Linear Scan Register Allocation in the HiPE Compiler

Erik Johansson and Konstantinos Sagonas

Computing Science Department, Uppsala University, Sweden
{happi,kostis}@csd.uu.se

Abstract. We describe and evaluate the *linear scan register allocator* proposed by Poletto and Sarkar, and extensively report on its implementation and behaviour in the context of a native code compiler for the concurrent functional programming language Erlang. We also investigate how different options of the basic algorithm and of the compilation process as a whole affect compilation time and quality of produced code.

1 Introduction

During the last years, we have been developing HiPE, a High-Performance native code compiler for the concurrent functional programming language Erlang [1]. Erlang has been designed to address the needs of large-scale soft real-time control applications and is successfully used in telecommunications industry products. The HiPE system significantly improves the performance characteristics of Erlang programs by allowing selective, user-controlled, "just-in-time" compilation of bytecode-compiled Erlang functions to native code. As reported in [6], HiPE currently outperforms all other implementations of Erlang.[1]

Achieving high performance has not come without a price: compilation times are higher than compilation to virtual machine code. Even though HiPE is currently not focussing on dynamic compilation—the compiler is usually used either at the interpreter or through `make`—compilation times are a significant concern in an interactive development environment that Erlang advocates. Investigating where the HiPE compiler spends time, we have found that HiPE's register allocator, which is based on a variant of graph coloring, is often a bottleneck being responsible for 30–40% of the total compilation time. (This is not atypical; experience reported in the literature is similar.) Worse yet, for functions containing a large number of temporaries, the memory needed for the interference graph is often prohibitive. So, we were looking for a more well-behaved register allocator.

Around the same time, we found out about the recently proposed *linear scan register allocator* [9]. The method was quite simple and seemed extremely intriguing: with relatively little machinery—and more importantly by avoiding the quadratic cost of heuristic-based coloring—almost equally efficient code would

[1] HiPE is available (currently only for SPARC architectures) as open-source from www.csd.uu.se/projects/hipe. HiPE is being integrated in Ericsson's Erlang implementation and will be included in the upcoming Open Source Erlang release.

be generated. At first, we were a bit skeptical whether the results would carry over to HiPE, because Erlang—or a functional language in general—programs may not benefit to the same extent from techniques used for compiling C or 'C [8] programs, but more importantly because HiPE precolors a relatively large set of physical registers which are heavily used by the underlying virtual machine; this increases register pressure and significantly departs from the contexts in which linear scan has previously been used [8, 9]. Despite our skepticism, we were willing to give it a try. In about two weeks, a straightforward implementation based on the description in [9] was ready, but this was the easy part. The results were encouraging, but we were curious to experiment with various options that [9] also explores, sometimes leaves open, or suggests as future research. In addition, we discovered that, in a purely functional language (i.e., without destructive updates), careful choice of data structures (and some fine-tuning) was needed to get a decent performance. Overall, our implementation effort ended up lasting four months; this includes time for fine-tuning the other allocators used here.

This paper, besides documenting our implementation (Section 3) and reporting on its performance characteristics (Section 4), describes our experience in using linear scan register allocation in a context which is different from those previously tried. It also presents results from investigating different options to linear scan and extensively reports on their effectiveness and on the trade-offs between speed of register allocation and quality of the resulting code (Section 5). We begin with a description of various schemes for register allocation.

2 Global Register Allocation

2.1 Graph coloring register allocation & iterated register coalescing

The idea behind coloring-based allocation is to formulate register allocation as a graph coloring problem [3], by representing liveness information with an *interference graph*. Nodes in the graph represent variables that can be allocated to registers. Edges connect variables that are simultaneously live and thus cannot use the same physical register. By using as many colors as physical registers, the register allocation problem can be solved by assigning colors to nodes in the graph such that all directly connected nodes receive different colors.

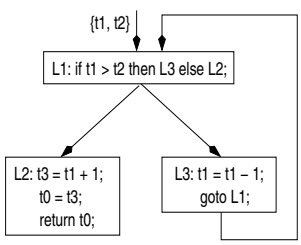
The classic method by Chaitin *et al* [3, 4] iteratively builds an interference graph and heuristically attempts to color it. If the graph is not colorable, nodes are deleted from the graph, spilled to memory, until the graph becomes colorable. Iterated register coalescing [5] is a more aggressive coloring-based technique aiming at eliminating many redundant `move`s: When the source and destination node of a `move` do not interfere (i.e., are not directly connected in the graph), these nodes can be coalesced into one, and the `move` instruction can be removed.

In practice, coloring-based allocation methods usually produce good code. However, the cost of register allocation is dominated by the construction of the interference graph, which can be quadratic in the number of nodes. When compilation time is a concern, such as in just-in-time compilers or interactive development environments, graph coloring may not be the best choice.

2.2 Linear scan register allocation

The linear scan allocation algorithm [9] is simple to understand and implement. It is based on the notion of the *live interval* of a temporary, which is an approximation of its liveness region. The live interval of a temporary is defined so that the temporary is dead at all instructions outside the live interval; the temporary might also be dead at some instructions within the interval. The idea is that the allocator can use this information to easily determine how these intervals overlap and assign temporaries with overlapping intervals to different registers.

The algorithm can be broken down into the following four steps: (1) order all instructions linearly; (2) calculate the set of live intervals; (3) allocate a register to each interval (or spill the corresponding temporary); and finally (4) rewrite the code with the obtained allocation. We review the steps, using Fig. 1 as example.



(a) Control-flow graph.

```
1 : L1:if t1 > t2 then L3
        else L2;
2 : L3:t1 = t1 - 1;
3 :     goto L1;
4 : L2:t3 = t1 + 1;
5 :     t0 = t3;
6 :     return t0;
```

(b) Linearization 1.

```
1 : L1:if t1 > t2 then L3
        else L2;
2 : L2:t3 = t1 + 1;
3 :     t0 = t3;
4 :     return t0;
5 : L3:t1 = t1 - 1;
6 :     goto L1;
```

(c) Linearization 2.

Fig. 1. Control-flow graph and two of its possible linearizations.

Ordering instructions linearly As long as the calculation of live intervals is correct, an arbitrary linear ordering of the instructions can be chosen. Different orderings will result in different approximations of live intervals, and the choice of ordering might impact the allocation and the number of spilled temporaries. An optimal ordering is one with as few contemporaneous live intervals as possible, but finding this information at compile-time is time-consuming. It is therefore important to *a priori* choose an ordering that performs best on the average. Poletto and Sarkar [9] suggest the use of a depth-first ordering of instructions as the most natural ordering and only compare it with the ordering in which the instructions appear in the intermediate code representation. They conclude that these two orderings produce roughly similar code for their benchmarks. We have experimented with other orderings and discuss their impact in Section 5.1. In our example, the control-flow graph of Figure 1(a) can be ordered in many ways; Figures 1(b) and 1(c) show two possible orderings.

Calculation of live intervals Given a linear ordering of the code, there is a minimal live interval for each temporary. For temporaries not in a loop, this

interval starts with the first definition of the temporary and ends with the last use of the temporary. For temporaries live at the entry of a loop, the interval must be extended to the end of the loop. The optimal interval can be found by first doing a precise liveness analysis and then by traversing the code in linear order extending the intervals of each temporary to include all instructions where the temporary is live. For the first linearization of our example, a valid set of live intervals would be:

$$\texttt{t0} : [5,6], \quad \texttt{t1} : [1,4], \quad \texttt{t2} : [1,3], \quad \texttt{t3} : [4,5]$$

and for the second linearization, a valid set of live intervals would be:

$$\texttt{t0} : [3,4], \quad \texttt{t1} : [1,6], \quad \texttt{t2} : [1,6], \quad \texttt{t3} : [2,3]$$

In the first set of intervals, `t2` is only live at the same time as `t1`, but in the second one, `t2` is simultaneously live with all temporaries.

There are alternatives to doing the somewhat costly liveness analysis that give correct—but sub-optimal—live intervals. [9] describes one such alternative based on strongly connected components in the control-flow graph. We will look closer at this alternative in Section 5.2.

Allocation of registers to intervals When all intervals are computed, they are ordered on increasing start-points so as to make the subsequent allocation scan efficient. For our first linearization this would result in:

$$\texttt{t1} : [1,4], \quad \texttt{t2} : [1,3], \quad \texttt{t3} : [4,5], \quad \texttt{t0} : [5,6]$$

Allocation is then done by keeping a set of allocatable free physical registers (FreeRegs), a set of allocated temporaries (Allocated), and a list of active intervals ordered on decreasing end-points (Active), while traversing the start-point-ordered list of intervals. For each interval in the list, that is, for each temporary $\texttt{t}_i$ (with interval $[t_start, t_end]$) do:

- Free all registers in Active which are assigned to temporaries whose interval ends before the current interval (i.e., before t_start).
- If there is a free register, r, in FreeRegs, allocate $\texttt{t}_i$ to r, remove r from FreeRegs, add $\texttt{t}_i \mapsto r$ to Allocated, and insert the interval $[\texttt{t}_i, t_start, t_end]$ to Active (sorted on t_end).

  If, on the other hand, there are no free registers and the end-point a_end of the first temporary `a` in Active is further away than the current (i.e., $a_end > t_end$), then spill `a`, otherwise spill $\texttt{t}_i$. By choosing the temporary whose live interval ends last, the number of needed spills is hopefully kept low. (Another way of choosing the temporary to spill is discussed in Section 5.4.)

Rewrite of the code Finally, when the allocation is completed, the code is rewritten so that each use or definition of a temporary involves the physical register where the temporary is allocated to. If the temporary is spilled, a `load` or `store` instruction is added to the code and a precolored physical register is used instead of the temporary.

3 Implementation of Register Allocators in HiPE

3.1 Graph coloring and iterated register coalescing allocators

The graph coloring register allocator was implemented long ago and has been in daily use in the HiPE system ever since. It is a variant of [2], is well-tested, and uses efficient data structures. Its biggest advantage is that it seldom needs to spill on the SPARC. The iterated register coalescing allocator was implemented with the main goal to get a working register allocator that faithfully follows the algorithm described in [5] so as to establish a lower bound of the number of spills that are possible in the benchmarks. As it is not clear whether it will be part of HiPE, the emphasis was on correctness and maintainability, not on compilation speed: this allocator currently uses sub-optimal data structures and, as a result, compilation times do not do justice to the algorithm and will not be reported.

3.2 Linear scan register allocator

As mentioned, the linear scan register allocator was first implemented based on the description in [9]. Subsequently, we experimented with various options and tuned the first implementation considerably. In this section, we only describe its default implementation by looking at how each step of the algorithm is implemented in HiPE; in Section 5 we will significantly improve on [9] by describing and quantifying the impact of the alternatives we tried.

Ordering instructions linearly Since there is no reason to reorder the instructions within a basic block, we just order the basic blocks. The default ordering of blocks is depth-first ordering (also called reverse postorder).

Calculation of live intervals Each live interval consists of a start position and an end position: these are instruction numbers corresponding to the first definition and last use of the register in a breadth-first traversal of the control-flow graph. We use liveness analysis to find out the live-in and live-out sets for each basic block. This information is then used to set up the live intervals for each temporary by traversing the set of basic blocks. All temporaries in the live-in set for a basic block starting at instruction i have a live interval that includes i. All temporaries in the live-out set for a basic block ending at instruction j have a live interval that includes j. Furthermore if a temporary, `t`, is included in either the live-in or the live-out set but not in both, then the live interval of `t` needs to be extended to either the first instruction defining `t` or the last instruction using `t` within the basic block.

Allocation of registers to intervals The allocation is performed by traversing the list of intervals sorted on increasing start-points. During the traversal we use four data structures:

Intervals This is a representation of the interval structure calculated in the previous step, sorted on the start-points.

FreeRegs Physical registers that are neither allocated to any temporary nor precolored.

Active A list of intervals sorted on increasing `EndPoint`, used to keep track of which temporaries are allocated to which physical register for what period, so as to deallocate physical registers when the allocation has passed the end-point. It is also used to find the temporary with the longest live interval when a temporary needs to be spilled.

Allocation An unsorted list containing the final allocation of temporaries to registers or to spill positions.

For each interval in Intervals, the allocation traversal then does the following:

1. Move each interval in the Active list that ends before `EndPoint` to the Allocation structure. Also add the physical register assigned to the interval to the FreeRegs list.
2. Find an allocation for the current interval:
   - If there is a physical register in the FreeRegs list then assign this register to the current interval by inserting the interval and the physical register into the Active list, and by removing the physical register from the FreeRegs list.
   - If there is no free register then spill the interval with the furthest `EndPoint` and move this interval to the Allocation list. If the spilled interval is not the current one, then assign the physical register of the interval that was spilled to the current interval.

Rewrite of the code When all intervals are processed, all instructions are rewritten using the Allocation mapping. Instructions that use a spilled temporary are rewritten to a read from (or write to) memory.

3.3 A naïve register allocator

To establish a base line for our comparisons, we have also implemented an extremely naïve register allocator: it allocates all temporaries to memory positions and rewrites the code in just one pass. Every use of a temporary is preceded by a `load` of that temporary to a register, and every definition is followed by a `store` to memory. This means that the number of spills, that is the number of added `load` and `store` instructions, is equal to the number of uses and defines in the program. Each spill will result in three instructions on the SPARC, two to set up the base address to the spill area and one to do the actual `load` or `store`. This register allocator is very fast since it only needs one pass over the code, but on the other hand the added `load`s and `store`s increase the code size which in turn increases the total compilation time. Obviously, we recommend this "register" allocator to nobody! We simply use it to establish a lower bound on the register allocation time and an upper bound on the number of spills in order to evaluate the performance and effectiveness of the other register allocators.

4 Performance Results

We evaluated the register allocator implementations by compiling and running a set of benchmarks. All four register allocators are integrated in the HiPE system (version 0.93) and a compiler option indicates which one to use. We have compiled the code in the same way before and after applying each allocator: The code is compiled to our internal representation of SPARC code as a control flow graph. Some simple optimizations are applied to the intermediate stages, such as constant propagation, constant folding, dead code elimination, and removal of unreachable code; see e.g. [7]. The mapping of temporaries into the set of physical registers creates false dependencies, since different temporaries are mapped to the same physical register. This means that the register allocation may possibly interfere with some optimization passes that in HiPE come after it: one allocation may prevent some optimizations that another would allow, and vice versa. Thus, to make the effect of spilling clear, and not be clouded by the effect of coloring on e.g., delay slot filling, we apply no further optimizations after register allocation.

There is, however, one small difference in the code generated by the different allocators: The graph coloring and linear scan allocators use a `multimove` instruction (which performs several `move`s at the same time), while the coalescing allocator does not. For the coalescing allocator this instruction is expanded to several `move` instructions before allocation (many of which are eliminated by coalescing), while for the graph coloring and the linear scan implementation this instruction is expanded after allocation. This results in slightly different code.

The system we used for our measurements is a 2-processor Sun Microsystems Enterprise 3000 with 1.2 GB main memory running SunOS 5.7. Each processor is a 248 MHz UltraSPARC-II. (However, HiPE uses only one processor at a time.)

4.1 The benchmarks

Some of the benchmarks (**decode**, **eddie**) have been chosen from the "standard" set of Erlang benchmarks because they incur spilling when compiled with linear scan. We have also included some large modules of the HiPE system that are troublesome for some register allocators.

spillstress A synthetic benchmark consisting of a recursive function with several continously live variables; its purpose is to stress the register allocators.

smith The Smith-Waterman DNA sequence matching algorithm.

life Conway's game of life on a 10 by 10 board where each square is implemented as a process.

decode Part of a telecommunications protocol.

huff A Huffman encoder compressing and uncompressing a short string.

prettypr Formats a large program file for printing, using a strict-style context passing implementation.

estone A standard Erlang/OTP benchmark that stresses all parts of an Erlang implementation (measures the number of estones of an Erlang system).

eddie An HTTP parser which handles http-get requests.

translate The part of the HiPE compiler that translates JAM bytecode into intermediate code. The program consists of a large function handling different combinations of instructions. This very large function is the reason we started examining different register allocators in the context of HiPE. To get measurable execution times, we run this benchmark 10 times.

Sizes of benchmark programs (lines of source code, the number of generated SPARC instructions before register allocation, and the total number of basic blocks and temporaries in each benchmark) are presented in Fig. 2. We note that choosing benchmark programs was not easy as many standard benchmarks do not spill at all using linear scan.

Benchmark	Lines	Instrs	BasBlks	Temps
spillstress	99	2771	105	503
smith	102	2877	318	858
life	191	3140	339	1038
decode	391	4782	521	1255
huff	186	6566	737	1874
prettypr	1051	12106	1065	3515
estone	1131	22374	2414	6379
eddie	1882	30209	3587	9263
translate	1362	30890	2057	6775

Fig. 2. Sizes of benchmark programs.

4.2 Compilation times

We have measured both the time to perform register allocation and the time to complete the entire compilation for each program. The results are presented in Table 1 where the Comp columns stand for the total compilation time, and the RA columns stand for the time spent in the register allocator. The table also shows the percentage of compilation time spent in register allocation (RA%).

Note that even though for the naïve implementation the allocation times are usually the shortest, for some benchmarks the total compilation time is the longest. This is due to the large number of extra instructions which are added to the code due to spilling.

	Naïve			**Linear Scan**			**Graph Coloring**		
	Comp	RA	RA%	Comp	RA	RA%	Comp	RA	RA%
spillstress	1.5	0.2	11.6	1.3	0.2	16.8	6.0	4.9	81.4
smith	1.9	0.2	11.6	1.7	0.3	18.4	2.2	0.7	33.0
life	2.1	0.3	14.4	1.9	0.4	18.9	2.2	0.7	30.6
decode	4.8	0.5	10.0	4.4	0.6	12.5	6.3	2.5	39.8
huff	6.8	0.7	10.7	6.3	0.8	13.2	7.8	2.8	35.4
prettypr	61.5	2.2	3.5	57.3	1.8	3.2	71.8	15.2	21.1
estone	15.4	1.7	10.7	14.2	2.4	16.7	17.4	5.8	33.4
eddie	68.6	2.8	4.1	22.2	3.5	15.5	26.2	8.7	33.3
translate	269.8	8.8	3.3	257.4	4.9	1.9	313.9	61.8	19.7

Table 1. Total compilation time in seconds (Comp), register allocation time (RA), and percentage of total compilation time spent on register allocation (RA%).

As one can see from Table 1, the complexity of the graph coloring register allocator is not directly dependent on what one could naïvely consider as the 'size' of the program. Instead the complexity is dependent on the number of edges in the interference graph, which is high for the **spillstress** benchmark. On the other hand, the linear scan allocator is not affected by the number of

simultaneously live temporaries; the allocation time is instead dominated by the time to traverse the code. The total number of temporaries has some impact on the register allocation time though. The compilation times of **prettypr** and **translate** programs stick out a bit since they each contain one large function that is responsible for most of the compilation time, while **estone** and **eddie** consist of a large number of small functions.

Compilation-time-wise, linear scan performs well: compared to graph coloring, the time for register allocation is significantly reduced, and pathological cases such as **spillstress** or **translate** are avoided. In fact, linear scan is sometimes even faster than the naïve algorithm: Register allocation times include time for performing the rewrite of the code with the allocation (step 4) and due to excessive spilling this code is larger for naïve than it is for linear scan; see Table 2.

4.3 Execution times

Figure 3 shows running times for each benchmark and allocator.[2] Even though linear scan spills more than the other allocators, the effect of spilling on execution times is limited. On a register-rich architecture as the SPARC, linear scan has in most cases performance comparable to that of the graph coloring and coalescing allocators. Also, note that different choices of registers might affect the dynamic instruction scheduling performed by the hardware, causing differences in execution times. These differences might even be large enough to outweigh the cost of a spill. This explains why code produced with linear scan sometimes executes faster than code produced by graph coloring, even though linear scan spills.

	Naïve	Linear Scan	Graph Color.	Iterat. Coal.
spillstress	7.8	1.4	1.4	1.5
smith	35.7	6.5	6.4	6.4
life	9.4	4.8	5.0	5.1
decode	31.5	17.9	14.7	14.8
huff	41.9	10.0	10.1	10.0
prettypr	4.0	1.8	1.8	1.7
estone	55.9	32.7	33.5	33.3
eddie	45.1	13.8	13.2	13.2
translate	—	1.9	1.8	—

Fig. 3. Execution times in seconds.

4.4 Spills and one more experiment

Table 2 shows the number of temporaries spilled and the number of instructions after allocation. From it, one can see that even though linear scan spills fewer temporaries than the graph colorer on **spillstress** and **decode** the total number of instructions for graph coloring is lower. This is because the linear scan allocator has a tendency to spill long live intervals with many uses while the graph colorer spills more temporaries but with shorter live intervals and fewer uses.

As expected, the coalescing allocator always generates fewer spilled temporaries. Also, since the coalescing allocator is usually able to coalesce `moves` and handles `multimoves` differently than the graph coloring and linear scan implementations, the resulting number of instructions is smaller for coalescing even when none of the algorithms spills. As mentioned, the naïve allocator spills all

[2] Two allocators can not handle the **translate** benchmark due to memory limitations.

non-precolored temporaries, adding `load` and `store` instructions at each use or define site. The number of instructions should be compared to the numbers in Table 2 to see the increase in size caused by spills introduced by each algorithm.

	Naïve		Linear Scan		Graph Coloring		Coalescing	
	Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
spillstress	395	9461	9	4173	33	3330	4	3126
smith	631	9057	0	2877	0	2877	0	2590
life	711	9668	1	3157	0	3140	0	2867
decode	1044	14790	10	5665	22	4952	4	4542
huff	1544	21101	0	6566	0	6566	0	5982
prettypr	2928	41833	4	13363	0	12106	0	11070
estone	4951	69516	1	22419	0	22374	0	20365
eddie	7392	98543	2	30306	0	30209	0	27657
translate	6321	104612	7	36804	0	30956	—	—

Table 2. Number of spilled temporaries and SPARC instructions after allocation.

HiPE currently runs exclusively on SPARCs. However, an x86 port is underway and possible designs under consideration involve precoloring a number of registers for referring to data structures in the virtual machine, and using the remaining registers (from 6 to 3) for temporaries. Thus, we were curious to see how our register allocators would perform in such a register-poor architecture by counting the number of spills. Besides satisfying our curiosity, this experiment is interesting since to the best of our knowledge the performance of linear scan in such a situation has nowhere been reported.

Our results are as follows: When the number of available registers is low, the coalescing algorithm is the clear winner as far as the number of spilled temporaries is concerned; see Table 3. It manages to minimally spill on three benchmarks, and to spill only relatively few temporaries on the other benchmarks as well. With only 3 available registers, the linear scan register allocator has real problems keeping temporaries in registers and the number of spills becomes quite high compared to coalescing. Even though the number of spills is lower than for the graph colorer, the number of instructions in the resulting code is always higher, suggesting that the choice of spilled temporaries is not a good one.

	Naïve		Linear Scan		Graph Coloring		Coalescing	
	Spills(6)	Spills(3)	Spills(6)	Spills(3)	Spills(6)	Spills(3)	Spills(6)	Spills(3)
spillstress	395	395	17	25	88	109	16	20
smith	631	631	15	39	10	74	1	3
life	711	711	9	28	18	56	1	4
decode	1044	1044	18	29	68	138	12	15
huff	1544	1544	8	40	8	108	0	5
prettypr	2928	2928	44	129	114	472	7	34
estone	4951	4951	35	107	76	224	7	19
eddie	7392	7392	52	174	72	382	4	22
translate	6321	6321	150	395	288	856	—	—

Table 3. Number of spilled temporaries after allocation using 6 or 3 registers.

5 Linear Scan: Impact of some alternatives

Before settling on a default setting, we have experimented with a number of options that one can consider when implementing linear scan. Some of these (e.g., spilling heuristics) are also considered in [9], some are suggested as possible future work (renaming), and some others (various orderings) are of our own invention. Nevertheless, as experimenting with all these options is time-consuming, we hope that our reporting on them will prove helpful to other implementors.

5.1 Impact of instruction ordering

Linear scan relies on a linear approximation of the execution order of the code to determine simultaneously live temporaries. In order to spill as few registers as possible, it is important that this linear approximation introduces as few false interferences as possible. A false interference is when the linearization places a basic block (B2) where a temporary (T) is not live between two blocks (B1, B3) which define and use T. The live interval for T will then include all instructions in B2, resulting in false interferences between T and any temporaries defined in B2; see Fig. 4. If, instead, the linearization places B2 and B3 in the opposite order, there will not be a false interference.

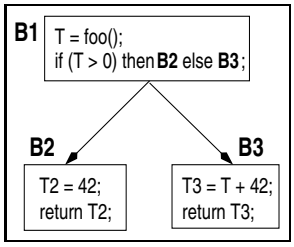
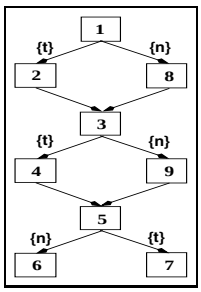


Fig. 4. A possible CFG.

Finding the optimal ordering (i.e., the one with the least number of false interferences) is not feasible. It is therefore important to try to find a general ordering that on average gives the best result. To determine such an ordering, we have applied the linear scan algorithm on six different orderings and counted the number of spills and the number of added instructions on the benchmarks used in Section 4. We will exemplify these orderings (most of them are standard; see e.g., [7]) using the control-flow graph shown on the right where edges are annotated with a static prediction (taken/not-taken).



Depth-first ordering The reverse of the order in which nodes are last visited in a preorder traversal. (1,8,2,3,9,4,5,7,6)

Preorder The start node is first visited, then the children nodes are placed on the top of the stack to visit. (1,2,3,4,5,6,7,9,8)

Postorder The reverse of the depth-first ordering. (6,7,5,4,9,3,2,8,1)

Breadth first ordering The start node is placed first, then its children followed by the grandchildren and so on. (1,2,8,3,4,9,5,6,7)

Prediction The static prediction of branches is used to order the basic blocks in a depth first order. This should correspond to the most executed path being explored first. (1,2,3,4,5,7,8,9,6)

'Random' (or rather an approximation of the source code order.) The blocks are ordered by converting the hash-table of basic blocks into a list; this list is approximately ordered on an increasing basic block numbering, which in turn corresponds to the order the basic blocks were created. (1,2,3,4,5,6,7,8,9)

The style in which a program is written has a big impact on which ordering performs best. Factors such as how nested the code is, or the size of each function come into play. The results therefore, as expected, vary from benchmark to benchmark, but provided the range of benchmarks is large and the (weighted) sum of results is taken, a "winner" can be found. As can be seen in Fig. 5, which shows the total percentage of spilled temporaries and total number of added loads and stores, the depth-first ordering performs best. This is currently used as default.

Ordering	% Spilled temps	Added instrs
Depth-first	0.11	9549
Preorder	0.26	12981
Postorder	0.39	16507
Prediction	1.25	27275
Breadth-first	1.93	35341
'Random'	3.13	41449

Fig. 5. Impact of basic block orderings.

5.2 Impact of performing liveness analysis

In [9], a fast live interval analysis is described that does not use an iterative liveness analysis, but instead extends the intervals of all live temporaries in a strongly connected component (SCC) to include the whole SCC. After presenting the method, the authors conclude that although this method is compilation-wise sometimes faster than normal linear scan, the resulting allocation is usually much worse. We have independently confirmed their findings. In fact, the allocation is sometimes so bad that excessive spilling increases the time it takes to rewrite the code so much that the SCC-based linear scan register allocation becomes slower than linear scan with a full liveness analysis. Looking at compilation times (shown in Table 4 in secs), one can see that linear scan using liveness analysis is faster on more than half of the benchmarks.

	Liveness					**SCC**				
	Comp	RA	RA%	Spills	Exec Time	Comp	RA	RA%	Spills	Exec Time
spillstress	1.3	0.2	16.8	9	1.5	1.7	0.5	27.7	139	1.7
smith	1.7	0.3	18.4	0	6.5	1.7	0.3	16.9	60	9.8
life	1.9	0.4	18.9	1	4.9	1.9	0.4	18.7	171	6.8
decode	4.4	0.6	12.5	10	18.6	4.5	0.7	16.3	139	28.9
huff	6.3	0.8	13.2	0	10.0	5.9	0.7	12.4	96	17.9
prettypr	57.3	1.8	3.2	4	2.0	*83.7*	*19.3*	*23.1*	1951	—
estone	14.2	2.4	16.7	1	33.7	14.7	2.8	19.0	1338	38.0
eddie	22.2	3.5	15.5	2	13.9	22.0	3.7	16.8	808	23.7
translate	257.4	4.9	1.9	7	0.7	*293.0*	*36.4*	*12.4*	1204	—

Table 4. Impacts of using liveness analysis or an SCC-based approximation of liveness.

In **prettypr**, without liveness analysis, one function spills more than 1024 registers. However, the current implementation of linear scan can spill at most 1024 temporaries per function.[3] One function in the **translate** benchmark also reaches this limit if renaming (which introduces more register names; see Section 5.3) is used. The violation of this limitation is discovered by the assemble-

[3] This limit is arbitrary and could be removed.

link stage in the compiler which is the last stage. This means that we still can get compilation times that are quite accurate but no execution times for these benchmarks. This is why some numbers are shown using a different font, and why some execution times are missing from Table 4. As can be seen from a comparison of the total number of spills and execution times for linear scan with a full liveness analysis and with an SCC-based approximation of liveness, the code generated when using liveness analysis performs better on all benchmarks.

5.3 Impact of renaming

The HiPE compiler translates bytecode for a virtual machine into an intermediate language with unlimited registers. This translation process introduces false dependencies since some stack positions in the virtual machine are reused at different points in the program for different values. These false dependencies have a negative impact on the performance of some optimizations in the compiler. This can be remedied by a renaming pass (see e.g., [7]), which gives a unique variable name to each value. The renaming pass results in some speedup in the generated code, both for graph coloring and for linear scan, as can be seen in Fig. 6. Since all optimizations after renaming become faster and more accurate, we recommend using this pass even though compilation may be slower.

	Linear Scan		**Graph Color**	
	w VR	w/o VR	w VR	w/o VR
spillstress	1.5	1.7	1.4	1.5
smith	6.5	6.8	6.4	7.0
life	4.9	5.0	5.0	5.1
decode	18.6	22.0	14.7	15.6
huff	10.0	11.9	10.2	11.6
prettypr	2.0	2.5	1.9	1.8
estone	33.7	35.1	33.8	34.6
eddie	13.9	14.2	14.2	13.7

Fig. 6. Execution times.

5.4 Impact of spilling heuristics

We also experimented with a spilling heuristic based on usage counts instead of interval length. Since information about the length of intervals is needed by the linear scan algorithm anyway (in order to free registers when an interval ends) this information can be used "for free" to guide spilling. The usage count heuristic is slightly more complicated to implement since it needs some extra information: the usage count of each temporary. There is also a compile-time cost for finding the temporary with the least use. As Fig. 7 shows, the usage count heuristic spills more (as expected) but spills temporaries which are not used much, so

	Spills (Instructions)				Execution Time	
	Interval length		**Usage count**		**Interval length**	**Usage count**
spillstress	9	(4173)	38	(3569)	1.5	1.3
smith	0	(2877)	0	(2877)	6.5	6.4
life	1	(3157)	6	(3176)	4.9	5.1
decode	10	(5665)	51	(5236)	18.6	19.6
huff	0	(6566)	0	(6566)	10.0	10.0
prettypr	4	(13363)	33	(12323)	2.0	1.9
estone	1	(22419)	3	(22389)	33.7	33.8
eddie	2	(30306)	20	(30320)	13.9	13.6
translate	7	(36804)	189	(32334)	1.9	1.9

Fig. 7. Impact of spilling heuristics.

the size of the resulting code is sometimes considerably smaller (see e.g., **translate**). However, the performance of the generated code using the two heuristics is comparable. We thus do not recommend the use of usage counts.

6 Concluding Remarks

We have experimented with linear scan register allocation and reported on its performance in the context of a just-in-time native code compiler for a virtual-machine-based implementation of the concurrent functional language Erlang. Besides describing our experiences in using linear scan in a significantly different context than those in which it has so far been employed, we thoroughly investigated how various options to the basic algorithm affect compilation time and the quality of the resulting code. In many cases, we have independently confirmed the results of [9]; in others, we provide a deeper look or improve on those results in the hope that this will prove useful to other functional language implementors.

Stated briefly, our experience is that in a register-rich environment, such as the SPARC (or the upcoming IA-64), linear scan is a very respectable register allocator: it is significantly faster than algorithms based on graph coloring, resulting in code that is almost as efficient. When compilation time is a concern, or at low optimization levels, it should be used.[4] On the other hand, we do not recommend using linear scan in register-poor architectures; the coalescing allocator is probably a better candidate.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Prog. Lang. Syst.*, 16(3):428–455, May 1994.
3. G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 98–105. June 1982.
4. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.
5. L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, May 1996.
6. E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, 2000.
7. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.
8. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Trans. Prog. Lang. Syst.*, 21(2):324–369, Mar. 1999.
9. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.

[4] On the SPARC, HiPE currently uses linear scan as default register allocator in optimization levels up to `-O2`; graph coloring is used in `-O3`.