Global Stack Allocation – Register Allocation for Stack Machines

Mark Shannon
University of York
marks@cs.york.ac.uk

Chris Bailey
University of York
chrisb@cs.york.ac.uk

Abstract

Register allocation is a critical part of any compiler, yet register allocation for stack machines has received relatively little attention in the past. We present a framework for the analysis of register allocation methods for stack machines which has allowed us to analyse current methods. We have used this framework to design the first truly procedure-wide register allocation methods for stack machines. We have designed two such methods, both of which outperform current techniques.

This work was funded by the AMADEUS project, part of the DTI's Next Wave Technologies and Markets Program, in collaboration with MPE Ltd.

1 Introduction

To design a compiler for a stack machine most of the conventional techniques for compiler design can be reused, with the exception of register allocation and, to a lesser extent, instruction scheduling. Register allocation for stack machines is fundamentally different from that for conventional architectures, due the arrangement of the registers. In this paper we describe a way of analysing the stack that is suitable for classifying and designing register allocation methods for stack machines. Most compilers specifically targetted at stack machines have been Forth compilers, where register allocation has to be done explicitly by the programmer. When developing a C compiler, however, it is important that it is the compiler handles register allocation since this is not the responsibilty of the programmer.

The first work on register allocation for stack machines was Koopman's work[4], although he uses the term 'stack scheduling', which was limited to basic blocks, although he does discuss the possibility of a global method to further improve this work. This work was later to shown to be near-optimal, in terms of removing memory acccesses, by Maierhofer and Ertl[6], and was extended beyond basic block boundaries by the second author[1]. Although this enhanced method was able to store values on the stack across edges in the flow graph, it has limitations and cannot be considered truly global.

This paper assumes a stack machine for which stack access is considerably faster than memory access, whether real or virtual, and that register allocation is the job of the compiler, not the programmer.

2 The stack

2.1 Views of the stack

It is possible to view the stack from a number of different perspectives. For example, when viewed from a hardware perpsective the stack consists of a number of discrete registers, a mechanism for moving values between these registers, a buffer, and some logic to control movement of data between the buffer and memory. This perspective is irrelevant to the programmer, who sees a first-in first-out stack, of potentially infinite depth, enhanced with a number of instructions allowing access to a few values directly below the top of stack. In oreder to develop register allocation methods a different, more structured view is required.

2.2 Stack regions

To aid analysis of the stack with regard to register allocation, the perspective chosen divides the stack into a number of regions. These regions are abstract, having no direct relation to the hardware and exist solely to assist our thinking. The boundaries between these regions can be moved without any real operation taking place, but only at well defined points and in well defined ways. This compiler oriented view of the stack consists of five regions. Starting from the top, these are:

- The evaluation region (e-stack)
- The parameter region (p-stack)
- The local region (l-stack)

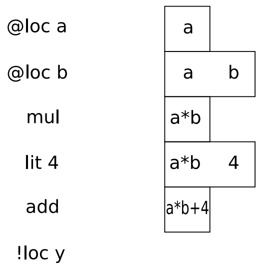
- The transfer region (x-stack)
- The remainder of the stack, included for completeness.

An example of stack region usage is illustrated in figure 6

2.2.1 The evaluation region

The evaluation region, or *e-stack*, is the part of the stack that is used for the evaluation of expressions. It is defined to be empty except during the evaluation of expressions when it will hold any intermediate sub-expressions[1]. See figure 1 for an example.

Figure 1: Evaluation of expression y = a ∗ b + 4

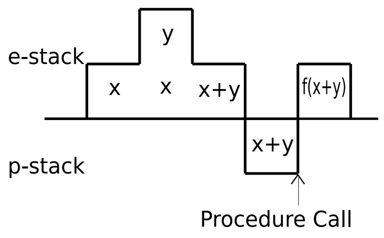


The e-stack is not modified during register allocation. Any compiler optimisations which would alter the e-stack, such as common sub-expression elimination, are presumed to have occurred before register allocation.

2.2.2 The parameter region

The parameter region, or *p-stack*, is used to store parameters for procedure calls. It may have values in it at any point, both in basic blocks[2] and across the boundaries between blocks. When a procedure is invoked all its parameters are removed from the p-stack. The p-stack is for *outgoing* parameters only; any value returned by a procedure is left on the e-stack and incoming parameters are placed in the x-stack at the point of procedure entry. Although parameters are kept on the p-stack before a procedure call, they are evaluated on the e-stack, like any other expression. Only when evaluation of the parameter is completed is it moved to the p-stack. This is illustrated in figure 2. Note that this movement may be entirely abstract; no actual operation need occur. The p-stack is, like the e-stack, fixed during register allocation.

Figure 2: Evaluation of expression f(x+y)



The e-stack and p-stack are the parts of the stack that would be used by a compiler that did no stack allocation. Indeed the stack use of the JVM[5] code produced by most Java[2] compilers corresponds to the e-stack and p-stack.

2.2.3 The local region

The local region, or *l-stack*, is the region directly below the p-stack. The l-stack is used for register allocation. It is always empty at the beginning and end of any basic block, but may contain values between expressions. In the earlier example, no mention was made of where either `a` or `b` came from or where `y` is stored. They could be stored in memory but it is better to keep values in machine registers whenever possible. So let us assume that in the earlier example, `y = a * b + 4`, `a` and `b` are stored in the l-stack, as shown in figure 3. To move `a` and `b` from the l-stack to the e-stack, we can copy them, thus retaining the value on the l-stack, or move them to the e-stack from the l-stack. In this example, `b` might be stored at the top of the l-stack, with `a` directly below it; to move them to the e-stack requires no actual move instruction, merely moving the logically boundary between the e-stack and l-stack. Likewise storing the result, `y`, into the l-stack is a virtual operation.

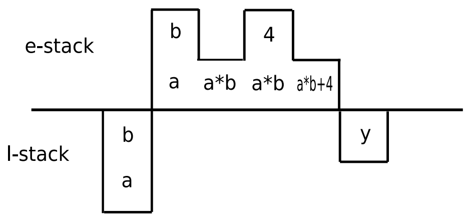
2.2.4 The transfer region

The transfer region or *x-stack* is used to store values both during basic blocks and on edges in the flow graph. The x-stack need only be empty at procedure exit. It holds the incoming parameters at procedure entry. Values may only be moved between the x-stack and l-stack at the beginning or end of basic blocks, and they must moved en bloc and retain their order. Values cannot

[1]This is by definition, any 'expression' that does not fulfil these criteria should be broken down into its constituent parts, possibly creating temporary variables if needed. The conditional expression in C is an example of such a compound expression.

[2]A basic block is a piece of code which has one entry point, at the beginning, and one exit point, at the end. That is, it is a sequence of instructions that must be executed, in order, from start to finish.

Figure 3: Using the l-stack when evaluating y = a ∗ b + 4



be moved directly between the x-stack and the e-stack, they must go through the l-stack. Since all ‘movement’ between the l-stack and x-stack is virtual it might seem that they are the same, but the distinction between the two is useful; the x-stack must be determined globally, while the l-stack can be determined locally. This separation allows a clear distinction between the different phases of allocation and simplifies the analysis.

2.2.5 The rest of the stack

The remainder of the stack or sub-stack, consists of the e-stack, p-stack, l-stack and x-stack of enclosing procedures. It is out-of-bounds for the current procedure.

2.3 Using the regions to do register allocation

Register allocation for stack machines is complicated by the moveable nature of the stack. A value may be stored in one register, yet be in a different one when it is retrieved. This complication can be sidestepped by regarding the boundary between the p- and l-stacks as the fixed point of the stack. Values stored in the l-stack do not move relative to this boundary. The ability of the hardware to reach a point in the l-stack depends on the height of the combined e- and p-stacks above it, but that height is fixed during register allocation, meaning it needs to be calculated only once at the start of register allocation.

2.3.1 The e-stack

The e-stack is unchanged during optimisations. Optimisation changes whether values are moved to the e-stack by reading from memory or by lifting from a lower stack region, but the e-stack itself is unchanged.

2.3.2 The p-stack

For a number of register allocation operations, there is no distinction between the e-stack and p-stack and they can be treated as one region, although the distinction can be useful. For certain optimisations, which are localised and whose scopes do not cross procedure calls, the p-stack and l-stack can merged increasing the usable part of the stack. For the register allocations method discussed later, which are global in scope and can cross procedure calls, the p-stack is treated essentially the same as the e-stack.

2.3.3 The l-stack

The l-stack is the most important region for localised register allocation. All intra-block optimisations operate on this region. Code is improved by retaining variables in the l-stack rather than storing them in memory. Variables must be fetched to the l-stack at the beginning of each basic block and, if they have been altered, restored before the end of the block, since by definition, the l-stack must be empty at the beginning and end of blocks.

2.3.4 The x-stack

The x-stack allows code to be improved across basic block boundaries. The division between the l-stack and x-stack is entirely notional; no actual instructions are inserted to move values from one to the other. Instead the upper portion, or all, of the x-stack forms the l-stack at the beginning of a basic block. Conversely, the l-stack forms the upper portion, or all, of the x-stack at the end of the basic block. Since the e-stack and l-stack are both empty between basic blocks, the p-stack and x-stack represent the complete stack which is legally accessible to the current procedure at those points. This makes the x-stack the critical part of the stack with regards to global register allocation. Code improvements using the x-stack can eliminate local memory accesses entirely by retaining variables on the stack for their entire lifetime.

2.4 How the logical stack regions relate to the real stack

The logical stack regions can be of arbitrary depth regardless of the hardware constraints of the real stack. However, the usability of the l-stack and x-stack depends on the capabilities of the hardware. Our real stack-machine, the UFO, has a number of stack manipulation instructions which allow it to access values up to a fixed depth of four below the top of the stack. However, as the e-stack and p-stack vary in depth, the possible reach into the l-stack also varies. Variables that lie below that depth are unreachable at that point, but, as they may have been reachable earlier and become reachable later, they can still be useful. We assume that the hardware allows uniform access to a fixed number of registers, so if we can copy from the nthregister we can also store to it and rotate through it.

2.5 Edge-sets

The second part of the analytical framework relates to flow-control. In order that programs behave in a sensible way, the stack must be in some predictable and fixed[3] state when program flow moves from one block to another. This means for all the successor edges of any given block, the state of the x-stack must be identical. Likewise, it means that for all the predecessor edges for any given block, the state of the x-stack must be the same. The set of edges for which the stack must contain the same variables is called an *edge-set.* An edge belongs to exactly one edge-set and if two edges share either a predecessor or successor node (block) they must be in the same edge-set. The state of the x-stack is the same for every edge in an edge-set. Edse-sets are defined as follows:

For any edge e and edge-set S_1: if $e \in S_1$ then for all other edge-sets $S_2 \neq S_1$, $e \notin S_2$.

For any two edges, $e_1 \in S_1$, $e_2 \in S_2$: if $\mathit{predecessor}(e_1) = \mathit{predecessor}(e_2) \vee \mathit{successor}(e_1) = \mathit{successor}(e_2)$ then $S_1 = S_2$.

Figure 4: C Factorial Function

```
            C source            | Assembly
--------------------------------+-----------
                                | !loc n
                                | lit 1
                                | !loc f
                                | jump L3
                                | L2:
int fact(int n)                 | @loc f
{                               | @loc n
    int f = 1;                  | mul
    while (n > 0) {             | !loc f
        f = f * n;              | @loc n
        n = n - 1;              | lit 1
    }                           | sub
    return f;                   | !loc n
}                               | L3:
                                | @loc n
                                | lit 0
                                | brgt L2
                                | @loc f
                                | exit
```

3 An example

In order to illuminate the process of using the stack regions to perform register allocation we will use an example. The program code in figure 4 is a simple iterative procedure which returns n factorial for any value of n greater than 0, otherwise it returns 1. The C source code is on the left, along side it is the output from the compiler without any register allocation.

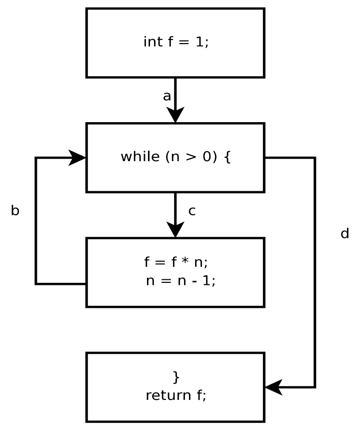
Before register allocation can be done the edge-sets are found; see figure 5. The first part of the stack to be determined is the x-stack. Firstly consider the edge-set $\{a, b\}$; both the variables `n` and `f` are live on this edge set. Presuming that the hardware can manage this, it makes sense to leave both variables in the x-stack. The same considerations apply for $\{c, d\}$, so again both `n` and `f` are retained in the x-stack. The order of variables, whether `n` goes above `f`, or vice versa, also has to be decided. In this example we choose to place `n` above `f`, since `n` is the most used variable, although in this case it does not make a lot of difference.

Once the x-stack has been determined, the l-stack should be generated in a way that minimises memory accesses. This is done by holding those variables which are required by the e-stack in the l-stack, whilst matching the l-stack to the x-stack at the ends of the blocks. Firstly `n`, as the most used variable, is placed in the l-stack. It is required on the l-stack thoughout, except during the evaluation of `n = n+1`, when it is removed, so that the old value of `n` is not kept. Secondly `f` is allocated in the l-stack, directly under `n`. In the final block the value of `n` is superfluous and has to be dropped.

The original and final stack profiles are shown in figure 6. Note the large number of stack manipulations, such as `rrot2` which is equivalent to `swap`, and `rrot1`, which does nothing at all. These virtual stack manipulations serve to mark the 'movement' of variables between the e-stack and l-stack. The final assembly stack code, with redundant operations removed, is shown in figure 7 on the right. Not only is the new code shorter than the original, but the number of memory accesses has been reduced to zero. Although much of the optimisation occurs in the l-stack, the x-stack is vital, since without it variables would have to be stored to memory in each block. Register allocation using only the l-stack can be seen in the centre column of figure 7. This would suggest that the selection of the x-stack is an important factor in register allocation. Although this is a very simple example, the underlying principles can be applied to much larger programs.

[3]A fixed x-stack means that the variables held in it are the same, regardless of the flow up to that point, the values those variables hold may vary.

Figure 5: Determining the edge-sets



The edges a and b share a common child, so form one edge set. The edges c and d share a common parent and form another edge set. So, the two edge-sets are $\{a, b\}$ and $\{c, d\}$

4 Analysis of Existing Algorithms

To demonstrate the value of the framework for analysis we will look at Koopman's and Bailey's methods for 'stack-scheduling', and show that the algorithm can be described more clearly and concisely with reference to our framework. The improvements to Koopman's method by Maierhofer and Ertl are not covered, mainly for space reasons, as they add relatively little to Koopman's work in terms of performance.

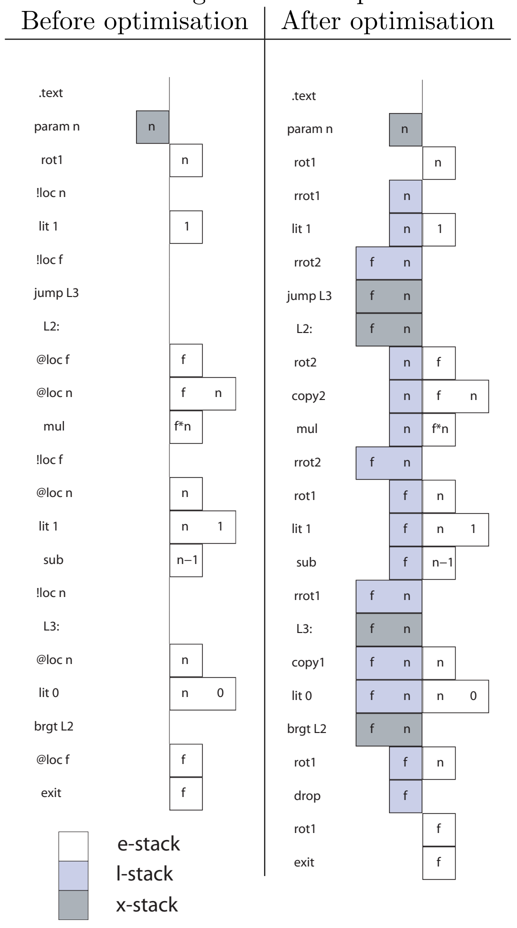
4.1 Koopman's algorithm

Koopman's algorithm, as described in his paper, was implemented as a post processor to the textual output of gcc[7] after partial optimisation. We have implemented it within lcc[3], where it acts directly on the intermediate form.

The algorithm is quite straightforward, as follows:

1. Clean up the output using simple peephole optimisation, replacing sequences of stack manipulations with shorter ones if possible.
2. Locate define–use and use–use pairs of local variables and list them in order of their proximity. That is, in ascending order of the number of instructions separating the pair.
3. For each pair:
   (a) Copy the variable at the point of definition or first use to the bottom of the stack.

Figure 6: Stack profile



   (b) Replace the second instruction with an instruction to rotate the value to the top of the stack.
4. Remove any dead stores.
5. Reapply the peephole optimisation.

4.1.1 Koopman's algorithm in terms of the framework

In Koopman's algorithm, when he refers to the bottom of the stack, he is referring to the portion of the stack used by the function being optimised. Since no inter-block allocation is done, thus the x-stack is empty, the bottom of the stack is clearly the bottom of the *l*-stack. Therefore step 3 above become:

   (a) Copy the variable at the point of definition or first use to the bottom of the *l*-stack.

(b) Replace the second instruction with an instruction to rotate the value from the bottom of the l-stack to the top of the stack.

Figure 7: Assembly listings

No register allocation	Local register allocation	Global register allocation
!loc n	!loc n	lit 1
lit 1	lit 1	swap
!loc f	!loc f	jump L3
jump L3	jump L3	L2:
L2:	L2:	tuck2
@loc f	@loc f	mul
@loc n	@loc n	swap
mul	tuck2	lit 1
!loc f	mul	sub
@loc n	!loc f	L3:
lit 1	lit 1	copy1
sub	sub	lit 0
!loc n	!loc n	brgt L2
L3:	L3:	drop
@loc n	@loc n	exit
lit 0	lit 0	
brgt L2	brgt L2	
@loc f	@loc f	
exit	exit	

4.2 Bailey's 'inter-boundary' algorithm

Bailey's 'inter-boundary' algorithm was the first attempt to utilise the stack across basic block boundaries. This is done by determining edge-sets; although in the paper the algorithm is defined in terms of blocks rather than edges. Then the x-stack, termed 'sub stack inheritance context', is determined for the edge-set. In outline the algorithm runs as follows:

1. Find co-parents and co-children for a block (determine the edge-set).
2. Create an empty 'sub stack inheritance context'.
3. For each variable in a child block, starting with the first to occur:
   - If that variable is present in all co-parents and co-children, then:
     - Test to see if it can be added to the base of the x-stack. This test is done for each co-parent and co-child to see whether the variable would be reachable at the closest point of use in that block.

Bailey's algorithm is designed to be used as a complement to an intra-block optimiser, such as Koopman's. It moves variables onto the stack across edges in the flow graph, by pushing the variables onto the stack immediately before the edge and popping them off the stack immediately after the edge. Without an intra-block optimiser this would actually cause a significant performance drop.

4.2.1 Bailey's algorithm in terms of the framework

1. Determine edge-sets
2. For each edge-set:
   (a) Create an empty x-stack state for that edge-set.
   (b) Determine the intersection of the sets of live variables for each edge in the edge-set.
   (c) Choose an arbitrary neighbouring block, presumably the first to occur in the source code.
   (d) For each variable in the intersection set, in increasing order of the distance of usage from the edge in question:
      - Test to see if it can be added to the x-stack, and if it can be, do so.

Although Bailey's algorithm is an inter-block algorithm, it is not genuinely global, as it makes fairly limited use of the x-stack. No values are left in the x-stack during blocks. No attempt is made to integrate the allocation within the x-stack to allocation within the l-stack. In terms of performance, the main failing of Bailey's algorithm is that it cannot handle variables which are live on some but not all edges of an edge-set.

5 A Global register allocator

The next step forward in register allocation for stack machines, is to try to do it globally, in a procedure wide fashion. Once full data-flow information, including edge-sets, has been found, the next step is to determine the x-stack on each edge-set. Our first approach was to modify Bailey's algorithm to use various combinations of unions and intersections of liveness and uses.

However, this revealed some important limitations in the localised push-on, pop-off approach, which are:

- **Excessive spilling**

  There is no attempt to make the x-stack similar across blocks, so variables may have to be saved at the start of a block, and other variables loaded at the end of a block.

- **Excessive reordering**

  Even when the x-stack state at the start and end of a block contain similar or the same variables, the order may be different and thus require extra instructions.

- **No ability to use the x-stack across blocks**

  The requirement for the entire x-stack to be transfered to the l-stack means that the size of the x-stack is limited. Variables cannot be stored deeper in the stack when they are not required.

5.1 A global approach

The problems to be solved are:

5.1.1 Determination of x-stack member sets

Although none of the modified versions of Bailey's algorithm produced better code than the original, some versions did seem to make promising selections of x-stack members. We decided to determine the x-stack set by starting with a large set of variables and reducing it towards an optimum.

5.1.2 Ordering of the variables within the x-stack

If variables are to be kept on the x-stack during blocks then the order of the lower parts of the x-stack is important. Since the ordering of variables on the x-stack cannot be changed, without moving variables to the l-stack, the order of the lower parts of the x-stack *must* match across blocks. The simple but effective approach taken was to choose a globally fixed ordering. This also solves the problem of excessive reordering of variables.

5.1.3 Handling the l-stacks to work with the x-stack

Since allocation of the l-stack depends on the x-stack at both beginning and end of the block. It is necessary to determine the x-stack first. However, in order to allocate x-stack that do not impede l-stack allocation, the l-stack, must be at least partially determined before the x-stack.

5.2 Outline Algorithm

The algorithm chosen runs, in outline, as follows:

1. Determine edge-sets
2. Determine ordering of variables.
3. For each edge-set:

   Determine x-stack using heuristic

4. For each basic block:

   Do local allocation, ensuring l-stacks match x-stack.

5.3 Determining x-stack

There are two challenges when determining the x-stack. One is correctness, that is, the x-stack must allow register allocation in the l-stacks to be both consistent with the x-stack and legal. The other challenge is the quality of the generated code. For example making all the x-stack empty is guaranteed to be correct, but not to give good code. Both the x-stack finding methods work by first using heuristics to find an x-stack which should give good code, then correcting the x-stack, if necessary. The algorithm for ensuring correctness is the same, regardless of heuristic used.

For the x-stack to be correct, two things need to be ensured:

1. **Reachability**

   Ensure all variables in the x-stack that are defined or used in successor or predecessor blocks, are accessible at this point.

2. **Cross block matching**

   Ensure that all unreachable variables in the x-stack on one edge do not differ from those in the x-stack on an other edge adjoining the same block.

5.3.1 Ordering of variables.

As stated earlier, a globally fixed ordering of variables is used. This is done by placing variables with higher 'estimated dynamic reference count' nearer the top of the stack. In our implementation, which is part of a port of lcc[3], the 'estimated dynamic reference count' is the number of static references to a variable, multiplying those in loops by 10 and dividing those in branches by the number of branches that could be taken. An alternative ordering could be based around 'density' of use, which would take into account the lifetime of variables. Profiling would provide the best estimate, but is impractical.

5.3.2 Heuristics

We use two different heuristics to demonstrate the utility of the framework. The first is simple and fast, whereas the second is more complex, and consequently slower.

5.3.3 Global 1

The first simpler heuristic is simply to take the *union* of live values. Its main flaw is that it selects variables for the x-stack, that cannot be allocated to the l-stack, and have to be spilled to memory.

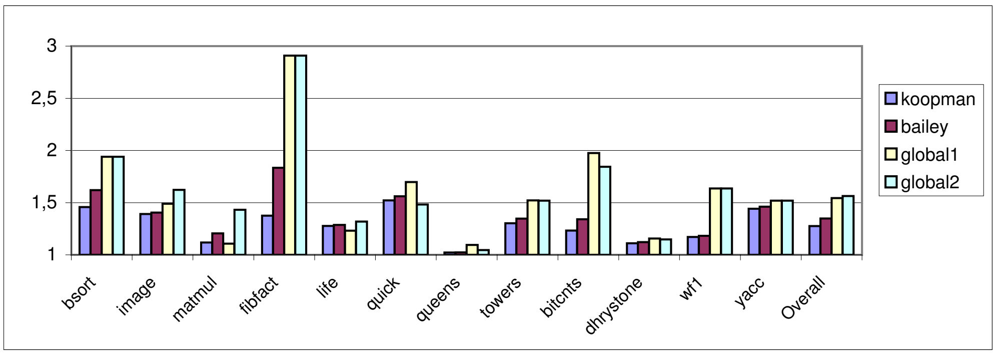
5.3.4 Global 2

This heuristic was developed to improve on 'Global 1'. It considers the ideal l-stack for each block and then attempts to match x-stack as closely to that as possible. Given that the ordering of variables is pre-determined, the x-stack can be treated as a set. In order to find this set, we determine a set of variables which would be counter productive to allocate to the l-stacks. The x-stack is then chosen as the union of live values less this set of rejected values. The set of 'rejects' is found by doing 'mock' allocation to the l-stack, to see which values can be allocated, then propagating the values to neighbouring blocks in order to reduce local variation in the x-stack. Overall this algorithm out performs 'Global 1', but can produce worse code for a few programs.

6 Results

The graph in figure 8 shows the simulated performance of the various register allocation methods, for a simple processor where memory accesses take three cycles and other operations take one cycle. The 'overall' result is the geometric mean of the other results. Although the results are for simple benchmarks on a simulated stack machine, we believe that the differences between the previous algorithms and the new ones are large enough to be significant.

Figure 8: Relative performance



7 Conclusion

As can be seen the global register allocation methods are generally better than the previous methods, but there is room for improvement. The framework laid out in this paper, enables us to analyse the two approaches, to see what those improvements could be, and can be used to find even better algorithms. Work is currently underway to find an allocator that performs at least as well as the two global allocators in all circumstances.

References

[1] C. Bailey. Inter-boundary scheduling of stack operands: A preliminary study. *Procedings of EuroForth 2000*, pages 3–11, 2000.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison Wesley, 2000.

[3] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley, 1995.

[4] P. Koopman, Jr. A preliminary exploration of optimized stack code generation. *Journal of Forth Application and Research*, 6(3):241–251, 1994.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[6] M. Maierhofer and M. A. Ertl. Local stack allocation. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 189–203, London, UK, 1998. Springer-Verlag.

[7] R. M. Stallman. *Using and Porting the GNU Compiler Collection, For GCC Version 2.95.* Free Software Foundation, Inc., pub-FSF:adr, 1999.