Integrated Register Rematerialization in JavaScript V8 JIT Compiler

Vahagn, Vardanyan
ISPRAS
Yerevan, Armenia
e-mail: vaag@ispras.ru

Seryozha, Asryan
ISPRAS
Yerevan, Armenia
e-mail: asryan@ispras.ru

Ruben, Buchatskiy
ISPRAS
Moscow, Russia
e-mail: ruben@ispras.ru

ABSTRACT
Nowadays most prominent dynamic languages, such as JavaScript, Python and Lua use Just-in-time (JIT) compilation technique to generate machine code. JIT compilers are limited in a complexity of optimizations they can perform without delaying the program execution. This paper is dedicated to the improvement of generated machine code quality for JavaScript programs. Our approach is to introduce a register rematerialization technique to JavaScript V8 JIT compiler. We have integrated instruction rematerialization with V8's linear scan register allocation to minimize the impact on compilation time. Both direct and reverse register rematerialization techniques are implemented in JavaScript V8 compiler and resulted in performance gain and code quality improvement for some JavaScript well-known benchmarks on ARM [1] platform.

Keywords
JavaScript, JIT, V8, register rematerialization, register allocation, spill code minimization

1. INTRODUCTION

1.1 Dynamic languages compilers
JavaScript language has gained increasing popularity due to its minimal verbosity, code maintainability, and ease of rapid prototyping. Due to increasing performance of personal computers and embedded systems, JavaScript is now used not only for executing small scripts in web browsers, but also as the main language for developing applications on some operating systems for mobile and media devices, such as Tizen [2] or FirefoxOS [3]. Dynamic properties of JavaScript language, such as presence of dynamic types and prototypes that can change during the execution make it almost impossible to compile the code effectively with static compiler without restricting the language features. Many recent works are focused on producing and improving of optimizing compilers for dynamic languages [4, 5]. Though the details of their approaches vary, the common technique used by today's state-of-the-art compilers (IonMonkey, V8, JSC LuaJIT, PyPy) is Just-in-time (JIT) compilation, where the compiler produces optimized code at runtime. To implement a tradeoff between quick startup and doing sophisticated optimizations, JavaScript engines usually use multiple tiers: lower tier JITs generate less efficient code, but can start almost immediately (e.g. even with interpretation), while higher tier JITs aim at generating effective code for hot places, but at the cost of long compilation time. JIT optimization takes advantage of the program's runtime behavior collected from the profiler to explore more optimization opportunities. Based on the assumption of the steady states, optimizations such as type specialization, inline caching for dynamic dispatch and profile-directed method inlining are applied.

1.2 Register allocation and rematerialization
Although being an old computer problem, register allocation remains an important optimization to address CPU/memory performance gap. There are many works dedicated to register allocation problem [6, 7]. One of the well-known techniques is register allocation through graph coloring. Unfortunately, graph coloring and other aggressive global register allocation algorithms are computationally expensive and using them in JIT compiler may have big impact on compilation time and delay program execution. Therefore, V8 compiler uses global register allocation algorithm called linear scan [8], which is simple, efficient, produces relatively good code and has less impact on compilation time than the traditional register allocation algorithms. During register allocation, when register pressure (the number of simultaneously live variables) is too high one can spill the data of one register into memory and reload it later when it needed, or alternatively one can try to recompute it from values currently still alive. The latter technique is called *rematerialization*.
In this paper, we describe the implementation of both direct and reverse register rematerialization in JavaScript V8 compiler. We have implemented the rematerialization technique integrated with linear scan register allocation and using advantage of V8's internal structures to minimize the impact on compilation time.

2. RELATED WORK
There are a number of works about register rematerialization problem [9, 10]. This optimization is implemented in several leading industrial compilers such as GCC [11] and LLVM [12]. Several recent works are dedicated to register reverse rematerialization technique [10]. Many of these algorithms use data dependency graphs (DDG) and register reuse chains to discover excessive registers and to detect rematerializable values. Once rematerialization decision is made, DDG transformation is done in order to move rematerializable values after the excessive nodes. Taking into account that V8 uses control flow graph (CFG) [13] as intermediate representation for its optimizing compiler (Crankshaft), building new data dependency graph, constructing reuse chains and performing graph transformation can result in big negative impact on compilation time, in fact, it can even cause program performance degradation. In order to reduce memory access instructions count without negative impact on compilation time we are using v8's linear scan register allocation and its internal structures to implement register rematerialization.

3. V8 JIT compiler Multi-Tier Architecture

V8 compiles source code into machine code using two separate compilers as shown in Fig. 1. The first compiler, Full-Codegen, compiles source code directly into machine code without any optimizations in order to produce code quickly. The second compiler, Crankshaft, is slower and produces optimized machine code. V8 first parses source code into abstract syntax tree (AST) and uses Full-Codegen compiler to produce code quickly without any optimizations. While executing unoptimized code, program profile information data is collected, such as type information, inline caches [14], etc. At the same time runtime profiler samples JavaScript code in order to determine hot (frequently executed) functions. When hot function is detected, V8 starts compiling that function using Crankshaft compiler. Crankshaft translates AST code into control flow graph (CFG) with SSA-like representation named Hydrogen. This representation is used to implement many well-known optimizations, such as dead code elimination, constant propagation, common subexpression elimination, bounds redundant check elimination, loop invariant code motion, etc [15]. Collected type and other runtime information allows Crankshaft to optimize functions speculatively, under the assumption that certain properties of the functions will not change during the next run. If the optimized code encounters a case that it cannot handle (for example, when type of value of the variable does not match profile information), it bails to unoptimized code (Full-Codegen). This transition is called on-stack replacement.

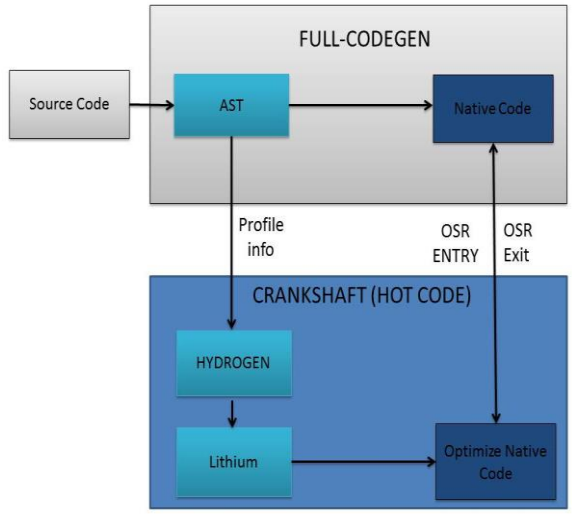

Figure 1. V8 Multi-Tier JIT Architecture

When all optimizations are performed, Crankshaft transfers Hydrogen graph to low-level, machine-dependent intermediate representation called Lithium. This representation is mainly used for the register allocation. Unlike Hydrogen, which is in SSA form, the Lithium form is closer to three-address code, with labels and gotos. Each Lithium instruction is represented by its output operand, input operands and temporary operands. These operands are initially declared with a number of constraints. For example, if the result of some instruction is a double number, its output will be declared with "operand must be in a double-precision register" constraint. Later, a register allocator considers the constraints and live intervals, and allocates each operand to a specific register or memory address. Code generation happens after register allocation.

4. Linear scan register allocation

Register allocation is the task of assigning local variables and temporary values to physical registers of a processor. The register allocation phase of code generation is often a bottleneck, and yet good register allocation is necessary for making today's processors reach their peak efficiency. It is thus important to understand the trade-off between the speed of register allocation and the quality of the resulting code.

A linear-scan register allocation [8] algorithm directs the global allocation of register candidates to registers based on a simple linear sweep over the program being compiled. It uses liveness information to find an appropriate candidate for allocating to physical register. This approach to register allocation makes sense for systems, such as those for dynamic compilation, where compilation speed is important. A live interval of a value **v** is the range *[i, j[* such as **i** is the instruction where **v** is first defined and **j** is the position where **v** ends living. Live intervals are obtained from variable liveness analysis. Fig 2 shows live intervals computed for several instructions.

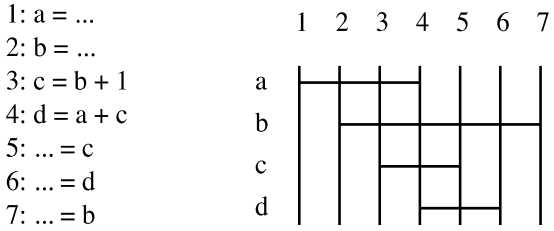

Figure 2. A simple instructions sequence and its live intervals

The linear scan algorithm first sorts all live intervals in ascending order of their starting point. The basic algorithm processes all live ranges maintaining list of active intervals (the intervals that overlap the start point of current interval). For every live interval **i** the algorithm performs these steps:

- Initially all registers are free.
- If there is live interval **j** in active that is already expired before **i** begins (i.e**. j.end ≤ i.beg**), remove it from active and add **j.reg** to the set of free registers.
- If there are still free registers, assign one of them to **i** and add **i** to active. If there are no free registers, spill the interval with the farthest end point among **i** and all the intervals in active. If an interval from active was spilled, then assign its register to **i**, and add **i** to active.

Assuming that we have two physical registers, **r1** and **r2**, algorithm processes the intervals as shown in Fig. 3.

The above algorithm describes the basic idea of linear scan register allocation. There are many improvements of this algorithm including live intervals splitting, more advanced spilling heuristic, hoisting spill code out of loops (if possible), etc. When a spilling decision is made, the algorithm can split the interval in a such way that some parts of the interval can still reside in physical register, e.g. the interval can be split across loops, or other hot code regions, so the part of interval which is in hot region can reside in physical register. The refined version of linear scan algorithm (called second-chance binpacking) was described by Traub et al. [16]. Although more complicated, this algorithm results in a better usage of registers.

Interval	Free	Active	Action
a	r1, r2	-	Assign r1 to a; make a active
b	r2	a_{r1}	Assign r2 to b; make b active
c	-	a_{r1}, b_{r2}	Spill b since it ends after c; Make r2 free
-	r2	a_{r1}	Assign r2 to c; make c active
d	-	a_{r1}, c_{r2}	Remove a from active (expired); make r1 free
-	r1	c_{r2}	Assign r1 to d; make d active

Figure 3. Result of linear scan algorithm

5. Integrated register direct and reverse rematerialization

Though linear scan register allocation is one of the best algorithms to use in JIT compilers, it still has to insert spill code to resolve all architecture and program constraints (spill code must be inserted when register pressure is high, some architectures require specific registers for some operations, procedure calls can rewrite all register, etc.). One of the important features of good register allocator is to minimize spill code. However, most problems of spill code minimization are known to be NP-complete [17]. Linear scan allocation uses live interval splitting and various spilling heuristics (e.g. spilling the interval with the largest end point) to reduce spill code. Another method of spill code minimization is rematerialization. Recomputation of some value **v** can be performed from its input operands still stored in registers; recomputation is done in the same way as specified in the program. But there is a part of information of **v** carried by other values **{w_i}** that were computed from **v**.

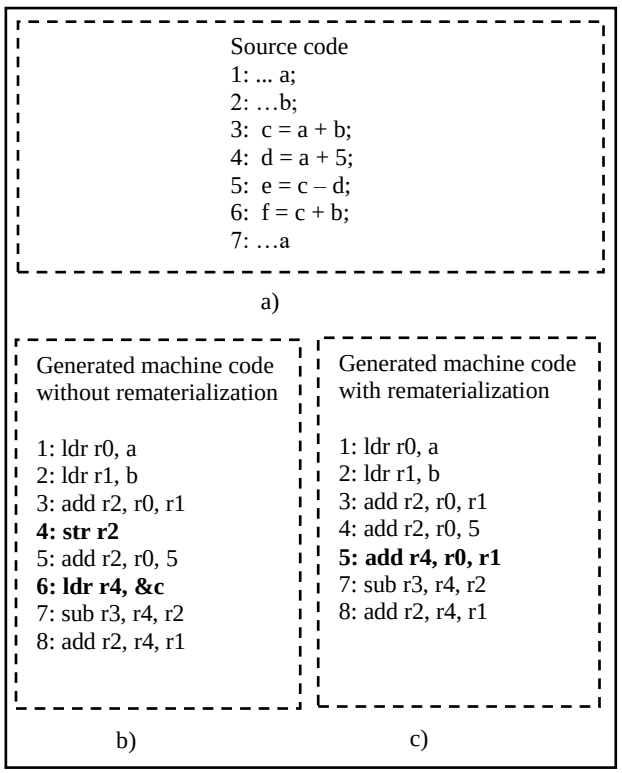

Figure 4. Example of register direct rematerialization

Hence, this gives new opportunities for recovering **v** value: undoing the computation from **{w_i}** values, or in other words, reversely computing **v**.

An example of direct rematerialization is shown in Fig. 4. Let's suppose that there are four general purpose registers (*r0-r3*), *r4* is a scratch register. It is used to hold reloaded spilled values immediately before their use points. Value **c** can be recomputed from values **a** and **b** because they are alive during the lifetime of **c**. As shown in Fig. 4.c we can replace two memory access instructions with one cheaper add instruction.

In our implementation, rematerialization is performed after register allocation because of the following reasons:

- Before register allocation there is no information about certain register requirements or excessive register demands.
- Implementing rematerialization before register allocation can insert additional constraints, create more dependencies and extend live intervals for values.

After register allocation all the information about excessive registers, spilled live intervals and rematerializable values is available.

When making rematerialization decision it is important to take into account whether recomputation of some value will be cheaper than reloading it from memory. For example, on several ARM processors it is cheaper to reload the spilled value than to recompute it using multiply instruction if that value is in L1 data cache [1]. In our current approach, addition, subtraction, shift and bitwise instructions are supported for direct rematerialization (all these instructions are cheaper than memory access instructions on ARM platform). Multiply and division instructions are supported partly, in the cases when they can be implemented using shift instructions. Operands of addition and subtraction instructions can be reversely recomputed if the output operand and one of the input operands are available. However, this rule is not common for all binary operations. For instance, the multiply operation needs at least one additional resulting bit for determining which of both operands was 0 if the result is 0. In our current implementation, only addition and subtraction instructions are supported for reverse rematerialization. Our approach can be logically divided into five steps.

The first step of our approach is to detect all possible rematerializable values. We have modified Hydrogen to Lithium transformation phase to assign both direct and reverse rematerialization information to corresponding instructions. In the Lithium level value **c** (Fig. 4.a) is represented as an output operand of instruction 3 (*OutputC3*) and an input operand of instructions 5 (*InputC5*) and 6 (*InputC6*) (all these operands will be mapped to the same register or memory address). *OutputC3* can be recomputed by direct rematerialization from **a** and **b**, *InputC5* can be recomputed by reverse rematerialization from **e** and **d**, *InputC6* by reverse rematerialization from **f** and **b**. Rematerialization information is assigned to all these operands (*LOperands*). The second step is to propagate obtained information to the variables' live intervals. Assuming that instruction 6 is the last use of value **c,** the live interval of **c** is *[3, 7[* with use positions 5 and 6. During live intervals building phase we pass rematerialization information from *LOperands* to live intervals, so the live interval of **c** will contain information from *OutputC3*, *InputC5* and *InputC6*. Hence, each use position of that interval (each use of value **c**) has three possible options to be recomputed. The third step is to modify spilling heuristic of register allocation in order to increase the possibility of

spilling live intervals with rematerialization information on the one hand, and decrease possibility of spilling the live intervals that are being used for recomputation of rematerializable values on the other hand. After this step, the values with rematerialization information are more likely to be spilled into memory, and the values, which are used for recomputation of spilled values, are more likely to reside in machine registers. The forth step is performed after register allocation. We iterate through all spilled live intervals which are marked as rematerializable and try to validate rematerialization information attached to their use positions, i.e. proving that the operands needed to recompute value at the given use position are alive and still reside in machine registers. After linear scan register is completed, all the necessary information for validating rematerialization information can be obtained from the variable live intervals (e.g. if certain variable is alive at the given point, if it is allocated into memory or register)

Finally, we have modified the code generation phase, so operands with valid rematerialization information are recomputed instead of being reloaded from memory. As mentioned above V8's linear scan allocator can split live interval of some value in a way that the parts of the interval may be allocated to different memory locations. After allocation is completed, allocator must insert move instructions (register to register, register to memory, memory to register or memory to memory) on the boundaries of split intervals. Such move instructions are also inserted for resolving control flow constraints (e.g., the same value can flow to basic block from two or more different edges). Our implementation of register rematerialization takes advantage of these cases as well and can recompute values on the live intervals boundaries, too.

6. Experimental Results

We have tested our approach on several JavaScript well known benchmarks such as SunSpider [18], Kraken [19] and Octane [20]. In SunSpider benchmark, we have managed to replace up to 30 memory access instructions to cheaper arithmetic instructions. On some tests, we have managed to replace 5-8 memory access instructions in nested loops, but unfortunately, these improvements of generated machine code have no effect on the performance of SunSpider benchmark. The reason is that SunSpider tests execution time is very short and they do not have heavy loops, so replacing memory access instructions to arithmetic ones even in the nested loops brings no impact on overall performance. On the contrary, Kraken benchmark's tests contain many complex nested loops. Our algorithm could replace up to 8 memory access instructions from nested loops in Kraken's *audio-beat-detection test*, which brings approximately 5% performance improvement on ARM platform. Similarly, replacement of 8 loads from memory instructions from nested loops in Kraken's *audio-fft* test brings about +7% performance improvement on this test. About 5 memory access instructions are replaced in *stanford-crypto-ccm* and *imaging-darkroom* tests. On Octane benchmark we have managed to replace up to 200 memory access instructions. Approximately 5-10 memory access instructions have been replaced in *CodeLoad*, *Zlib* and *Typescript* tests, 20-30 instructions in *Crypto*, *Pdf* and *NavierStokes* tests, 45 in *Gameboy* and more than 70 load instructions in *Mandreel* test. It brings approximately 1-2% performance improvement on *GameBoy* and 3-4% on *Mandreel* tests.

Due to our implementation of rematerialization adds only one linear pass through all spilled live intervals, the impact on compilation time is minimal. In fact, there was no observed performance degradation on JavaScript benchmarks we have tested.

7. Conclusion

We have developed both direct and reverse register rematerialization technique in V8 open source JavaScript engine for ARM platform. Experimental results on popular JavaScript benchmarks (Octane, SunSpider, Kraken, Browsermark) show up to 7% speedup on certain tests without any performance degradation on the others.

We plan to continue our work by adding rematerialization support for other platforms (x86, etc).

REFERENCES

[1] ARM architecture. http://infocenter.arm.com

[2] Tizen platfrom website. http://tizen.org/

[3] Mozzila website. https://www.mozilla.org

[4] Google Inc. V8 - Project Hosting on Google Code. http://code.google.com/p/v8/

[5] A. Gal; B. Eich; M. Shaver; D. Anderson; D. Mandelin; M. R. Haghighat; B. Kaplan; G. Hoare; B. Zbarsky; J. Orendorff; J. Ruderman; E. Smith; R. Reitmaier; M. Bebenita; M. Chang; M. Franz, "Tracebased just-in-time type specialization for dynamic language," In Proceedings of the Conference on Programming Language Design and Implementation, pages 465–478, 2009

[6] G. Chaitin et al., "Register Allocation via Coloring," Computer Languages, 6, pp. 47–57, 1981

[7] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," SIGPLAN Notices, 17(6):201–107, June 1982

[8] M. Poletto, and V. Sarkar. "Linear scan register allocation." ACM Transactions on Programming Languages and Systems (TOPLAS), v.21 n.5, p.895-913, Sept. 1999

[9] Mukta Punjani. "Register rematerialization in gcc." In GCC Developers' Summit 2004, June 2004

[10] Bahi, M., Eisenbeis, C. "Rematerialization-based register allocation through reverse computing." In: Proceedings of the 8th ACM International Conference on Computing Frontiers, CF '11, pp. 24:1– 24:2. New York, NY, USA, ACM (2011)

[11] Gnu Compiler Collection website. http://gcc.gnu.org/

[12] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization."— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL

[13] Bruce A. Cota, Douglas G. Fritz, Robert G. Sargent "Control flow graphs as a representation language" Simulation Conference Proceedings, 1994. Winter

[14] Urs Hölzle, Craig Chambers, David Ungar "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches" ECOOP '91 Proceedings of the European Conference on Object-Oriented Programming, 21-38, 1991

[15] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman "Compilers : principles, techniques, and tools", 583-703, 2006.

[16] Traub, O., Holloway, G., Smith, M.D.: "Quality and Speed in Linear-Scan Register Allocation." Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation 142-151, 1998

[17] Florent Bouchez. "A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases." PhD thesis, ENS Lyon, 2009

[18] SunSpider benchmark website: http://www.webkit.org/perf/sunspider/sunspider.html

[19] Kraken benchmark website: http://krakenbenchmark.mozilla.org/

[20] Octane benchmark website: https://developers.google.com/octane/