Linear Scan Register Allocation in a High-Performance Erlang Compiler

Erik Johansson and Konstantinos Sagonas

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

Abstract. In the context of an optimizing native code compiler for the concurrent functional programming language Erlang, we experiment with various register allocation schemes focusing on the recently proposed *linear scan register allocator*. We describe its implementation and extensively report on its behaviour both on register-rich and on register-poor architectures. We also investigate how different options to the basic algorithm and to the compilation process as a whole affect compilation time and quality of the produced code. Overall, the linear scan register allocator is a good choice on register-rich architectures; when compilation time is a concern, it can also be a viable option on register-poor architectures.

1 The story

During the last years, we have been developing HiPE, a High-Performance native code compiler for the concurrent functional programming language Erlang [2]. Erlang has been designed to address the needs of large-scale soft real-time control applications and is successfully being used in products of the telecommunications industry. The HiPE system significantly improves the performance characteristics of Erlang applications by allowing selective, user-controlled, "just-in-time" compilation of bytecode-compiled Erlang functions to native code. As reported in [7], 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 focusing on dynamic compilation—the compiler is usually used either interactively or through `make`—compilation times are a significant concern in an interactive development environment which 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 to build

[1] Since October 2001, HiPE is part of Ericsson's Open Source Erlang release 8 (available from www.erlang.org). See also: www.csd.uu.se/projects/hipe.

the interference graph is often prohibitive. So, we were looking for a more well-behaved register allocator.

In doing so, we found out about the recently proposed *linear scan register allocator* [11]. The method is quite simple and extremely intriguing: with relatively little machinery—and more importantly by avoiding the quadratic cost of heuristic coloring—almost equally efficient code is generated. At first, we were skeptical whether the results of [11] would carry over to HiPE: One reason is because programs written in Erlang—or a functional language in general—may not benefit to the same extent from techniques used for compiling C or 'C [10] programs. More importantly because, even on a register-rich architecture such as the SPARC, HiPE reserves 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 [10, 11]. Finally, because we had no idea how the results would look on a register-poor architecture such as IA-32; to our knowledge nobody has ever experimented with linear scan in this setting. Despite our skepticism, we were willing to give it a try. In about two weeks, a straightforward implementation based on the description in [11] was ready, but this was the easy part. The results were encouraging, but we were curious to experiment with various options that [11] also explores, sometimes leaves open, or suggests as future research. We did so and report our findings in this paper.

This paper, besides documenting our implementation (Section 3) and extensively reporting on its performance characteristics (Section 4), describes our experience in using linear scan register allocation in a context which is significantly different from those previously tried. In particular, we are the first ones to investigate the effectiveness of linear scan on the IA-32. Finally, we present results from trying different options to linear scan and report on their effectiveness and on the trade-offs between the speed of register allocation and the quality of the resulting code (Section 5). We thus believe that this paper and our experiences should prove useful not only to other declarative language implementors, but also to all compiler writers considering linear scan register allocation or being involved in a project where compilation time is a concern.

We begin with a description of various schemes for register allocation, focusing on linear scan.

2 Global register allocation

2.1 Graph coloring register allocation & iterated register coalescing

The idea behind coloring-based schemes is to formulate register allocation as a graph coloring problem [4], by representing liveness information with an *interference graph*. Nodes in the graph represent variables that are to 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.* [4, 5] iteratively builds an interference graph and heuristically attempts to color it. If the graph is not colorable, nodes are deleted from the graph, the corresponding temporaries are spilled to memory, and the process is repeated until the graph becomes colorable. Since Chaitin's paper, many variations or improvements to the basic scheme have emerged. In particular, we note the method by Briggs *et al.* [3] which is the one we are using in the HiPE compiler. Compared to [3], iterated register coalescing [6] is a more aggressive coloring-based technique aiming at eliminating many redundant `move` instructions: 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.

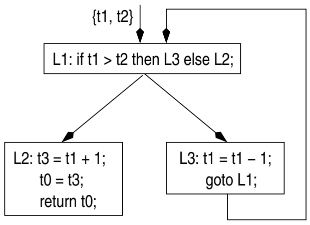
In practice, coloring-based allocation schemes usually produce good code. However, the cost of register allocation is dominated by the construction of the interference graph, which can take time (and space) quadratic in the number of nodes. When compilation time is a concern, as in just-in-time compilers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

2.2 Linear scan register allocation

The linear scan allocation algorithm [11] 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.[2] 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.

Let us look at each step, using Figure 1 as our 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.

[2] Schemes such as [12] for utilizing holes in live ranges will not be considered here.

Ordering instructions linearly As long as the calculation of live intervals is correct, an arbitrary linear ordering of the instructions can be chosen. In our example, the control-flow graph of Figure 1(a) can be linearized in many ways; Figures 1(b) and 1(c) show two possible orderings. 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 in [11] 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.

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 its last use. 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:

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

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

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

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 performing the somewhat costly liveness analysis that give correct—but sub-optimal—live intervals. One approach is to use strongly connected components in the control-flow graph; see [11].

Allocation of registers to intervals When all intervals are computed, the resulting data structure (Intervals) gets ordered in increasing start-points so as to make the subsequent allocation scan efficient. For our first linearization this would result in:

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

Allocation is then done by keeping a set of allocatable free physical registers (FreeRegs), a set of already allocated temporaries (Allocated), and a list containing a mapping of active intervals to registers (Active). The active intervals are ordered on increasing end-points while traversing the start-point-ordered list of intervals. For each interval in Intervals, that is, for each temporary $\mathtt{t}_i$ (with interval $[start_i, end_i]$) do:

- For each interval j in Active which ends before or at the current interval (i.e., $end_j \leq start_i$), free the corresponding register and move the mapping to Allocated.
- If there is a free register, r, in FreeRegs, remove r from FreeRegs, add $\mathtt{t}_i \mapsto r$ with the interval $\mathtt{t}_i : [start_i, end_i]$ to Active (sorted on end).

  If, on the other hand, there are no free registers and the end-point end_k of the first temporary $\mathtt{t}_k$ in Active is further away than the current (i.e., $end_k > end_i$), then spill $\mathtt{t}_k$, otherwise spill $\mathtt{t}_i$. By choosing the temporary whose live interval ends last, the number of spilled temporaries is hopefully kept low. (Another way to choose the spilled temporary 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. On RISC architectures, 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. On architectures that have fancy addressing modes, such as the IA-32, the location of the spilled temporary can often be used directly in the instruction, in which case no extra instruction for loading or storing the temporary is needed.

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 [3], is well-tested, and uses efficient data structures. Despite being *pessimistic* in its spilling strategy (see [9]), it seldom needs to spill on the SPARC.

The iterated register coalescing allocator on the other hand, is *optimistic* in its spilling (similar to the strategy described in [9]), and was implemented having as main goal to obtain a register allocator that follows the algorithm described in [6] so as to establish an approximate lower bound on the number of spills for the benchmarks.

3.2 Linear scan register allocator

As mentioned, the linear scan register allocator was first implemented based on the description in [11]. Afterwards, we experimented with various options and tuned the first implementation considerably. In this section, we only describe the chosen default implementation (i.e., the one using the options that seem to work best in most cases) by looking at how each step of the algorithm is implemented in HiPE. In Section 5 we will improve on [11] by describing and quantifying the impact of the alternatives that 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 basic blocks 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 not included in both the live-in and the live-out set, 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 A list of {`Temporary`,`StartPoint`,`EndPoint`} triples. This is a sorted (on `StartPoint`) representation of the interval structure calculated in the previous step.

FreeRegs Allocatable physical registers (`PhysReg`) which are currently not allocated to a temporary.

Active A list of {`Temporary`,`PhysReg`,`EndPoint`} triples 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 `EndPoint`. 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. Tuples have forms {`Temporary`,{`reg`,`PhysReg`}} or {`Temporary`,{`spill`,`Location`}}.

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

1. Move the information about each interval in the Active list that ends before[3] `StartPoint` 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:

[3] Here one could use "before or at" instead of "before" in order to increase the likelihood that two registers (`t1`,`t2`) in a move: `t1 = t2` are assigned to the same register. Since HiPE has a `multimove` instruction (sometimes called parallel move), this is not straightforward since several live intervals might begin at the same program point.

- If there is a physical register in the FreeRegs list then tentatively 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, the Allocation structure is turned into a tuple that can be used for $O(1)$ mapping from a temporary to its allocated physical register (or spill position). Then all instructions are rewritten using this mapping.

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. For example, on the SPARC, 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 added `load` and `store` instructions is equal to the number of uses and defines in the program. Currently each spill requires three instructions on the SPARC: two to set up the base address to the spill area[4] 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.

3.4 The SPARC back-end

To keep garbage collection times low, all dead objects should look dead to the garbage collector as soon as possible, preferably from the moment that they die. In order to achieve this, it is important to not let dead temporaries reside in the root set of a process's memory, for example on the stack. To ensure this, on the SPARC back-end we have chosen to only save live temporaries on the stack during a function call. We also use variable-sized stack frames and place the live temporaries so that there are no holes on the stack. This way, the garbage collector does not need any additional information about the stack other than a pointer to the top and a pointer to the bottom.

[4] A large constant takes two instructions to create on the SPARC.

This scheme has some drawbacks, one being that all live temporaries have to be written to and read from the stack at function calls. Also, since the frame is trimmed so that only live values reside on the stack, another drawback is that there is no easy way to spill to the stack: the relative offset from the stack pointer to the spill position needs to be calculated *for each use and define* of spilled temporaries. However, since each temporary is saved on the stack at each function call, we know that any spilled values only need to survive till the next function call. This in turn means that we can store all spilled temporaries for a function at a fixed memory area: Even though there might be several processes executing the same code and calling the code recursively, only one of them will use the spill area at a time; all other processes will save their temporaries on their stack.[5]

HiPE is a just-in-time native code compiler extension to a virtual-machine-based runtime system. This has influenced the compiler in several ways: There are for example several special data structures that are part of the virtual machine. Since we know that they are important for the execution of Erlang programs, as they are heavily used, we would like to keep them in registers. On a register-rich architecture such as the SPARC, we have chosen to cache 12 data structures in registers (the stack pointer, stack limit, heap pointer, heap limit, a pointer to the process control block, the reduction counter, the return address, and the first five arguments of Erlang functions), and we have reserved another 4 registers (Zero (%g0), C's SP (%o6) and FP (%i6), and the SPARC API reserved register (%g7)) leaving 16 registers for use by the register allocators. (The allocators can also use the 5 argument registers when they are unused.) As mentioned in the introduction, in this respect, compilation of Erlang programs differs from the contexts to which linear scan has been previously applied [10, 12].

3.5 The IA-32 back-end

The IA-32 back-end was developed after the SPARC one. In its implementation, we decided to try out some new design choices. The most important difference with the SPARC back-end is the use of stack descriptors (*stack maps*), enabling spilling on the stack, while having stack frames of fixed size for each function.

Another difference is in the handling of temporaries that are live across function calls. On the SPARC, a pre-pass ensures that all such temporaries are saved on the stack before the call and are restored afterwards. The register allocators can then pretend that all registers are callee-saves. On the IA-32, the allocators assume that the `call` instruction defines all physical registers, thus preventing any temporaries that are live across a function call from being allocated to any physical register. This means that all these temporaries will be allocated on (spilled to) the stack.[6] The two approaches differ in when a temporary that lives

[5] This approach assumes that the runtime system only executes one Erlang process at a time (i.e., that it is single-threaded), which is currently true for the underlying Erlang/OTP system and thus naturally also for HiPE.

[6] For the IA-32 back-end, we had to adjust the linear scan allocator slightly to handle the case when a temporary is defined several times but never used; such is the case

across function calls needs to be read from or written to memory. In the worst case, on the SPARC, a read is needed after each call; on the IA-32, a read is needed at each use. On the SPARC, a write is needed at each call; on the IA-32, a write is needed at each definition. In a pure functional language such as Erlang we suspect that the number of uses plus the number of defines is less than two times the number of calls a temporary is live over. If so, the approach used by the IA-32 back-end is a winner. It remains future work to verify this.

We pre-allocate much fewer, only three, registers on the IA-32: The stack pointer is allocated to `%esp` and the process pointer to `%ebp`. At function calls, all arguments are passed on the stack and the return value is passed in `%eax`. The register allocator will not try to allocate the arguments in registers but keep them in memory (on the stack). The return value on the other hand is always moved to a new temporary directly after the return of the function. Hence, we can use the `%eax` register as a general purpose register, leaving six registers for the register allocators to play with.

Most instructions of the IA-32 ISA can take a memory location (e.g. register+immediate offset) as one of the operands (or the destination). By using these addressing modes, we can in many cases use spilled temporaries directly, without having to first load them from memory to a register.

Prior to register allocation, the code is in a pseudo-IA-32 intermediate form in which arbitrary operands are allowed in each instruction. After register allocation, a post-pass ensures that each instruction complies with the real IA-32 ISA, e.g. that no binary operation uses more than one memory operand. This is done by rewriting the code to use `load`s and `store`s to new temporaries. If any instruction has to be rewritten in this way, then the register allocator is called again with the additional constraint than none of the newly introduced temporaries may be spilled.

The naïve allocator is also using the memory operands of the IA-32. Thus, despite the fact that each temporary is considered spilled by the allocator, an additional `load` or `store` instruction is not always needed. (If `load`s or `store`s are inserted they use a pre-allocated physical register instead of introducing a new temporary.)

4 Performance results

The register allocator implementations described in the previous section have been evaluated by compiling and running a set of benchmarks. All four register allocators are integrated in the HiPE system (version 0.98) 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

of physical registers at function calls. For these temporaries we effectively perform *live-range splitting* and end up with temporaries that have several one-instruction intervals. The alternative would have been that the live intervals of all physical registers would range over most of the function, rendering allocation of other temporaries, and thus use of linear scan, impossible.

representation of SPARC or pseudo-IA-32 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. [8]. The mapping of temporaries to the set of physical registers creates false dependencies, since different temporaries are mapped to the same physical register. This means that register allocation may possibly interfere with optimization passes that come after it; instruction scheduling in particular. In our experimental evaluation we have thus turned off all instruction scheduling. Still, the allocation might affect the scheduling done in hardware.

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

The two platforms we used are: A Linux Pentium-II 333 MHz, 128 MB memory Dell Inspiron 7000 laptop, and a 2-processor Sun Enterprise 3000, 1.2 GB main memory running SunOS 5.7. Each processor is a 248 MHz UltraSPARC-II. (However, the HiPE system uses only one processor.)

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 the largest module of the HiPE compiler which is quite troublesome for some register allocators.

spillstress A synthetic benchmark consisting of a recursive function with several continously live variables; its only 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.

estone Measures the number of Estones that an Erlang system can produce. This is an Erlang/OTP benchmark that aims at stressing all parts of an Erlang implementation.

beam2icode The part of the HiPE compiler that translates BEAM bytecode into intermediate code. The program contains a very big function handling different combinations of instructions. To get measurable execution times, we run this benchmark 10 times.

eddie An Erlang application implementing an HTTP parser which handles http-get requests.

		SPARC			IA-32		
Benchmark	Lines	Insts	BasBlks	Temps	Insts	BasBlks	Temps
spillstress	94	1710	338	412	644	188	190
smith	93	2666	764	1004	1637	410	524
life	189	3018	976	1318	1855	516	603
decode	381	3692	1028	1114	2437	612	600
huff	177	4780	1350	1484	2938	768	749
estone	1134	18284	5604	6574	11768	3142	3372
beam2icode	1470	27452	7780	6452	18638	4736	4161
eddie	2233	45324	13155	16210	28625	7250	8242

Table 1. Sizes of benchmark programs.

Sizes of benchmark programs (lines of source code, the number of SPARC or IA-32 instructions before register allocation, and the total number of basic blocks and temporaries) are shown in Table 1. We note in passing that, on the SPARC, no allocator spills on many other standard Erlang benchmarks.

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 2 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 (see e.g. **estone** on the SPARC). This is due to the large number of extra instructions which are added to the code due to spilling.

Also, as one can see from Table 2, the complexity of the graph coloring and the coalescing register allocator is not directly dependent on what one could naively consider as the 'size' of the program. Instead the complexity depends on the number of edges in the interference graph, which is for example high for the **spillstress** benchmark on the SPARC. On the other hand, the linear scan allocator is not affected much by the number of simultaneously live temporaries; the allocation time is instead dominated by the time to traverse the code.

The compilation times of **beam2icode** stick out since this program contains one large function with many simultaneously live temporaries. This becomes troublesome either when many iterations are needed to avoid spilling (which is what happens with iterated register coalescing on the SPARC), or when the number of available registers is low, the produced allocation does not respect the constraints imposed by the ISA, and small corrections to the allocation are needed (such is the case on the IA-32). On the other hand, **estone** and **eddie** consist of a large number of small functions that do not exhibit this behaviour.

Compilation-time-wise, linear scan performs very well: compared to graph coloring, the time for register allocation is significantly reduced, and pathological cases such as **beam2icode** are avoided. In fact, compilation with linear scan is

	Naïve			Linear Scan			Graph Coloring			Iterated Coalesc.		
	Comp	RA	RA%	Comp	RA	RA%	Comp	RA	RA%	Comp	RA	RA%
SPARC												
spillstress	2.7	0.2	7.1	2.5	0.3	12.4	3.4	1.3	37.2	7.0	4.8	68.5
smith	4.4	0.3	7.0	3.8	0.5	14.0	4.1	0.9	21.0	6.1	3.0	49.8
life	4.8	0.3	6.7	4.3	0.7	15.2	4.7	0.9	20.2	6.0	2.2	37.3
decode	7.7	0.5	6.6	7.0	0.8	11.6	7.8	1.7	21.6	11.8	5.8	49.6
huff	7.3	0.6	8.2	6.2	1.0	16.6	6.8	1.5	22.3	12.6	7.4	58.9
estone	35.5	2.4	6.8	23.8	3.2	13.5	25.2	4.9	19.5	34.3	14.8	43.3
beam2icode	84.1	6.3	7.4	56.7	6.6	11.6	57.8	10.6	18.3	2769.2	2721.5	98.3
eddie	70.0	5.8	8.2	66.3	9.0	13.6	76.4	20.7	27.1	110.6	54.0	48.8
IA-32												
spillstress	1.3	0.03	2.3	1.4	0.24	17.1	1.5	0.35	23.3	1.8	0.73	40.6
smith	1.9	0.04	2.1	2.3	0.62	27.0	2.8	1.14	40.7	3.6	1.86	51.7
life	2.1	0.03	1.4	2.7	0.74	27.4	3.1	1.06	34.2	3.8	1.79	47.1
decode	2.9	0.02	0.7	3.7	1.07	28.9	5.3	2.56	48.3	8.5	5.82	68.5
huff	3.2	0.01	0.3	4.0	1.15	28.7	5.1	2.22	43.5	6.9	3.97	57.5
estone	10.6	0.15	1.4	14.1	4.15	29.4	15.6	5.74	36.8	20.6	10.97	53.3
beam2icode	40.1	0.10	0.2	45.8	21.03	46.0	142.9	117.54	82.3	191.9	167.58	87.3
eddie	34.8	0.21	0.6	44.0	10.20	23.2	51.6	17.92	34.7	73.5	40.25	54.8

Table 2. For each allocator, columns show total compilation time in seconds (Comp), register allocation time (RA), and percentage of total compilation time spent on register allocation (RA%).

sometimes even faster than the naïve algorithm. This is due to the time needed for rewrite of the code with the allocation. Due to excessive spilling this code is larger for naïve than it is for linear scan; see also Table 4.

4.3 Execution times

The execution times, in seconds, for each benchmark and allocator are presented in Table 3.

Even though on the SPARC the cost for spilling is higher than necessary (three instructions for each use or define instead of one; see Section 3.4), and linear scan spills more than the other allocators, the effect of spilling on execution times is limited. On a register-rich architecture such as the SPARC, linear scan has in most cases performance comparable to that of the graph coloring and iterated coalescing allocators. Linear scan also performs well on the IA-32; this is partly due to the different calling convention (passing arguments on the stack), and also partly due to the L1 cache being accessed almost as fast as the registers on the Pentium.

Also, note that different register assignments might affect the dynamic instruction scheduling done by the hardware, causing small 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 and graph coloring does not (see e.g. **estone** on SPARC).

	SPARC				*IA-32*			
	Naïve	**Linear Scan**	**Graph Color.**	**Iterated Coalesc.**	**Naïve**	**Linear Scan**	**Graph Color.**	**Iterated Coalesc.**
spillstress	26.1	8.1	8.1	7.7	12.0	9.4	9.4	9.4
smith	31.9	7.1	7.0	6.9	9.0	5.9	6.7	6.0
life	29.1	21.6	21.7	21.6	15.4	14.6	14.4	14.3
decode	21.2	9.3	9.3	9.1	11.8	6.2	6.8	6.3
huff	27.2	9.1	8.9	8.9	11.5	9.5	9.6	10.1
estone	37.2	27.7	27.6	27.8	22.3	19.9	19.6	19.8
beam2icode	13.0	12.2	12.1	12.1	15.4	12.3	13.8	12.1
eddie	22.4	11.2	11.4	11.6	13.6	12.2	13.1	11.8

Table 3. Execution times in seconds.

4.4 Spills on SPARC

Table 4 shows the number of temporaries spilled and the number of instructions after allocation. From these numbers, one can see that even though linear scan spills fewer temporaries than the graph colorer on **decode** and **eddie**, 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 in number but with shorter lives and fewer uses.

As expected, the iterated 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 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 1 to see the increase in size caused by spills introduced by each algorithm.

4.5 Spills on IA-32

With the IA-32 port of HiPE ready, we were curious to see how our register allocators perform in this back-end by also looking at the number of spills.

	Naïve		**Linear Scan**		**Graph Color**		**Iter. Coalesc.**	
	Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
spillstress	116	4701	8	2335	4	1981	3	1853
smith	412	6971	0	2666	0	2666	0	2421
life	393	6840	0	3018	0	3018	0	2782
decode	522	9299	27	4360	38	4037	17	3628
huff	633	12061	4	4859	0	4780	0	4334
estone	2393	43430	2	18362	0	18284	0	16650
beam2icode	3936	69140	13	29405	0	27452	0	25198
eddie	6849	113676	31	47688	144	46870	20	43998

Table 4. Number of spilled temporaries and SPARC instructions after allocation.

Besides satisfying our curiosity, this experiment is interesting since to the best of our knowledge the performance of linear scan on a register-poor architecture such as the IA-32 has nowhere been reported.

Table 5 reports, for each benchmark, the number of temporaries that are placed on the stack by the calling convention, and the number of additional spills and instructions after allocation.

		Naïve		Linear Scan		Graph Color		Iter. Coalesc.	
	on stack	Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
spillstress	42	60	721	0	701	9	701	0	701
smith	49	299	2056	11	1879	23	1914	0	1868
life	63	265	2322	16	2229	13	2220	3	2205
decode	95	329	3079	8	2949	5	2971	0	2935
huff	125	371	3716	17	3526	25	3521	0	3458
estone	281	1848	14360	72	13493	43	13501	5	13416
beam2icode	801	2612	24534	44	23478	131	23804	0	23345
eddie	821	4638	38387	102	36158	108	36203	0	35871

Table 5. Number of spilled temporaries and IA-32 instructions after allocation.

Our results are as follows: When the number of available registers is low, the iterated coalescing algorithm is the clear winner as far as its ability to place temporaries in registers is concerned. It manages to minimally spill on this benchmark set. In comparison to graph coloring, this is partly due to the fact that the coalescing allocator is *optimistic* in its spilling strategy. With only few available registers, the linear scan register allocator has trouble keeping temporaries in registers and the number of spills is high compared to coalescing. Compared to the graph colorer, sometimes even though the number of spills is lower, the number of instructions in the resulting code is higher, suggesting that the choice of spilled temporaries is not a good one.

We stress that, due to the different calling conventions used by the SPARC and IA-32 back-ends, the number of spills in Tables 4 and 5 are incomparable.

5 A deeper look on linear scan: Impact of some alternatives

As mentioned, before settling on a default setting, we have experimented with a number of options that one can consider when implementing linear scan. One of these (spilling heuristics) is also considered in [11], another is 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. All experiments of this section are conducted on the SPARC.

5.1 Impact of instruction ordering

The linear scan algorithm 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 approximation introduces as few false interferences as possible. An interference is false 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 Figure 2(a). If, instead, the linearization places B2 and B3 in the opposite order, there will not be a false interference.

Finding the optimal ordering (i.e., the one with the least number of false interferences) is not feasible; this would seriously increase the complexity of the algorithm. 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 five different orderings and counted the number of spills and the number of added instructions on our benchmarks.

We will exemplify the following orderings which we have tried (most of them are standard; see e.g. [1, 8]) using the control-flow graph shown in Figure 2(b) 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)

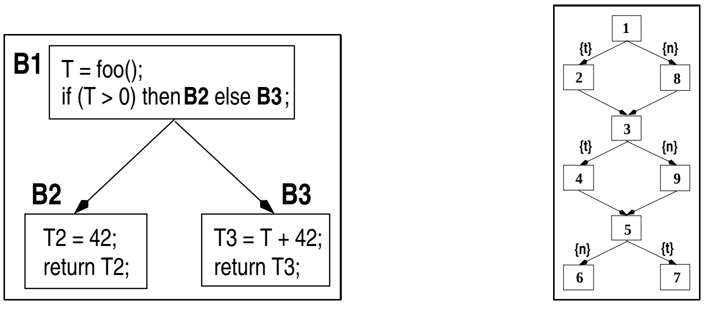
Postorder The reverse of the depth-first ordering. (6,7,5,4,9,3,2,8,1)

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)

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)

'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



(a) Basic block ordering with false interferences.

(b) A CFG

Fig. 2. Control-flow graphs used to illustrate effects of orderings.

Ordering	% Spilled temporaries	Added instructions
Depth-first ordering	0.24	7259
Postorder	0.28	7796
Prediction	6.27	18396
Breadth-first ordering	2.11	21287
'Random'	9.07	26484

Table 6. Impact of using different basic block orderings.

benchmark, but provided the range of benchmarks is large and the (weighted) sums of results is taken, a "winner" can be found. As can be seen in Table 6, which shows the total percentage of spilled temporaries and the total number of added `loads` and `stores`, the depth-first ordering performs best. In HiPE, we are currently using it as the default.

5.2 Impact of performing liveness analysis

In [11], a fast live interval analysis is described that does not use an iterative liveness analysis. Instead, it 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 compilation using linear scan based on this method is 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 allocator becomes slower than linear scan with a full liveness analysis. In our benchmark set, even compilation-time-wise linear scan with liveness analysis is faster on more than half of the benchmarks. Execution-time-wise, performing liveness analysis pays off.

5.3 Impact of renaming

The HiPE compiler translates bytecode for a register-based virtual machine into an intermediate language with unlimited temporaries. This translation process introduces false dependencies since some registers of the virtual machine will be 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 either by SSA conversion or by a renaming pass which gives a unique variable name to each value; see e.g. [8]. The effects of the renaming on execution times can be seen in Table 7; the difference in the number of spills w.r.t. the spills reported in Table 4 is shown within parentheses. Since all optimizations after the renaming become faster and more accurate, we recommend the use of this pass.

	Linear Scan			Graph Coloring			Iter. Coalesing		
	w VR	w/o VR		w VR	w/o VR		w VR	w/o VR	
spillstress	8.1	8.0		8.1	7.9	(-2)	7.7	8.0	
smith	7.1	7.0		7.0	7.0		6.9	6.9	
life	21.6	21.6		21.7	21.7		21.6	21.8	
decode	9.3	10.5	(-4)	9.3	9.3	(-16)	9.1	9.8	(+1)
huff	9.1	8.9	(-3)	8.9	8.4		8.9	8.9	
estone	27.7	27.8	(+1)	27.6	27.8		27.8	28.0	
beam2icode	12.2	12.2	(+8)	12.1	12.3		12.1	12.3	
eddie	11.2	11.4	(+16)	11.4	11.6	(-18)	11.6	11.8	

Table 7. Execution times in secs with (w) and without (w/o) variable renaming (VR).

5.4 Impact of spilling heuristics

We have also experimented with the use of 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 cost in finding the temporary with the least use. As Table 8 shows, the usage count heuristic spills more (as expected) but spills temporaries which are not used much, so the size of the resulting code is sometimes smaller; see e.g. **decode** or **estone**. However, looking at the performance of the generated code, a clear winner between the two heuristics cannot be found. We thus do not recommend the use of usage counts.

	Spills (Instructions)		Execution Time	
	Interval length	*Usage count*	*Interval length*	*Usage count*
spillstress	8 (2335)	27 (2298)	8.1	8.5
smith	0 (2666)	0 (2666)	7.1	7.1
life	0 (3018)	0 (3018)	21.6	21.0
decode	27 (4360)	78 (4336)	9.3	10.8
huff	4 (4859)	52 (5077)	9.1	9.7
estone	2 (18362)	9 (18333)	27.7	27.2
beam2icode	13 (29405)	490 (31642)	12.2	12.5
eddie	31 (47688)	256 (47698)	11.2	11.9

Table 8. Impact of spilling heuristics.

6 Concluding remarks

We have experimented with register allocation, focusing on linear scan, and reported on its performance in the context of a just-in-time native code compiler for a virtual-machine-based implementation of a concurrent functional language. Besides describing our experiences in using linear scan in a significantly different context than those in which it has so far been employed, and reporting on its performance on the IA-32, 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 results of [11]; in many others, we provide a deeper look or improve on those results in the hope that this will prove useful to other declarative programming 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.[7] Disregarding compilation-time concerns, on register-poor architectures, an optimistic iterated coalescing register allocator (which can eliminate most register-register `moves`) is a better approach to obtaining high-performance.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 1996.
3. 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.
4. G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 98–105. ACM Press, June 1982.
5. 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, Jan. 1981.
6. L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, May 1996.
7. 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, Sept. 2000.
8. S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufman Publishers, San Fransisco, CA, 1997.
9. J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques*, pages 196–204. IEEE Press, Oct. 1998.
10. 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.
11. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.
12. O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151. ACM Press, June 1998.

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