SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; (in press) (DOI: 10.1002/spe.533)

Experimental evaluation and improvements to linear scan register allocation

Konstantinos Sagonas[1,*,†] and Erik Stenman[2]

[1]*Computing Science, Department of Information Technology, Uppsala University, Sweden*
[2]*School of Computer and Communication Sciences, Swiss Federal Institute of Technology, Lausanne, Switzerland*

SUMMARY

We report our experience from implementing and experimentally evaluating the performance of various register allocation schemes, focusing on the recently proposed *linear scan register allocator*. In particular, we describe in detail our implementation of linear scan and report on its behavior both on register-rich and on register-poor computer architectures. We also extensively investigate how different options to the basic algorithm and to the compilation process as a whole affect compilation times and quality of the produced code.

In a nutshell, our experience is that a well-tuned linear scan register allocator is a good choice on register-rich architectures. It performs competitively with graph coloring based allocation schemes and results in significantly lower compilation times. When compilation time is a concern, such as in just-in-time compilers, it can also be a viable option on register-poor architectures. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: global register allocation; just-in-time compilation; code generation; Erlang

1. INTRODUCTION

During the last few 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 being used in products of the telecommunications industry. The HiPE system significantly improves the performance

*Correspondence to: Konstantinos Sagonas, Department of Information Technology, Uppsala University, P.O. Box 337, SE-75105, Uppsala, Sweden.
†E-mail: kostis@it.uu.se

Received 3 July 2002
Revised 25 November 2002
Accepted 23 February 2003

characteristics of Erlang applications by allowing selective, user-controlled, 'just-in-time' compilation of bytecode-compiled Erlang functions to native code. As reported in [2], HiPE currently outperforms all other implementations of Erlang[‡].

Achieving high performance does not come without a price: when compiling to native code, compilation times are higher than compilation to virtual machine code. Even though HiPE is currently not focusing exclusively 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 the register allocator, which was based on a variant of graph coloring, was often a bottleneck being responsible for 30–40% of the total compilation time. (Note that it is typical for a global register allocator to be a compiler bottleneck.)

In looking for a more well-behaved register allocator, we found out about the recently proposed *linear scan register allocator* [3]. The method is quite simple and extremely intriguing: with relatively little machinery—and more importantly by avoiding the typically quadratic cost of heuristic coloring—almost equally efficient code is generated. At first, we were skeptical as to whether the results of [3] 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 [4] programs. More importantly because, even on a register-rich computer 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 [3,4]. Finally, because we had no idea how the results would look on a register-poor architecture such as IA-32; to our knowledge there is no published description of the performance of 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 [3] was ready, but this was the easy part. The results were encouraging, but we were curious to experiment with various options that [3] also explores, sometimes leaves open, or suggests as future research. We did so and report our findings in this article.

This article, 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 to investigate the effectiveness of linear scan on the IA-32. Furthermore, 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). In particular, we extensively investigate the impact of conversion to a static single assignment (SSA) form [5] to compilation time and quality of the resulting code. We thus believe that this article and our experiences should prove useful to all programming language implementors who are considering linear scan register allocation or are involved in a project where compilation time is a concern.

We begin with a description of different global register allocation algorithms with emphasis on the relatively new linear scan register allocation algorithm.

[‡]HiPE is incorporated in Ericsson's Erlang/OTP system (available also as open-source from www.erlang.org). See also: www.it.uu.se/research/projects/hipe.

2. GLOBAL REGISTER ALLOCATION

Register allocation aims at finding a mapping of source program or compiler generated variables (henceforth referred to as *temporaries*) to a limited set of physical machine registers. Local register allocation algorithms restrict their attention to the set of temporaries within a single *basic block*. In this case, efficient algorithms for optimal register allocation exist; see e.g. [6,7]. When aiming to find such an allocation for temporaries whose lifetimes span across basic block boundaries (e.g., for all temporaries of a single function), the process is known as *global register allocation*. In this case, control-flow enters the picture and obtaining an optimal mapping becomes an NP-complete problem; see [8,9]. Since the early 1980s, global register allocation has been studied extensively in the literature and different approximations to the optimal allocation using heuristics and methods from related NP-complete problems have been used to solve the problem such as *bin-packing*, *0-1 integer programming*, and *graph coloring* based approaches.

2.1. Graph coloring register allocation

The idea behind coloring-based register allocation schemes is to formulate register allocation as a graph coloring problem [8] by representing liveness information with an *interference graph*. Nodes in the interference graph represent temporaries that are to be allocated to registers. Edges connect temporaries that are simultaneously live and thus cannot use the same physical register. By using as many colors as allocatable 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.

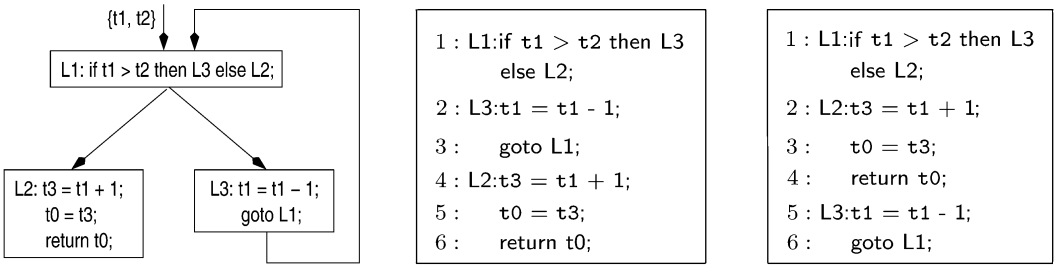
The classic heuristics-based method by Chaitin *et al.* [8,10] iteratively builds an interference graph, *aggressively* coalesces any pair of non-interfering, `move`-related nodes, and heuristically attempts to color the resulting graph by simplification (i.e., removal of nodes with degree less than the number of available machine registers). If the graph is not colorable in this way, 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 [11,12] or improvements [13–15] to the basic scheme have emerged and some of them have been incorporated in production compilers.

2.2. Iterated register coalescing

Iterated register coalescing, proposed by George and Appel [16], is a coloring-based technique aiming at combining register allocation with an aggressive elimination of 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. This coalescing of nodes a and b is iteratively performed during simplification if, for every neighbor t of a, either t already interferes with b or t is of insignificant degree. This coalescing criterion is *conservative*, i.e., coalescing will be performed only if it does not affect the colorability of the graph.

In practice, coloring-based allocation schemes usually produce good code. However, the cost of register allocation is often heavily dominated by the construction of the interference graph, which can take time (and space) quadratic in the number of nodes. Moreover, since the coloring process is heuristics-based, there is no guarantee that the number of iterations will be bounded



(a) (b) (c)

Figure 1. Control-flow graph (a) and two of its possible linearizations: (b) linearization 1; (c) linearization 2.

(by a constant). When compilation time is a concern, as in just-in-time compilers or interactive development environments, graph coloring or iterated register coalescing may not be the best method to employ for register allocation.

2.3. Linear scan register allocation

The linear scan allocation algorithm, proposed by Poletto and Sarkar [3], is simple to understand and implement. Moreover, as its name implies, its execution time is linear in the number of instructions and temporaries. 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. Note that this is an approximation as 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.

Let us look at each step of the algorithm, using Figure 1 as our example.

2.3.1. 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 simple 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 of course 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 and as such contrary to the spirit of the linear scan algorithm. It is therefore important to *a priori* choose an ordering that performs best on the average. Poletto and Sarkar in [3] 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 many other different orderings and discuss their impact on the quality of the produced code in Section 5.1.

2.3.2. *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:

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

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, `t2` is simultaneously live with all other temporaries.

A natural improvement to the above calculation of lifetime intervals is to also employ a scheme such as that described in [17] for utilizing lifetime holes or to perform some form of live range splitting. We will remark on the use of these methods in Section 5.4. Also, there are alternatives to performing the somewhat costly liveness analysis. Such alternatives give correct—albeit sub-optimal—live intervals. One approach is to use strongly connected components in the control-flow graph; see [3]. We will look closer at this alternative in Section 5.2.

2.3.3. *Allocation of registers to intervals*

When all intervals are computed, the resulting data structure (Intervals) gets ordered in increasing start-points 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 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 $\texttt{t}_i$ (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 $\texttt{t}_i \mapsto r$ with the interval $\texttt{t}_i : [start_i, end_i]$ to Active (sorted on *end*).

Alternatively, if 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 interval (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 5.3.)

2.3.4. 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 reserved 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 register allocator

The graph coloring register allocator is a simple variant of [13], is well-tested, and uses efficient data structures. It uses a rather simple spill cost function: the static count of a temporary's uses (i.e., the number of occurrences of the temporary in the function's code). This cost function works quite well since loops in our context are few in number and of low nesting depth; see Section 3.5 for reasons why this is so. A natural improvement would be to use the static prediction to give higher spill cost to temporaries used in the most likely taken path. We have not investigated the effects of other cost functions since this allocator seldom needs to spill on register-rich architectures such as the SPARC, and we subsequently developed an iterated register coalescing allocator which typically spills in even fewer cases and therefore is to be preferred when compilation time is not a concern.

In the graph coloring register allocator, the following three steps are iterated until no new temporaries are added: (1) build interference graph; (2) color the graph; and (3) rewrite instructions with spills to use new temporaries. These steps are described in more detail below.

3.1.1. Build interference graph

To build the interference graph we first calculate liveness information for the temporaries and we then traverse the instructions inserting edges between interfering temporaries into the interference graph. While doing this, we also calculate the number of uses of each temporary. This number is used in the spill cost function.

3.1.2. Color the graph

Coloring is done straightforwardly. We use a work-list `Low`, a stack of colorable registers `Stack`, and apply the algorithm shown in Figure 2. In the first step all nodes are examined to find nodes of insignificant degrees. The total number of iterations of the two while loops is related to the number


```
1. Low is initialized to contain all nodes of insignificant degree   /* i.e., trivially colorable */
2. While the interference graph is non-empty
      While Low is non-empty
         Remove X from the interference graph
         Remove X from Low
         Push {X, colorable} on the Stack
         Decrement degree of neighbors of X
         For each neighbor Y of low degree, put Y on Low
      otherwise
         Select a node Z to spill
         Remove Z from the interference graph
         Push {Z, spilled} on the Stack
         Decrement the degree of neighbors of Z
         Add all insignificant degree neighbors of Z to Low
3. Traverse the Stack choosing a free color for each node not marked as spilled.
```

Figure 2. Heuristic graph coloring implementation.

of nodes (in each iteration a node is removed from the interference graph and pushed on the stack). The operations done at each step of the while loop are relative to the degree of the node being pushed. When no spilling occurs this degree is less than K (the number of available registers), otherwise the degree is only bounded by the number of nodes, but in practice few nodes are related to all other nodes. In the third step all neighbors of each node that is not marked as spilled have to be examined in order to find a free color.

3.1.3. Rewrite of the code

The final step of the graph coloring implementation traverses the code and rewrites each instruction so that the constraints imposed by the instruction set architecture are respected. We explain how this is done in Sections 3.6 and 3.7, which present the SPARC and IA-32 back-ends, respectively. This rewrite might introduce new temporaries, in which case the allocation is repeated with the added constraint that none of these new temporaries may be spilled.

3.2. Iterated register coalescing allocator

The iterated register coalescing allocator closely follows the algorithm described in [16]. It is *optimistic* in its spilling (similar to the strategy described in [15]), and was implemented having as its main goal to provide the HiPE compiler with a register allocator that offers good performance when the speed of compilation is not critical. As far as this article is concerned, this allocator establishes an approximate (practical) lower bound on the number of spills for the benchmarks. However, note that a technique for optimal spilling by optimal live range splitting and taking into account fancy addressing


```
While the interference graph is non-empty:
    /* In each step below, the interference graph is updated */
    Simplify: remove all non move-related nodes of insignificant degree.
    Coalesce: join all move-related nodes satisfying the coalescing criteria.
    If any coalescing was performed, start over with the simplify step.
    Freeze nodes: mark a move-related node of low degree as non move-related.
    If any freezing was done, start over with the simplify step.
    Otherwise spill a node.
```

Figure 3. Iterated register coalescing implementation.

modes that machines like the IA-32 offer has recently been proposed. This technique achieves an even lower number of spills than iterated coalescing, albeit at the expense of (often significantly) increased compilation times; see [18].

The main structure of the iterated coalescing allocator (Figure 3) is similar to that of the graph coloring allocator; the difference lies mainly in the coloring of the interference graph. When the interference graph is built all nodes are marked as either *move-related*, i.e. the source or destination of a move, or *non move-related*, i.e. never the source or destination of a move. Two move-related nodes, a and b, might be coalesced if they satisfy the coalescing criteria. (As mentioned previously the criterion is that for every neighbor t of a, either t already interferes with b or t is of insignificant degree.) If there are move-related nodes that cannot be coalesced, a low degree node might be freezed, i.e. marked as non move-related (any neighbors that were move-related to only this node will also be marked as non move-related, recursively).

By keeping each node of the interference graph in one of four work-lists (simplify, coalesce, freeze, or spill) we can easily find all nodes matching the current step of the algorithm. The cost for each step of the algorithm is proportional to the degree of the node being processed and the length of the work-list of neighboring nodes. We can get an upper bound of the number of iterations of the algorithm by considering how many times each operation might be performed on each node. For each node either the simplify or the spill action will be performed once. In addition the freeze action might be performed once for each node, and the coalescing might be performed at most once with every other node.

3.3. Linear scan register allocator

As mentioned, the linear scan register allocator was first implemented based on the description in [3]. 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 [3] by describing and quantifying the impact of the alternatives that we tried.

3.3.1. *Ordering instructions linearly*

Since there is no reason to reorder the instructions within a basic block, we order only the basic blocks. The default ordering of blocks is *depth-first ordering* (also called *reverse postorder*). The impact of basic block orderings on the quality of the allocation obtained by linear scan is quantified in Section 5.1.

3.3.2. *Calculation of live intervals*

Each live interval consists of a start and an end position: given a fixed traversal of the basic blocks, these are instruction numbers corresponding to the first and last instruction where the temporary is live. Given an approximation of the live-in and live-out sets for each basic block, we can set up the live intervals for each temporary by traversing the 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. If a temporary, `t`, occurring in an instruction of the basic block is not included in the live-in set, then the live interval of `t` needs to be extended to the first instruction defining `t`. If `t` is not included in the live-out set, the live interval of `t` needs to be extended to the last instruction using `t` within the basic block. If `t` is not included in either the live-in or the live-out set, then a new interval is added starting at the first definition and ending with the last use of `t` within the basic block.

3.3.3. *Allocation of registers to intervals*

This step uses the following data structures:

Intervals A list of `{Temporary,StartPoint,EndPoint}` triples. This is a sorted (on increasing `StartPoint`) representation of the interval structure calculated in the step described in Section 3.3.2.

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, in order to deallocate physical registers when the allocation has passed the `EndPoint`. This list 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 either the form `{Temporary,{reg,PhysReg}}` or `{Temporary,{spill,Location}}`.

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

1. Moves to the Allocation structure the information about each interval in the Active list that ends before or at `StartPoint`. Also adds the physical register assigned to the interval which has now ended to the FreeRegs list.

2. Finds an allocation for the current interval:

    - If there is a physical register in the FreeRegs list then tentatively assigns 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 the interval with the furthest `EndPoint` is spilled and moved into the Allocation list. If the spilled interval is not the current one, the step taken is to assign the physical register of the interval that was spilled to the current interval.

3.3.4. Rewrite of the code

When all intervals are processed, the Allocation structure is turned into a data structure that can be used for $O(1)$ mapping from a temporary to its allocated physical register (or spill position).

In contrast to the graph coloring and iterated coalescing allocators, the linear scan register allocator does not use an iterative process to handle spills. We instead reserve two registers so that they are not used during allocation; these registers can then be used to rewrite instructions that use spilled registers. The downside of doing so is that our implementation of the linear scan register allocator will spill slightly more often than really necessary. However, we found that this keeps compilation times down for functions that spill, requiring just one more linear pass over the code to rewrite instructions in accordance to the allocation mapping.

3.4. A naïve register allocator

To establish a baseline 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 temporary uses and definitions in the program. This register allocator is very fast since it only needs one pass over the code, but the added `loads` and `stores` 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.

3.5. The HiPE compiler: issues common to all back-ends

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. How many registers are preallocated in this way depends on the number of registers of the target architecture and is explained in Sections 3.6 and 3.7 below. Because of not starting the allocation with the full set of available machine registers, compilation of Erlang programs differs from the contexts to which linear scan has been previously applied [4,17]. Other differences of the code handled by the HiPE compiler compared

with a compiler for, e.g., C or FORTRAN code are the relatively high number of function calls, the presence of tail-calls, and the relatively low number and nesting depth of loops[§].

Starting from a symbolic form of bytecode for a register-based virtual machine called BEAM, the code is compiled to our internal 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. [9]. Since the bytecode from which compilation starts is already register allocated, but for registers of the virtual machine, it contains many artificial dependencies between these registers. These dependencies follow the code during the translation from the bytecode into HiPE's intermediate code representation (which has an unlimited number of temporaries) and can have a negative impact on the performance of some optimizations in the compiler. To remedy this, we can perform either some ad hoc renaming pass or a more systematic conversion to a static single assignment (SSA) form early in the compiler.

Memory management in the code generated by the HiPE compiler is performed using a precise generational copying garbage collector [19,2]. To keep garbage collection times low, garbage objects should appear dead to the garbage collector as soon as possible, preferably from the exact moment that they become unreachable. 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 this end, the HiPE compiler generates *stack descriptors* (also known as *stack maps* [19]) to indicate to the garbage collector which stack slots are live and which are dead.

3.6. The SPARC back-end

On the register-rich SPARC architecture, we have chosen to cache six data structures in registers (the stack pointer, stack limit, heap pointer, heap limit, a pointer to the process control block, and the reduction counter); see Table I. We have reserved one register (%g1) that the assembler can use to shuffle arguments on the stack at tail-calls. There are another five registers that the HiPE compiler cannot use (Zero (%g0), C's SP (%o6) and FP (%i6), and the two SPARC ABI reserved registers (%g6 and %g7)). Since we are using the ordinary SPARC `call` instruction, the return address is saved in %o7. (At the moment we do not let the register allocators use this register even in non-leaf functions where the return address is also saved on the stack.) We use 16 registers to pass arguments to Erlang functions, but since these can also be used by the register allocators, we get a total of 19 allocatable registers. For linear scan, two of these 19 registers (%l6 and %l7) are reserved to handle spills without having to iterate the allocation process.

The final step of all register allocators traverses the code and rewrites each instruction—except `move` instructions whose handling is postponed—that defines or uses a spilled temporary. For each use we first insert an instruction that moves the spilled temporary to a new temporary and then we rewrite the original instruction so that it uses the new temporary instead. For instructions defining a spilled temporary, we do a similar rewrite. After this rewrite, spilled temporaries are only referenced by `move`

[§]Recursion is the only way to express iteration in Erlang. Functions that are self-tail-recursive are transformed into loops, but in these cases most instructions in the function will be inside the loop. Also, since the HiPE compiler currently does not inline self-recursive functions, the loop nesting depth will never be higher than one in our context. The loop nesting depth is therefore not as important for the cost function as it is in FORTRAN or C compilers.

Table I. Use of SPARC registers in HiPE. M: A, allocatable; R, reserved, G, global; –, reserved by C/OS; 0, zero.

reg	Name	M	Note	reg	Name	M	Note
%g0	ZERO	0	Always 0	%l0	ARG6	A	(caller-save)
%g1	TEMP0	R	Scratch register	%l1	ARG7	A	(caller-save)
%g2	ARG11	A	(caller-save)	%l2	ARG8	A	(caller-save)
%g3	ARG12	A	(caller-save)	%l3	ARG9	A	(caller-save)
%g4	ARG13	A	(caller-save)	%l4	ARG10	A	(caller-save)
%g5	ARG14	A	(caller-save)	%l5	TEMP3	A	Local scratch
%g6	[OS]	–	Reserved by OS	%l6	TEMP2	A	Used in emu $\Leftrightarrow$ native transitions
%g7	[OS]	–	Reserved by OS	%l7	TEMP1	A	Local scratch
%o0	ARG16	A	Return value	%i0	P	G	Current process pointer
%o1	ARG1	A	(caller-save)	%i1	HP	G	Heap pointer
%o2	ARG2	A	(caller-save)	%i2	H-limit	G	Heap limit
%o3	ARG3	A	(caller-save)	%i3	SP	G	Stack pointer
%o4	ARG4	A	(caller-save)	%i4	S-limit	G	Stack limit
%o5	ARG5	A	(caller-save)	%i5	FCALLS	G	Reduction counter
%o6	[sp]	–	C-stack pointer	%i6	[fp]	–	C-frame pointer
%o7	RA/CP	G	Return address	%i7	ARG15	A	(caller-save)

instructions which can be replaced by `loads` and `stores` in a later stage that performs the register assignment and frame handling.

3.7. The IA-32 back-end

On the IA-32, the allocators assume that the `call` instruction defines all physical registers, thus preventing temporaries that are live across a function call from being allocated to a physical register. This means that all these temporaries will be allocated on (spilled to) the stack. The approaches used by the two back-ends differ when a temporary that lives across function calls needs to be read from or written to memory. In the worst case, on the SPARC, a read might be 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 functional language with assign once variables such as Erlang we suspect that the number of temporary uses plus the number of definitions 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.

[¶]On the SPARC, we do the obvious and easy optimization of eliminating redundant load/store pairs, e.g., `ld [%sp+N],rM; ... ; st rM,[%sp+N]`, where none of the instructions between the `ld` and the `st` accesses `rM`. The analogous optimization of removing the second `ld` in the IA-32 equivalent of `ld [%sp+N],rM; <use rM>; ld [%sp+N],rM; <use rM>` is currently not performed because the spilled value is most often not read to a register (it is used directly by the instruction instead). Consequently, the opportunities to perform this optimization on the IA-32 are quite low in number.

Table II. Use of IA-32 registers in HiPE. M: A, allocatable; G, global.

reg	Name	M	Note	reg	Name	M	Note
%eax		A	(caller-save)	%esp	SP	G	Stack pointer
%ebx		A	(caller-save)	%ebp	P	G	Current process pointer
%ecx		A	(caller-save)	%esi	HP	G	Heap pointer
%edx	TEMP0	A	(caller-save)	%edi	TEMP1	A	(caller-save)

We preallocate much fewer, only three, registers on the IA-32: the stack pointer is allocated to %esp, the process pointer to %ebp, and the heap pointer is allocated to %esi; see Table II. 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 will keep them in memory (on the stack). The return value, however, 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 five registers for the register allocators to play with. As on the SPARC, for linear scan, two of these five registers (%ebx and %ebi) are reserved to handle spills without ever having to iterate the allocation process.

Most instructions of the IA-32 instruction set architecture 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 instruction set architecture, e.g., that no binary operation uses more than one memory operand. This is done by rewriting the code to use `loads` and `stores` to new temporaries. If any instruction has to be rewritten in this way, then the register allocator is called again with the additional constraint that none of the newly introduced temporaries may be spilled.

The naïve register 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 `loads` or `stores` are inserted, they use a preallocated physical register instead of introducing a new temporary.)

3.8. Tweaks for linear scan on the IA-32

All published accounts of experience using the linear scan register allocator have so far been in the context of register-rich architectures (mainly Alphas) using a calling convention similar to the one used by our SPARC back-end. When adapting linear scan to work in the context of the IA-32 back-end, we found out that some small adjustments to the basic algorithm were needed. We describe them below.

First, since we represent physical registers in the register allocator as preallocated temporaries, we had to slightly adjust the linear scan allocator to handle the case when a temporary is defined several

times but never used. In the calling convention used by the IA-32 back-end, such is the case with physical registers at function calls. For these temporaries we effectively perform *live range splitting* (see e.g. [14]) 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. However, this could render allocation of other temporaries impossible, and thus disallow the use of linear scan.

We also discovered that extra care has to be taken when preallocated registers[‖] are used with the linear scan algorithm. This is a generic issue, but it manifests itself more often on a register-poor architecture as the IA-32. The crude approximation of live ranges by live intervals used by linear scan forces a temporary to appear live from its first definition to its last use. If some physical register is used often, for example for parameter passing, then this register will appear live throughout most of the code, preventing any other temporary from being allocated to that register. If many, or even all the physical registers are preallocated, special care must be taken or the program will not be allocatable at all. One way of avoiding this situation is by handling preallocated registers separately, for example by allowing them to have several live intervals. (A non-preallocated temporary that is live through most of the code is not a problem, since it can be handled by, e.g., live range splitting, or by simply spilling the temporary. A preallocated register, however, cannot be spilled since its use indicates that the value really has to be in that register.)

4. PERFORMANCE EVALUATION

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 1.1) and a compiler option indicates which one to use. We have compiled the code in the same way before and after applying each allocator.

We present most of the measurements of this section in two views: one without SSA conversion and one with SSA conversion. In doing so, we also evaluate in detail the impact of a systematic renaming pass prior to register allocation in general and to the linear scan algorithm in particular. For register allocation, SSA conversion is a mixed blessing: on the one hand it introduces many new temporaries, but on the other it reduces the live ranges of temporaries and achieves an effect similar to performing *live range splitting*.

The two platforms on which we performed the experimental evaluation are: a Pentium-III 850 MHz, 256 MB memory Dell Latitude laptop running Linux, and a dual-processor Sun Enterprise 3000, 1.2 GB main memory running Solaris 7. Each processor is a 248 MHz UltraSPARC-II. (However, the HiPE system uses only one processor.)

4.1. The benchmarks

The set of benchmarks we used together with a brief description of them appears in Table III. Some of them (decode, eddie) have been chosen from the 'standard' set of Erlang benchmarks because they

[‖] In the context of coloring-based allocators, these registers are often referred to as 'precolored'.

Table III. Description of benchmark programs.

quicksort	Ordinary quicksort. Sorts a list with 45 000 elements 30 times.
spillstress	A synthetic benchmark consisting of a recursive function with several continuously live variables; its only purpose is to stress the register allocators.
smith	The Smith-Waterman DNA sequence matching algorithm. Matches one sequence against 100 others; all of length 32.
life†	Executes 1000 generations in 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 decoding an incoming message.
huff	A Huffman encoder compressing and uncompressing a short string.
md5	Calculates an MD5-checksum on a file. The benchmark takes a file of size 32 026 bytes and concatenates it 10 times before calculating its checksum twice.
prettypr	Consists mainly of a very large function which formats its input (a large file) for printing, using a strict-style context passing implementation.
estone†	Measures the ranking in 'estones' of an Erlang implementation. This is a 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. Because of its size, this function is problematic for some register allocators. To get measurable execution times, we run this benchmark 10 times.
raytracer	A raytracer that traces a scene with 11 objects (two of them with textures) and two light sources to a 80×70 24-bit color bitmap file in ppm format. The time to perform I/O is not included in the benchmark's execution time.
eddie†	An Erlang implementation of an HTTP parser which handles http-get requests.

incur spilling when compiled with linear scan. Note that, on the SPARC, no allocator spills on many other standard Erlang benchmarks. We have also included a module of the HiPE compiler (beam2icode) containing a very large function which is quite troublesome for some register allocators. Benchmarks which are concurrent are marked with a †. Of them, only life spends a significant amount of its execution time in routines of the runtime system that support concurrency (e.g., the scheduler).

Sizes of benchmark programs (lines of source code, the number of temporaries and the number of instructions before register allocation) for both SPARC and IA-32 are shown in Table IV. Benchmark programs marked with a ‡ use functions from standard Erlang libraries that are also dynamically compiled. The lines reported are the number of lines excluding functions from libraries, but the other columns in the table (and compilation times in the subsequent tables) include measurements for library functions. The table shows the number of temporaries and instruction without SSA conversion and with SSA conversion.

The SSA conversion often adds a significant number of temporaries (e.g., over 2500 for beam2icode). However, due to better opportunities for optimizations, the number of instructions on the SPARC is often reduced with SSA conversion. Smaller programs, where there are not as many opportunities for optimizations as in large programs, might increase in size due to added `move` instructions at *ϕ-nodes* during the SSA conversion.

Table IV. Sizes of benchmark programs.

		SPARC				IA-32			
		Temporaries		Instructions		Temporaries		Instructions	
Benchmark	Lines	no SSA	SSA	no SSA	SSA	no SSA	SSA	no SSA	SSA
quicksort	41	87	109	414	428	83	111	554	573
spillstress	94	74	112	573	623	81	156	768	764
smith	93	337	410	1765	1418	301	374	1596	1609
life	189	344	449	1973	1695	292	400	1916	1998
decode	381	330	552	2521	2157	288	510	2392	2536
huff	177	473	615	3114	2582	424	573	3100	3073
md5	286	611	882	4601	3950	543	809	4190	4013
prettypr	1051	1336	1992	10 070	7326	1047	1718	8696	8905
estone‡	1134	1958	2662	12 257	10 291	1674	2456	12 095	12 429
beam2icode	1704	3046	5559	26 115	21 026	2422	4935	24 042	24 663
raytracer‡	924	4617	6149	29 220	23 718	3778	5304	25 439	26 195
eddie‡	2233	5022	6651	31 660	24 685	4191	5889	28 609	29 380

The BEAM has one register (`x0`) that is heavily used. It is often the case that BEAM code looks like:

```
x0 := x0 + x1
```

Without SSA conversion, this maps nicely to the 2-address instructions on the IA-32. However, after SSA conversion the above code is turned into:

```
t3 := t1 + t2
```

Now this code maps nicely to the 3-address instructions on the SPARC, but on the IA-32, it has to be translated to:

```
t3 := t1
t3 := t3 + t2
```

As a result, SSA conversion tends to increase the code sizes more for IA-32 than for SPARC.

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 (minimum of three compilations) are presented in Figures 4–7 where bars show the total compilation time and their stripped part stands for the time spent in the register allocator. In all these figures, the compilation time using the naïve allocator is denoted by N, and the ones using linear scan, graph coloring, and iterated register coalescing by L, G, and C, respectively.

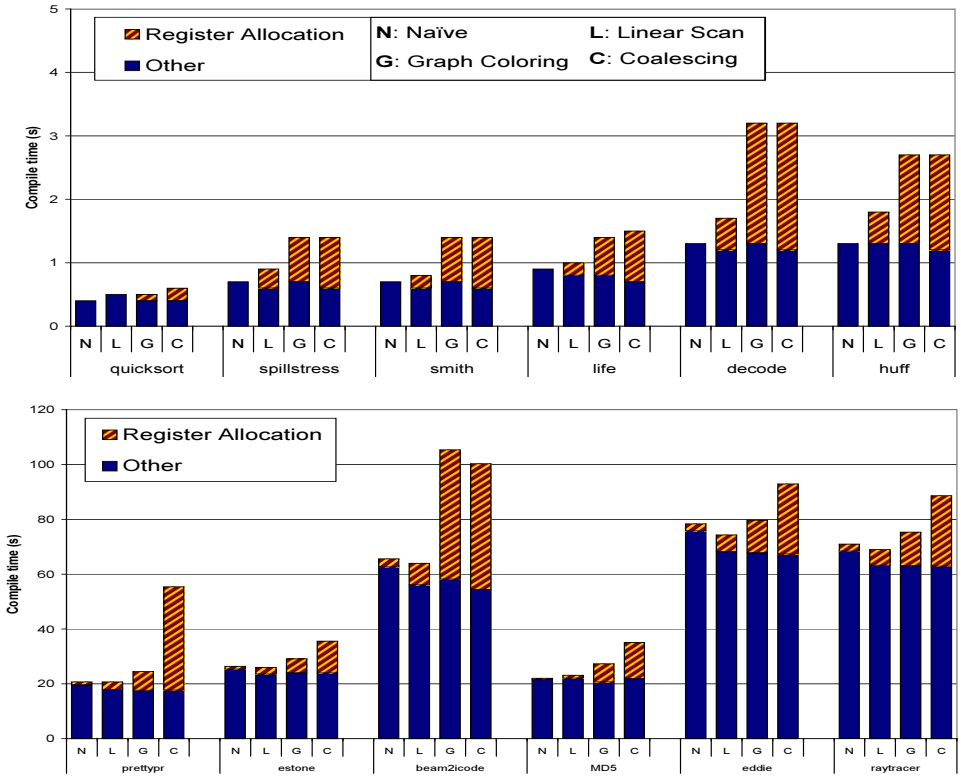


Figure 4. Compilation times on SPARC.

In general, both compilation times and register allocation times increase with SSA conversion, even when the number of instructions is reduced. 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 decode and prettypr benchmarks. In contrast, 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 and register allocation times of beam2icode stand out since, as mentioned, this program contains a 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 and SSA conversion on the SPARC), or when the number of available registers is low, the produced allocation does not respect the constraints imposed by the instruction

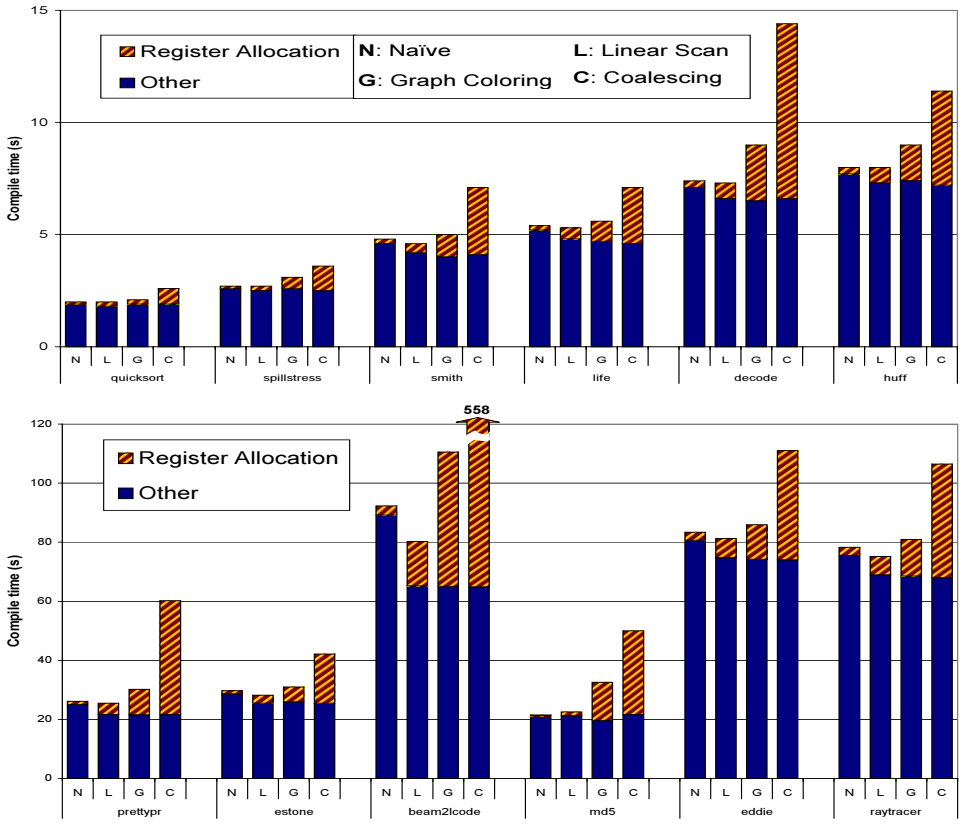


Figure 5. Compilation times, with SSA conversion, on SPARC.

set architecture, and small corrections to the allocation are needed (such is the case on the IA-32). However, estone, raytracer, and eddie which are also big programs consist of a large number of small functions that do not exhibit this behavior to the same extent.

Compilation-time-wise, linear scan performs very well: compared with graph coloring, the time for register allocation is significantly reduced (by at least 50% in general), and pathological cases such as beam2icode are avoided. In fact, for eddie and especially for beam2icode with SSA conversion, compilation with linear scan is even faster than the naïve algorithm; see e.g. Figure 5. This is due to the time needed for rewrite of the code with the obtained allocation. Due to excessive spilling this code is larger for the naïve allocator than it is for linear scan; cf. also Table V.

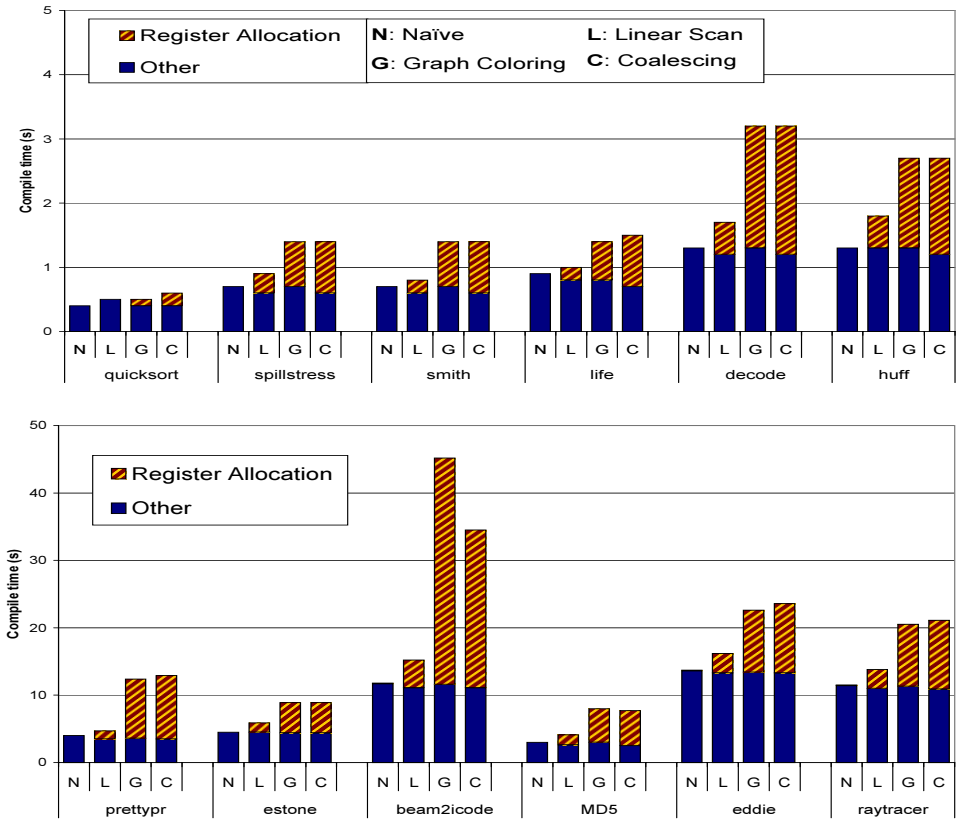


Figure 6. Compilation times on IA-32.

4.3. Execution times

Normalized execution times (w.r.t. the naïve allocator) for each benchmark and allocator are presented in Figures 8–11. These times correspond to the minimum of nine executions of the programs. For the estone benchmark, which contains artificial delays and its execution time does not make sense, we report the number of 'estones' (Erlang stones) assigned to each execution in Figures 12 and 13. Contrary to execution times, more estones means faster execution.

Even though linear scan and graph coloring spill more than the iterated coalescing allocator (see data in Tables V–VIII), the effect of spilling on execution times is limited. On a register-rich architecture such as the SPARC, linear scan offers in most cases performance comparable to that of the graph coloring and iterated coalescing allocators. Despite the low number of available registers, linear scan also performs well on the IA-32. Possible reasons for this are the different calling convention that the

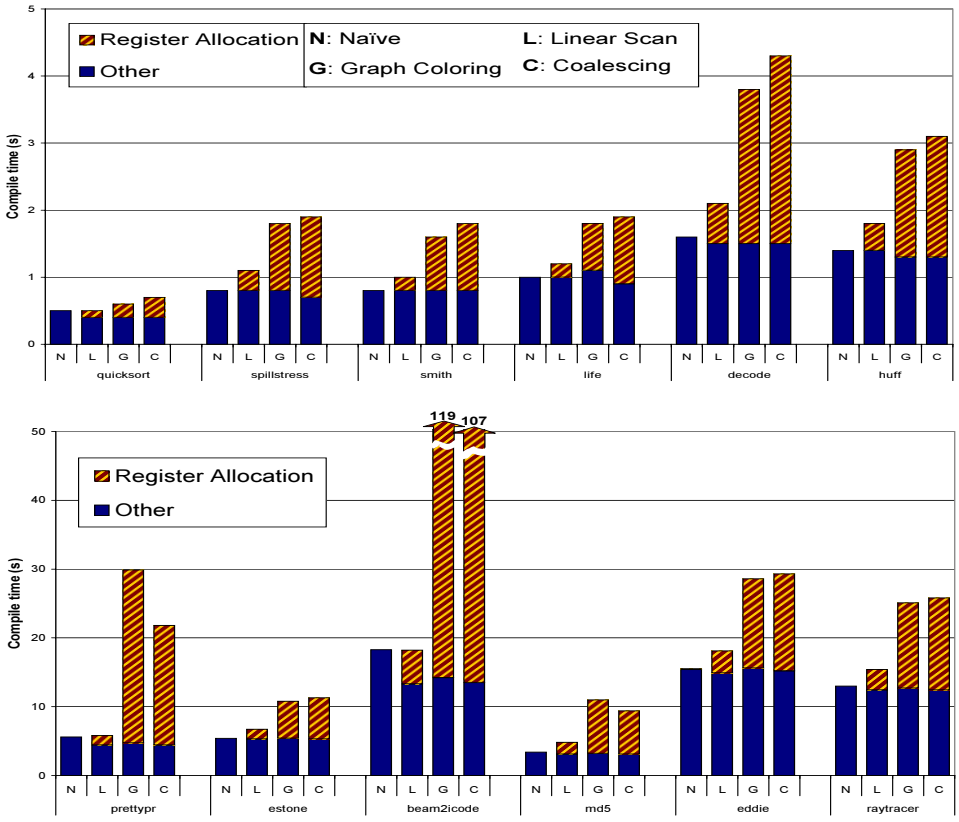


Figure 7. Compilation times, with SSA conversion, on IA-32.

back-end uses (passing arguments on the stack), the fact that the L1 cache can be accessed almost as fast as the registers on the Pentium**, and IA-32's ability to access spilled temporaries directly from the stack in most instructions.

Also, note that different register assignments might affect the dynamic instruction scheduling done by the hardware, causing small differences in execution times. See for example the execution times of quicksort for which none of the allocators spills on SPARC (data shown in Tables V and VI), but still the code produced by the graph coloring allocator executes either marginally faster (without SSA) or marginally slower (with SSA) than the code produced by the other allocators.

**Hardware-level measurements on an Intel Xeon, 2.4 GHz indicate that a register to register move takes about 0.45 clock cycles, a stack to register move takes 0.85 cycles, and a register to stack move takes 1.62 cycles. Still, when dependencies enter the picture memory accesses can take 4–10 cycles while register moves still take around half a clock cycle.

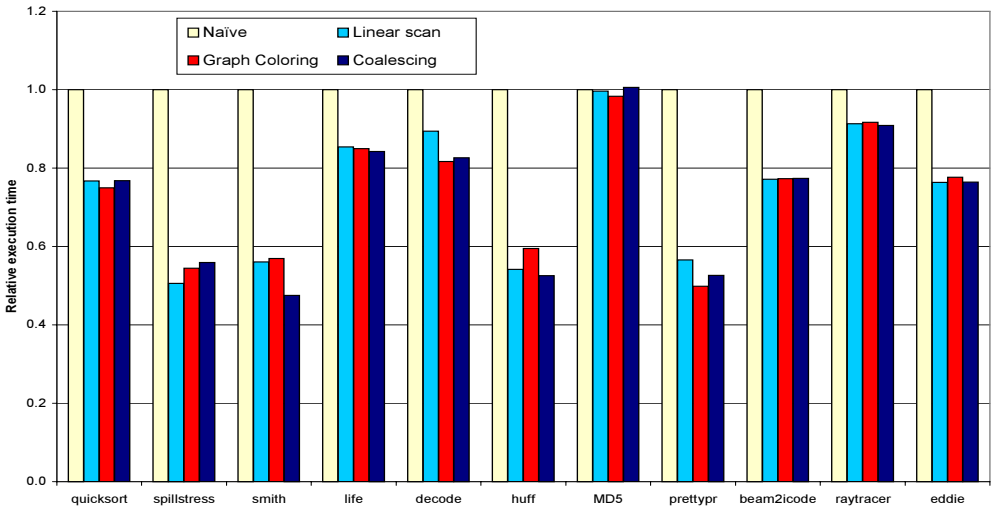


Figure 8. Normalized execution times on SPARC.

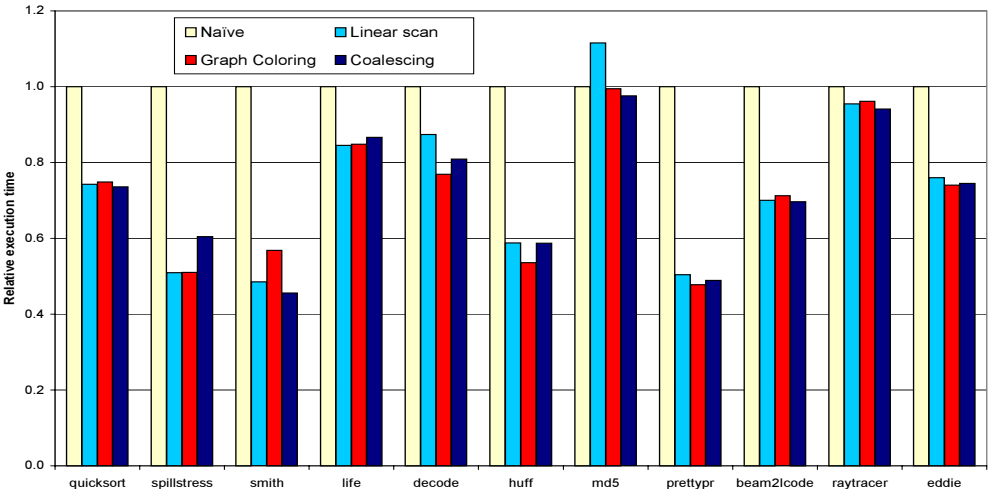


Figure 9. Normalized execution times, with SSA conversion, on SPARC.

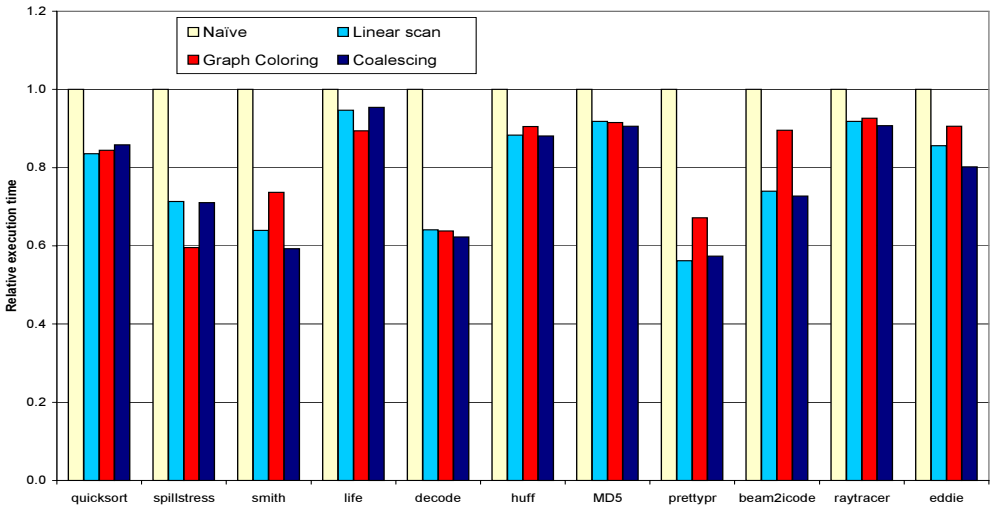


Figure 10. Normalized execution times on IA-32.

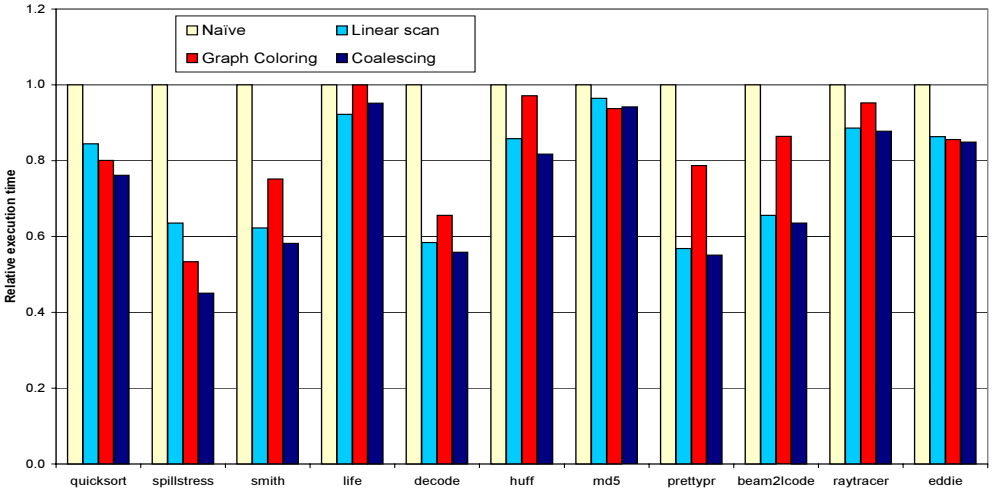


Figure 11. Normalized execution times, with SSA conversion, on IA-32.

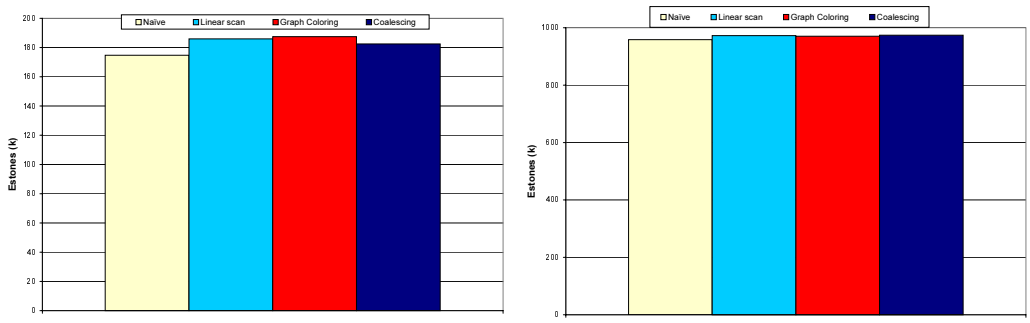


Figure 12. Estone ranking on SPARC (left) and IA-32 (right). The higher the estone, the better.

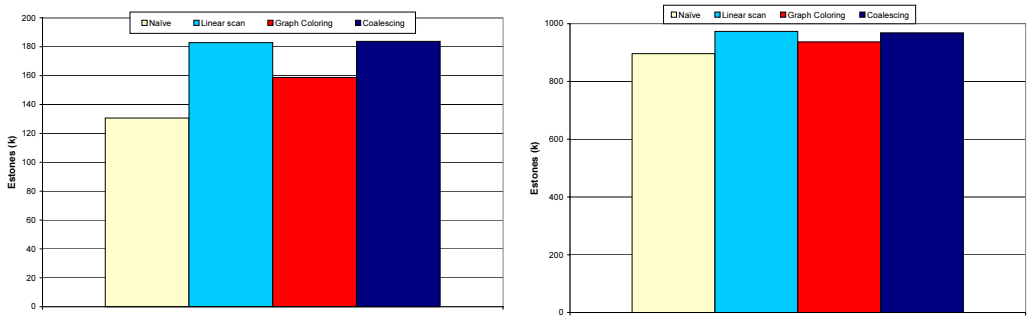


Figure 13. Estone ranking on SPARC (left) and IA-32 (right) with SSA conversion.
The higher the estone ranking, the better.

On the SPARC, one benchmark (`md5`) shows a rather unexpected behavior. This benchmark spends most of its time in calls to built in binary operations (implemented in the runtime system) and very little real work is performed in native code. The impact of register allocation is therefore small. Linear scan, which manages to spill the fewest number of temporaries on this benchmark with SSA (Table VI), performs even worse than the naïve allocator; see Figure 9. This is partly due to the calling convention that is used by the SPARC back-end: linear scan allocates some temporaries that are live over many function calls to registers, forcing these temporaries to be written to and read from the stack several times, while the naïve allocator spills these temporaries to the stack once and for all. The IA-32 back-end, which always spills all temporaries that are live over function calls on the stack, does not suffer from this problem.

4.4. Spills on SPARC

Table V shows the number of temporaries spilled and the number of instructions after allocation without SSA conversion. Table VI shows the same information for compiling these programs with

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

	Naïve		Linear scan		Graph coloring		Iterated coalescing	
	Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
quicksort	87	473	0	341	0	341	0	341
spillstress	74	573	12	482	16	480	8	480
smith	337	1765	1	1164	11	1171	0	1157
life	344	1973	0	1388	4	1456	0	1388
decode	330	2521	28	1965	48	1920	20	1776
huff	473	3114	3	2261	9	2139	0	2124
md5	611	4601	22	3382	57	3495	15	3140
prettypr	1336	10 070	11	7718	14	6714	1	6797
estone	1958	12 257	3	8504	33	8596	1	8455
beam2icode	3046	26 115	38	20 444	83	20 780	4	17 737
raytracer	4617	29 220	61	19 862	124	19 746	0	19 560
eddie	5022	31 660	64	21 242	119	21 118	0	20 861

Table VI. Number of spilled temporaries and SPARC instructions after allocation (with SSA).

	Naïve		Linear scan		Graph coloring		Iterated coalescing	
	Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
quicksort	109	487	0	355	0	355	0	355
spillstress	112	542	11	449	31	451	12	449
smith	410	1776	1	1169	9	1179	0	1168
life	449	2028	1	1450	4	1450	0	1443
decode	552	2615	35	1910	51	1922	20	1870
huff	615	3093	6	2120	8	2115	0	2103
md5	882	4427	21	2988	228	3293	23	2976
prettypr	1992	10 232	31	7109	49	6686	0	6595
estone	2662	12 438	16	8656	31	8691	1	8631
beam2icode	5559	26 527	96	18 562	149	18 204	0	17 998
raytracer	6149	29 758	94	20 448	100	20 220	0	20 098
eddie	6651	32 104	83	21 616	91	21 434	0	21 305

SSA conversion. These numbers show 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 when not using SSA conversion. 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 live ranges and fewer uses. When applying SSA conversion, the number of live ranges increases but they also become shorter which means that the number of instructions can decrease even though the number of spilled temporaries increases.

As expected, the iterated coalescing allocator always generates fewer spilled temporaries. Also, since the coalescing allocator is usually able to coalesce `moves`, the resulting number of instructions is smaller for coalescing even with the same number of spills. As mentioned, the naïve allocator spills all non-allocated temporaries, adding `load` and `store` instructions at each use or definition site. The number of instructions should be compared to the numbers in Table IV to see the increase in size caused by spills introduced by each algorithm. Note that the number of instructions might decrease after register allocation, as some `move` instructions might be removed.

4.5. Spills on IA-32

Table VII reports, for each benchmark, the number of temporaries that are placed on the stack by the calling convention, the number of additional spills, and the number of instructions after allocation. That is, the first column shows the number of temporaries that are live over a function call and hence have to be saved on the stack during the call. The number of spills in the following columns shows the additional number of temporaries that are stored on the stack. Table VIII shows the corresponding numbers with SSA conversion turned on.

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. Compared with 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; sometimes an order of magnitude higher. Compared with the graph colorer, even though the number of spills is often much lower, the number of instructions in the resulting code is lower to a smaller extent, 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 shown in Tables V and VI are not comparable with the numbers in Tables VII and VIII.

5. A DEEPER LOOK ON LINEAR SCAN: IMPACT OF SOME ALTERNATIVES

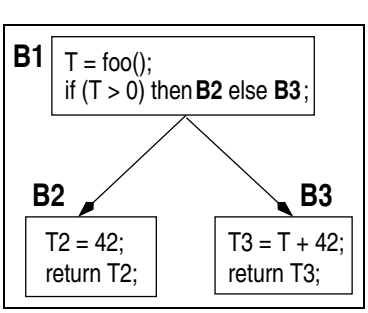
As mentioned, before settling on a default setting, we have experimented with a number of options that one can consider when implementing linear scan. One of these (spilling heuristics) is also considered in [3], the question of whether to perform liveness analysis or not comes naturally, and some others (various orderings) are of our own invention. Nevertheless, as experimenting with all these options

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

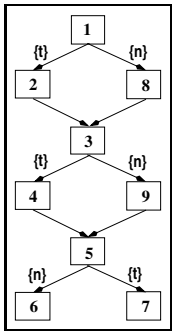
	On stack	Naïve		Linear scan		Graph coloring		Iterated coalescing	
		Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
quicksort	15	68	651	3	608	12	613	0	602
spillstress	31	50	902	6	863	11	877	0	841
smith	44	257	1956	19	1767	90	1833	0	1724
life	70	222	2292	17	2149	65	2178	2	2106
decode	76	212	2849	8	2679	72	2744	1	2664
huff	91	333	3744	44	3450	100	3511	1	3346
md5	74	469	5128	54	4650	95	4792	0	4555
prettypr	61	986	10 933	26	10 055	241	10 764	0	9993
estone	293	1381	14 284	147	13 277	482	13 653	6	12 978
beam2icode	291	2131	29 043	57	27 189	1048	28 519	3	26 974
raytracer	625	3153	31 044	158	28 545	922	29 391	16	28 138
eddie	603	3588	34 908	207	32 042	1208	33 060	22	31 342

Table VIII. Number of spilled temporaries and IA-32 instructions after allocation (with SSA).

	On stack	Naïve		Linear scan		Graph coloring		Iterated coalescing	
		Spills	Instrs	Spills	Instrs	Spills	Instrs	Spills	Instrs
quicksort	33	78	679	2	628	26	651	0	625
spillstress	68	88	890	7	828	16	846	0	819
smith	72	302	1978	27	1763	107	1880	0	1728
life	110	290	2409	26	2214	77	2264	5	2183
decode	111	399	3040	17	2725	136	2860	2	2698
huff	160	413	3712	23	3367	141	3467	1	3312
md5	197	612	4798	84	4262	166	4449	0	4169
prettypr	353	1365	11 247	85	10 084	606	10 781	2	9935
estone	528	1928	14 782	185	13 499	641	14 069	7	13 227
beam2icode	1111	3824	29 863	199	26 825	1614	28 149	12	26 410
raytracer	1178	4126	32 117	253	29 041	1433	30 377	31	28 588
eddie	1048	4841	35 998	274	32 086	1767	33 901	31	31 565



(a)



Postorder	$\langle 6,7,5,4,9,3,2,8,1\rangle$
Rev. postorder	$\langle 1,8,2,3,9,4,5,7,6\rangle$
Inorder	$\langle 6,5,7,4,3,9,2,1,8\rangle$
Rev. inorder	$\langle 8,1,2,9,3,4,7,5,6\rangle$
Prediction	$\langle 1,2,3,4,5,7,6,9,8\rangle$
Preorder	$\langle 1,2,3,4,5,6,7,9,8\rangle$
Breadth-first	$\langle 1,2,8,3,4,9,5,6,7\rangle$
Source	$\langle 1,2,3,4,5,6,7,8,9\rangle$

(b)

Figure 14. Control-flow graphs used to illustrate effects of orderings: (a) a simple control flow graph; (b) a control flow graph and its orderings.

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.

A preliminary, conference version of this article ([20]) also measured the effect of performing an *ad hoc* renaming pass before register allocation. We do not report on that experiment, as the effect of a more systematic renaming pass, based on SSA conversion, has already been extensively presented in the previous section.

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 14(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 from linear in the number of temporaries to exponential in the number of basic blocks. It is therefore important to try to find an ordering that on average gives the best result. To determine such an ordering, we have applied the linear scan algorithm on eight 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. [7,9]) using the control-flow graph shown in Figure 14(b) where edges are annotated with a static prediction (taken/not-taken).

Postorder All children are visited before the node is visited.

Reverse postorder (or ***Depth-first ordering***) The reverse of the order in which nodes are visited in a postorder traversal.

Preorder First the node is visited then the children.

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.

Inorder The left (fallthrough) branch is visited first, then the node followed by other children.

Reverse inorder The reverse of the inorder traversal.

Breadth-first ordering The start node is placed first, then its children followed by the grandchildren and so on.

Source 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 appear in the source and were created.

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 a 'winner' can be found. Tables IX and X show the number of spilled temporaries for each benchmark and ordering. (The number of added instructions is omitted as it shows a similar picture.) As can be seen from these tables, the reverse postorder gives the best result. In HiPE, we are currently using it as the default.

5.2. Impact of performing liveness analysis

In [3], 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 based on liveness analysis. In our benchmark set, even compilation-time-wise linear scan with liveness analysis is faster on more than half of the benchmarks. Also, using liveness analysis gives an execution time speedup ranging from 1.13 to 1.80 compared with execution times obtained using the SCC-based linear scan allocator.

5.3. 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

Table IX. Number of spilled temporaries using different basic block orderings.

	Rev PO	*Post-*	*Pre-*	*Predict*	*Rev IO*	*In-*	*Breadth-*	*Source*
quicksort	0	0	0	0	0	0	0	1
spillstress	12	12	12	12	12	12	12	12
smith	1	6	1	3	7	13	3	15
life	0	1	0	0	0	1	0	1
decode	28	29	30	31	35	35	30	42
huff	3	4	0	4	4	4	0	4
md5	22	22	30	30	38	39	22	30
prettypr	11	10	9	10	15	14	15	60
estone	3	9	10	11	16	22	5	47
beam2icode	38	40	46	51	48	53	46	136
raytracer	61	61	70	75	92	109	75	230
eddie	64	62	66	69	117	139	71	190
Sum	243	256	274	296	384	441	279	768

Table X. Number of spilled temporaries using different basic block orderings (with SSA).

	Rev PO	*Post-*	*Pre-*	*Predict*	*Rev IO*	*In-*	*Breadth-*	*Source*
quicksort	0	0	0	0	0	0	0	5
spillstress	11	11	11	11	11	11	11	33
smith	1	6	4	15	11	21	3	48
life	1	4	2	1	5	8	2	18
decode	35	38	48	54	52	54	35	117
huff	6	7	11	19	7	8	3	74
md5	21	21	125	130	45	49	20	220
prettypr	31	29	39	66	62	60	359	359
estone	16	28	22	38	36	50	10	165
beam2icode	96	117	116	270	272	291	1223	1186
raytracer	94	102	119	181	199	235	213	798
eddie	83	88	84	130	240	279	160	551
Sum	395	451	581	915	940	1066	2039	3574

(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 XI 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 not much bigger, and for life it is even smaller. However, looking at the performance of the generated code, one can see that they perform on a par in many cases, but the usage count heuristic performs much worse on, e.g., decode and prettypr. We thus do not recommend the use of usage counts.

Table XI. Impact of spilling heuristics (with SSA conversion on the SPARC).

	Spills (instructions)		Execution time	
	Interval length	Usage count	Interval length	Usage count
quicksort	0 (355)	0 (355)	11.5	11.9
spillstress	11 (449)	32 (452)	9.0	8.0
smith	1 (1169)	8 (1182)	6.3	7.6
life	1 (1450)	2 (1446)	11.3	11.2
decode	35 (1910)	200 (2114)	11.6	21.5
huff	6 (2120)	83 (2292)	11.8	12.0
md5	21 (2988)	382 (3614)	12.7	12.9
prettypr	31 (7109)	1277 (7326)	7.6	14.0
beam2icode	96 (18 562)	2786 (22 373)	13.9	18.6
raytracer	94 (20 448)	485 (20 843)	11.0	10.7
eddie	83 (21 616)	426 (21 988)	12.5	12.6
estone	16 (8656)	125 (8839)	183 k	155 k

5.4. On using lifetime holes and live range splitting

As stated in the introduction, our motivation for implementing linear scan was to have a fast, provably linear, register allocator to be used in a just-in-time compilation setting. Besides liveness analysis, we have purposely avoided using any technique which requires iteration and could subtly undermine the linear time characteristics of the linear scan algorithm.

We have therefore not considered the use of *lifetime holes* as proposed in [17] (a technique which requires an iterative dataflow calculation to be performed during register allocation and thus does not have a linear time bound) nor have we tried to integrate a separate live range splitting [14] pass in the linear scan register allocator. We feel that either of these approaches would significantly complicate and slow down the allocator without any clear benefits. In our implementation, by using SSA conversion, we usually get most of the benefits from live range splitting, namely mostly short live ranges. Even though there might still be situations where, e.g., a temporary is defined only once and then has several late uses giving it a long live range which might force it to be spilled, we do not think that splitting this live range would result in a significant improvement in execution performance. Some evidence why this is so can be seen from the fact that even though the coalescing register allocator spills much less than our other allocators, the execution time performance of the generated code is not significantly better.

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 being the first to report 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 the results of [3]; in many others, we provide a deeper look or improve on those results in the hope that this will prove useful to other 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. Our experiments show that on the IA-32, when the calling convention passes arguments on the stack anyway, the impact of (modest) spilling on the execution time is limited. This makes linear scan a viable option even though the number of available register on this architecture is small. When compilation time is a concern, or at low optimization levels, linear scan should be used[††]. Disregarding compilation-time concerns, an optimistic iterated coalescing register allocator (which can eliminate most register-register `moves`) is a better approach to obtaining high performance.

ACKNOWLEDGEMENTS

This research was conducted while the second author was affiliated to Uppsala University. The work has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Development. We thank Mikael Pettersson and the anonymous reviewers for comments on the article, Thorild Selén, Andreas Wallin, and Ingemar Åberg for the initial implementation of the coalescing allocator.

REFERENCES

1. Armstrong J, Virding R, Wikström C, Williams M. *Concurrent Programming in Erlang* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1996.
2. Johansson E, Pettersson M, Sagonas K. HiPE: A High Performance Erlang system. *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press: New York, 2000; 32–43.
3. Poletto M, Sarkar V. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems* 1999; **21**(5):895–913.
4. Poletto M, Hsieh WC, Engler DR, Kaashoek MF. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems* 1999; **21**(2):324–369.
5. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 1991; **13**(4):451–490.
6. Sethi R. Complete register allocation problems. *SIAM Journal on Computing* 1975; **4**(3):226–248.
7. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques and Tools*. Addison-Wesley: Reading, MA, 1986.
8. Chaitin GJ. Register allocation & spilling via graph coloring. *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*. ACM Press: New York, 1982; 98–105.
9. Muchnick SS. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers: San Fransisco, CA, 1997.
10. Chaitin GJ, Auslander MA, Chandra AK, Cocke J, Hopkins ME, Markstein PW. Register allocation via coloring. *Computer Languages* 1981; **6**(1):47–57.
11. Chow FC, Hennessy JL. The priority-based coloring approach to register allocation. *ACM Transactions Programming Languages and Systems* 1990; **12**(4):501–536.

[††] HiPE currently uses linear scan by default in optimization levels up to `-O2`; iterated coalescing is used in `-O3`.

12. Hendren LJ, Gao GR, Altman ER, Mukerji C. A register allocation framework based on hierarchical cyclic interval graphs. *Journal of Programming Languages* 1993; **1**(3):155–185.
13. Briggs P, Cooper KD, Torczon L. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems* 1994; **16**(3):428–455.
14. Cooper KD, Simpson LT. Live range splitting in a graph coloring register allocator. *CC'98: Compiler Construction, 7th International Conference*, Koskimies K (ed.) (*Lecture Notes in Computer Science*, vol. 1383). Springer: Berlin, 1998; 174–187.
15. Park J, Moon S-M. Optimistic register coalescing. *Proceedings 1998 International Conference on Parallel Architecture and Compilation Techniques*. IEEE Press: Los Alamitos, CA, 1998; 196–204.
16. George L, Appel AW. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems* 1996; **18**(3):300–324.
17. Traub O, Holloway G, Smith MD. Quality and speed in linear-scan register allocation. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York, 1998; 142–151.
18. Appel AW, George L. Optimal spilling for CISC machines with a few registers. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York, 2001; 243–253.
19. Jones RE, Lins R. *Garbage Collection: Algorithms for Automatic Memory Management*. John Wiley & Sons: New York, 1996.
20. Johansson E, Sagonas K. Linear scan register allocation in a high performance Erlang compiler. *Practical Applications of Declarative Languages: Proceedings of the PADL'2002 Symposium* (*Lecture Notes in Computer Science*, vol. 2257). Springer: Berlin, 2002; 299–317.

