%	0.12%	0.16%	0.11%	0.17%	0.26%	0.16%	0.11%
Optimal spill set	2950	2984	6408	3765	3210	16821	3830	3877	6400
Remaining spills %	2.60 %	3.51%	3.34%	5.07%	3.05%	4.07%	8.23%	5.03%	3.69%
Bytecode %	0.9%	6.9%	0.9%	6.9%	3.4%	0.5%	0.9%	1.1%	0.6%

Table 1. Annotation compression

Benchmark	check	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack	average
Original JikesRVM	1.31	1.38	1.16	1.19	1.59	1.41	1.39	1.14	1.27	1.32
All live ranges Annotation	1.02	1.30	1	1.17	1.01	1.25	1.03	1.19	1.03	1.11
LIR live ranges Annotation	1.02	1.30	1	1.17	1.01	1.25	1.03	1.19	1.03	1.11
Java local variables Annotation	1.25	1.44	1.02	1.19	1.59	1.36	1.32	1.13	1.18	1.28

Table 2. Allocation cost penalty compared to optimal

Benchmark	check	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack	average
All live ranges Annotation	0%	12.0%	-1.0%	0.9%	-0.4%	-0.6%	7.5%	1.2%	0.2%	2.2%
LIR live ranges Annotation	0%	12.1%	0.2%	1.0%	-0.3%	-0.7%	5.1%	1.1%	0.2%	2.1%
Java local variables Annotation	0%	5.1%	0.8%	0.0%	-0.3%	-0.2%	-1.4%	1.1%	-0.3%	0.4%

Table 3. Wall-clock speedups of split register allocation

Table 1 illustrates the effectiveness of the annotation compression scheme: it shows the total number of live ranges (*Live ranges*); the effective number of live ranges within the annotations (*Annotations*); the *Annotations/Live ranges* ratio (*Compression*, in percentage); the number of live ranges within the optimal spill sets (*Optimal spill set*); the *Annotations/Optimal Spill set* ratio (*Remaining*

spills, in percentage); and the size overhead w.r.t. the bytecode itself (*Bytecode*, in percentage, counting 4 bytes per annotation).

Preserving the information collected in the offline stage requires at most 0.26% of the live ranges to be annotated. This is several orders of magnitude more effective than state-of-the-art approaches [5], and even comes with a formal guarantee about optimality. The addition compression row reports the benefits of Algorithm 3, and confirm its important role in making the annotation size negligible w.r.t. the bytecode size.

Table 2 Considers the analytical cost model of JikesRVM as a metric. *All live ranges annotation* correspond to annotation produced by Algorithm 3; *LIR*[4] *live ranges annotation* correspond to the intersection between the set of live ranges present in the LIR and *all live ranges annotation*; *Java local variables annotation* correspond the set of Java local variables present in *all live ranges annotation*. Table 2 shows the penalty of using the *Original JikesRVM* (linear scan), *All live ranges annotation*, *LIR live ranges annotation* and *Java local variable annotation* methods in terms of percentage of the optimal spill cost achieved by the ILP model. The JikesRVM linear-scan misses the optimal cost by 32% on average, whereas the split allocation only incurs a 11% average penalty. The case for annotation portability is validated by the very close figures for the full annotation (All live ranges) and the LIR-only annotation (*LIR live ranges*). However, when only annotating Java variables, the annotation loses its effectiveness. Using the LIR-only annotation appears as the best performance/portability trade-off.

Considering wall-clock execution time as a metric (JIT compilation plus execution time), Table 3 shows the speedup of split register allocation w.r.t. original JikesRVM's allocation algorithm. In most cases, the speedup is consistent between the optimal and split approaches. Nevertheless, the annotation does not help much on some benchmarks like `javac`. The strong improvement in the corresponding column in Table 2 indicates that the cost model itself misses the complex interplay between optimizations and important components of the target architecture.

3.3   Portability Across Variations of the Register Count

We showed there is no formal inclusion property among optimal spill sets in general. Nevertheless, for every method and among millions of live ranges, we varied R from a minimum equal to the number of pre-allocated physical registers for the method to the spill-free number of registers. Through all these allocation problems only 0.13% of the intervals spilled for $R + 1$ registers did not belong to the optimal spill set for R registers.

To make the annotation portable across variations in the register count, the compression algorithm must not eliminate a live interval that may be useless for a given number of registers but useful for a smaller number of registers. We thus run Algorithm 3 on $R = R_{\min}$ registers, where $R_{\min}$ is the minimal number of registers to enable code generation on the target.

---

[4] Low-level Intermediate Representation of JikesRVM, which does not include yet all the characteristics of the target architecture.

4 Looking Forward

So far, we ignored important issues related with the practical applicability of split register allocation.

4.1 Portability of the Annotation

Let us first consider the portability of annotation names. The names of the annotated live ranges must remain consistent between the two stages. Some annotations may be missing or extraneous, but an annotation designating a live range during the offline stage must designate to the same live range during the online stage. There are practical solutions for most portability scenarios.

1. The majority of live ranges correspond to java variables, locations in the operand stack, and other live ranges synthesized in the intermediate, target-independent passes of JikesRVM (the LIR). For those live ranges, a non-ambiguous name can be crafted that is independent of the execution context when the JIT compiler is triggered.
2. A fraction of live ranges are synthesized along the target-dependent compilation flow: address computation temporaries, conditional predicates, etc. We discard annotations regarding those live ranges when compiling for another instruction-set architecture (ISA).[5] Fortunately, besides representing a small minority, these live ranges also feature a very short temporal locality and a low degree of interference with other live ranges. This reduces the chances of impacting an important allocation decision that would result in a significant performance difference. Indeed, we showed that annotation associated with target-dependent live ranges have negligible impact on performance.

Besides the live range names, annotation properties themselves need to be portable over multiple targets: liveness properties may vary significantly over the targets if no assumption is made on the optimization flow. To achieve portability, we thus make one important assumption: optimizations selected by different JIT compilers must not vary significantly *before* the pass where annotations are loaded and attached to the intermediate representation. This restriction does not impact target-specific, post-register allocation passes like instruction selection and local scheduling.

This restriction does not solve all portability problems: reusing annotations across ISAs remains an issue. There are multiple reasons to be optimistic. Some of these are due to the context in which JIT compilation is employed, and some to the nature of the optimizations being performed before register allocation:

– Embedded system designs value the code compression and safety benefits of bytecode languages, but do not stress portability to the extreme. Although

---

[5] Such annotations remain usable when varying the register count (or the calling convention) for a given ISA.

many processors and hardware configurations may exist, Java or CLI applications are likely to run on some variant of the ARM instruction set. Varying the number of registers is important to support the ARM's compact instruction encoding options, and to support extensions like vector instructions of ARM NEON. On general-purpose platforms, an analogous situation holds, with portability issues from the 32 and 64 bit variants of the x86 instruction set, different vector instruction sets and sizes, etc.

– Bytecode languages are important for link-time optimization. Complex software architectures built of thousands of independently designed components bring many opportunities for inter-module optimization at link-time. Again, the ISA portability issue is only secondary to many of these applications.

– Beyond ISA portability, bytecode languages are used for operating system portability. In this case, the JIT compiler is minimally impacted, and annotations are expected to be robust to changes to the underlying OS.

– Eventually, the software provider may easily specialize the offline stage to generate annotations for a particular family of targets and for a particular optimization flow, tagging the annotated bytecode accordingly. This consists of constructing a (lossless) union annotation considering all live ranges that occur when compiling to the different targets. Since many live ranges will remain the same (e.g., those associated with Java local variables and constant pool, as opposed to operand stack or target-specific temporaries), the union will not significantly increase the size of the annotation.

4.2 Separate Compilation

Realistic compilation scenarios will run the offline stage separately on the different modules of the application and on its library dependences. This raises a modularity problem for any annotation-based online compilation approach.

In the context of object-oriented and functional languages, function inlining is of utmost importance to reach performance levels on par with lower level imperative implementations. It raises the following dilemma:

– what is the point of annotating code in functions that will later be inlined, since the effective interference graph will only be known after inlining;

– what is the point of annotating functions whose calling context heavily influences the internal control flow, hence the spill costs?

Our approach to modular split compilation is twofold.

No performance regression. First of all, if one module depends on a module without annotations (such as a package form the Java Development Kit), only the code in the annotated module will benefit from split compilation. This is not ideal, but not worse than the usual penalty of separate compilation in offline, static compilers. Conversely, when optimizing a "library" module, it is always possible to run a *context-insensitive* split-compilation flow, relying on a representative execution profile; this again is consistent with the traditional way of optimizing libraries in static compilation.

Multiversioning for cross-boundary optimization. Nevertheless, JIT compilation opens many opportunities for *link-time optimization*, and JIT compilers for object-oriented and functional languages do implement such advanced techniques, effectively optimizing across module boundaries (e.g., across application-library boundaries). Split register allocation is possible in this context.

First of all, a *context-sensitive annotation* of the callee can be tuned according to the most frequent calling context(s). This is only impactful when the costs of the live ranges depend on the calling context, which may be the case when the callee contains complex, data-dependent control-flow.

A more aggressive approach consists in generating multiple versions of the annotations for the most frequent call trees. For example, if a library method m_2 is frequently called from an application method m_1, the offline stage of the split register allocation may inline m_2 into m_1, optimize the resulting new method, and generate the annotation for it. This specialized version of the inlined methods can later be checked for consistency with the dynamic execution context (indeed, the library code may have changed in the mean time, or dynamic class loading may have occurred), and used directly in favor of performing all the optimizations online and dropping the (irrelevant) per-method annotation. Practical ways to implement this scheme have been proposed in the QuickSilver project [10]. This scheme has all the benefits of running a JIT compiler offline (better optimizations, lower overhead) while preserving modularity (up to dynamic class loading) and the effectiveness of split compilation.

5   Related Work

Annotations are an optional part of the Java bytecode specification from the start and are part of the *class file attributes*. They have been used in debugging and integrated development environments. Syntactic support has been added in recent versions of Java. The same applies to the ECMA-335 CLI.

Interestingly, annotation-driven JIT compilation was first directed to register allocation, with the pioneering work of Azevedo et al. [11]. This work demonstrated how to achieve performance competitive with native priority-based graph coloring allocation. Jones and Kamin [5] extended their virtual register allocation approach, dealing with correctness, calling conventions and portability (addressing variations of the number of physical registers only).

The *split compilation* term was first coined in the context of JIT vectorization [12]. Split register allocation improves on Jones and Kamin's annotation-driven approach by leveraging the decoupled allocation (spilling) and assignment (coloring) phases of register allocation. Decoupled register allocation is the key to the compactness and the portability of our annotation. The intuition behind decoupled register allocation is that the assignment problem (mapping of variables to registers with no additional spill) is very easy, as long as the cost of live-range splitting (the introduction or register moves) is neglected. This intuition is backed by the important property that spill-free assignment is always possible if the maximal number of simultaneously live variables (MaxLive) is lower than

the number of available registers. The online stage can rely on the colorability guarantee inherited from the offline stage through the annotation: these strong ties between the offline and online stages are specific to *split compilation* algorithms, as opposed to classical annotation-driven JIT compilation.

A fully decoupled approach has been used by Appel and George [13], and studied in the context of SSA-based register allocation [7, 14, 15]. Notice that recent versions of the linear scan algorithm are capable of live range splitting [16, 17]; they are implicitly based on this decoupled approach. This is not the case for the linear scan implemented in JikesRVM, and leads in practice to spurious spills (to our disadvantage), as we confirmed in our evaluation.

Pominiville et al. [18] used annotations to mitigate the performance penalty of Java pointers and arrays, and designed a generic annotation-driven compilation framework (Soot). Eventually, Krintz and Calder [3] proposed a comprehensive method to reduce the compilation time overhead through bytecode annotations, enabling rapid method selection and optimization selection, and precomputing simple method statistics.

Several papers address two additional important questions related to register allocation in JIT compilers: is there any room for performance improvement, and is it important to use a linear-time allocation algorithm? Cavazos provides an original answer relying on adaptive optimization [19]. Annotation-enhanced versions of this method would be worth investigating.

When using annotations for optimization, safety issues immediately arise because of incorrect or malicious uses. Solutions can be found in proof-carrying code [20], encryption, or correct-by construction annotation designs. We choose the latter approach, relying on annotations whose misuse can at worst lead to performance degradations.

6   Conclusion

We designed a split compilation framework dedicated to register allocation. We experimentally validated the effectiveness of split register allocation and its portability with respect to register count variations, relying on annotations whose impact on the bytecode size is negligible. This combination of results is a strong improvement over the state of the art. It was made possible by revisiting the decoupling of the spilling and coloring (a.k.a. assignment) phases.

Nevertheless, the approach still depends on the stability of the upstream optimization flow in the JIT compiler. Although this restriction is acceptable in a majority of use cases, it would be useful to design a split register allocation framework that would be more robust to changes in the optimization flow. One direction of work consists in revisiting the context of pre-pass allocation to control register-pressure by inserting additional constraints in the data dependence graph [21]. This would accommodate for scheduling (local and global) changes, and possibly for code motion, redundancy elimination and hoisting as well. Beyond register allocation, we would like to investigate the potential of

split compilation through the development, debugging and optimization cycle of software development.

References

1. Chaitin, G.J., Auslander, M.A., Cocke, A.K.C.J., Hopkins, M.E., W.Markstein, P.: Register allocation via coloring. Computer languages **6** (1981) 47–57
2. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. **16**(3) (1994) 428–455
3. Krintz, C., Calder, B.: Using annotations to reduce dynamic optimization time. In: PLDI'01, New York, NY, ACM Press (2001) 156–167
4. Alpern, B., et al.: The Jikes RVM project: Building an open source research community. IBM Systems Journal **44**(2) (2005) 399–418
5. Jones, J., Kamin, S.N.: Annotating java class files with virtual registers for performance. Concurrency – Practice and Experience **12**(6) (2000) 389–406
6. Bouchez, F., Darte, A., Rastello, F.: On the complexity of spill everywhere under ssa form. In: LCTES'07. (2007) 103–112
7. Pereira, F.M.Q., Palsberg, J.: Register allocation via coloring of chordal graphs. In: APLAS'05, Springer-Verlag (2005) 315–329
8. Blackburn, S.M.: The dacapo benchmarks: java benchmarking development and analysis. In: OOPSLA'06, New York, NY, ACM (2006) 169–190
9. Georges, A., Eeckhout, L., Buytaert, D.: Java performance evaluation through rigorous replay compilation. SIGPLAN Not. **43**(10) (2008) 367–384
10. Serrano, M., Bordawekar, R., Midkiff, S., Gupta, M.: Quicksilver: A quasi-static compiler for java. In: OOPSLA'00. (2000)
11. Azevedo, A., Nicolau, A., Hummel, J.: Java annotation-aware just-in-time (ajit) compilation system. In: Proc. ACM 1999 Conf. on Java Grande. (1999) 142–151
12. Lesnicki, P., Cohen, A., Cornero, M., Fursin, G., Ornstein, A., Rohou, E.: Split compilation: an application to just-in-time vectorization. In: GREPS'07, Brasov, Romania (September 2007)
13. Appel, A.W., George, L.: Optimal spilling for CISC machines with few registers. In: PLDI'01, Snowbird, Utah, USA, ACM Press (June 2001) 243–253
14. Hack, S., Grund, D., Goos, G.: Register allocation for program s in SSA-form. In: CC'06. (2006) 247–262
15. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In: LCPC'06. LNCS, New Orleans, Louisiana, Springer Verlag (2006)
16. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: VEE'05, New York, NY, ACM (2005) 132–141
17. Sarkar, V., Barik, R.: Extended linear scan: An alternate foundation for global register allocation. In Krishnamurthi, S., Odersky, M., eds.: CC'07. Volume 4420 of Lecture Notes in Computer Science., Springer (2007) 141–155
18. Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L.J., Verbrugge, C.: A framework for optimizing java using attributes. In: CC'01. LNCS, London, UK, Springer-Verlag (2001) 334–354
19. Cavazos, J., Moss, J.E.B., Boyle, M.F.O.: Hybrid optimizations: Which optimization algorithm to use? CC'06 (2006)
20. Necula, G.: Proof-carrying code. In: PoPL'97. (January 1997)
21. Touati, S., Eisenbeis, C.: Early periodic register allocation on ilp processors. Parallel Processing Letters **14**(2) (June 2004)