The Journal of Functional and Logic Programming

The MIT Press

Volume 1998, Article 7
10 December, 1998

ISSN 1080–5230. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493, USA; (617)253-2889; *journals-orders@mit.edu, journals-info @mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/JFLP/*
- *gopher.mit.edu*
- *ftp://mitpress.mit.edu/pub/JFLP*

The Journal of Functional and Logic Programming is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.

Improved Register Usage for Functional Programs through Multiple Function Versions

M. Satpathy A. Sanyal G. Venkatesh

10 December, 1998

Abstract

To use registers effectively, functional programs rely on interprocedural register allocation. Existing interprocedural strategies adopt a naive approach in the presence of recursion, and spill registers whenever necessary. Moreover, such recursion-induced spills cannot be avoided, even by increasing the supply of registers. In this paper, we describe a strategy that reduces memory spills due to recursion by keeping multiple versions of the same function. Each version gets a different register assignment and has different spilling characteristics. Such a strategy shows better spilling behavior as compared to the original (single version) program, but the extent of gain is largely dependent on the control paths followed by the program during execution.

We first determine the number of versions of each function, so that regardless of the execution path, the program with multiple function versions is guaranteed to perform better than the original program. Since some of these versions may be useless in the sense that they may never be called during any course of execution, we also have the problem of determining the number of meaningful versions. We solve this problem by casting it in terms of voltage graphs. We then show that by using properties of voltage graphs, we can reduce the number of versions even further without adding to the number of spills.

1 Introduction

Compilers for imperative programs rely primarily on intraprocedural register allocation. Such strategies consider one procedure at a time for allocation, with a call convention to handle the interface between procedures at the call/return boundaries. This simple treatment seems to be satisfactory for imperative programs. Since compilation is done on a per-procedure basis, register allocation can be done as part of the regular compilation phase, and does not have to be treated in a separate pass. Moreover, the number of variables in a procedure matches the number of registers (8–16) in many architectures.

Register-allocation algorithms for imperative programs usually consist of a local allocation phase in which registers are allocated to variables local to a basic block, and a global allocation phase in which registers are allocated to variables that are live across several basic blocks inside a procedure body. There are sophisticated code-generation algorithms [AJ76, AJU77] for doing local register allocation, and then there are good global register-allocation algorithms like the graph-coloring [Cha82] and TN-binding [WJW75] algorithms. Such global allocation algorithms work well because (1) there are large stretches of code between function calls which result in large *live ranges* of variables, and (2) the number of variables in a procedure body is reasonably large.

Intraprocedural allocation does not go well with functional languages. The reason is that a function body is usually small, and so is the number of variables in it. Moreover, a function body mostly involves function calls, and hence hardly produces good stretches of code where global register allocation might be applied. The experiments conducted by Steenkiste and Hennessy [SH89] on Lisp benchmark programs corroborate these facts. They found that on average, only 11 instructions are executed between 2 consecutive function calls or returns. In addition, the most heavily used control construct is recursion, which further aggravates the problem. Therefore, one has to rely more on *interprocedural allocation* with provisions to handle recursion.

1.1 Related Work on Interprocedural Register Allocation

Existing interprocedural register-allocation techniques can be classified into two categories. In *program-wide allocation*, register allocation is taken out of the regular compilation phase and done on all the variables of the program in a separate phase. This approach is taken by Wall [Wal86], and by Santhanam and Odnert [SO90]. In an *incremental strategy*, register allocation is done as a part of the procedure-by-procedure compilation process, and interprocedural information is used to optimize the register save/restore operations at individual call sites. This approach is taken by Steenkiste and Hennessy [SH89], Chow [Cho88], and Appel [AS92].

Wall's program-wide allocator first groups variables that cannot be live concurrently into pseudoregisters. This property exists between variables in sibling procedures in a call graph. Each global variable is also assigned its own pseudoregister. The usage frequencies of local and global variables are then used to determine the global usage frequency for each pseudoregister. The pseudoregisters with the largest global usage counts are assigned to registers, while others are allocated to memory. The code is then changed at link time to reflect the result of the allocation. To handle recursion, the same method is applied after collapsing each strongly connected component (SCC) to a single node, so that the call graph is reduced to a directed acyclic graph (DAG). After register allocation, the compiler inserts save and restore instructions around the backward edges for all the registers that are in use in an SCC.

Chow and Steenkiste and Hennessy use methods that work bottom-up on a call graph (a DAG in the absence of recursion). Chow's allocator initially divides the register set into a *caller-saved set* and a *callee-saved set*. The basic optimization is that the caller-saved registers that are not used by callees do not have to be saved and restored by the parent procedure. The compiler allocates variables to such registers as long as they are available, and then starts using callee-saved registers in caller-save mode. However, to save caller-saved registers for the upper regions of the call graph, a register is used for a variable in caller-save mode only if the variable is used in all execution paths in a procedure. Otherwise, the register is used in callee-save mode. In the presence of recursion, the allocator switches back to the initial division by spilling all callee-saved registers being used in caller-saved mode.

Steenkiste and Hennessy's method works in the context of Lisp programs.

It allocates the same set of registers to sibling functions, and attempts to give a parent function a set of registers different from any of its children. If the allocator runs out of registers while doing a bottom-up allocation, as it might in the upper regions of the call graph, it switches to a plain strategy. In this strategy, a function spills whatever registers it uses before making a call, and restores them once the call returns. To handle recursion, each SCC is replaced by a single compound node and each function in an SCC uses registers from a common set of registers allocated to the SCC. Within an SCC, a called function saves all the registers allocated to the calling function. Moreover, functions outside an SCC do not use any of the registers allocated to the SCC.

While Steenkiste and Hennessy's method assumes that it is always beneficial to save spills in the lower regions of the call graph, the method due to Santhanam and Odnert tries to actually identify such regions [SO90]. They call such a region a *cluster*. Informally, a cluster is a collection of nodes in the call graph such that (1) if registers are spilled at the root of the cluster, then calls corresponding to the interior nodes of the cluster do not need to spill; and (2) interior nodes are called more frequently than the root. Clearly, under such conditions it would be beneficial to save spills within a cluster. Their other concern is to share registers among global variables. This is in contrast to Wall's method [Wal86], where a register is allocated to a global variable for the entire program. This approach is very similar to the approach by Murtagh [Mur91] for allocating frames to procedures in block-structured languages. Murtagh identifies clusters in a call graph, and whenever the root of a cluster is called, he allocates space required by all the members of the cluster. Therefore, no frame allocation needs to be done for a procedure that is not the root of any cluster. As a consequence, such procedures have much shorter calling sequences.

In *continuation-passing style* (CPS) -based compilers, the context of a calling function is saved in a *continuation*, which is passed as an argument to the function being called [App92]. In implementation terms, the continuation is a *closure* containing a pointer to the code sequence corresponding to the continuation, and the bindings of its free variables. Appel and Shao [AS92] have adapted the ideas of caller-saved and callee-saved conventions of register allocation to CPS. Their essential idea is to pass the values of the free variables of the closure in registers during a call. If the called function does not use these registers for other purposes, then these values would continue to remain in registers during the execution of the function, and would not

have to be spilled. It is easy to see that this assumption does not hold in the case of recursive functions, and therefore the registers containing these values would have to be saved and restored around each recursive call.

Another approach to register allocation for functional programs is *inlining* or *unfolding*. In a function definition, function calls are unfolded, perhaps repeatedly, so that they are either eliminated (in the case of nonrecursive functions), or are replaced by larger stretches of straight-line code. Functions are usually unfolded under several optimization criteria to prevent unmanageable growth in code size. That is why unfolding of a recursive call is usually prohibited. For instance, the subprogram inliner developed by Davidson and Holler [DH92] has such a restriction. However, if program-analysis techniques like *control-flow analysis* or *binding-time analysis* reveal that the arguments of a recursive call are partially known, then the call can be unfolded. The unfolding is done only if the unfolded instance of the call satisfies certain size constraints. The partial evaluator *mix* follows such an approach [Ses87]; Jagannathan and Wright also describe such an approach in [JW96].

The inliners considered so far do not depend on any profile information. The inliner developed by Chang and colleagues [CMCH92] takes profile information into account to inline C programs. The profile information is used to construct a *weighted call graph* in which weights at nodes indicate the number of times the function is called by its callers, and weights along arcs indicate the number of times a call is invoked during execution. Here also inlining is done under code size constraints. If there are cycles in the call graph, then all the functions in the cycle except one are inlined. If the cycle involves a single function, then more than one copy of the function may be created inside the function body.

Tail-recursion elimination is another program-transformation technique in which a tail-recursive call is compiled as an iterative loop. In the context of functional programs, this means that the tail-recursive call uses the same space for its own activation record as its caller [PJ87]. While we are not directly concerned with this issue, our method at least ensures that the result of a tail call need not be spilled.

1.2 Our Work

To summarize, existing interprocedural register-allocation strategies do not work well for highly recursive programs. Steenkiste and Hennessy [SH89] ob-

serve that their interprocedural allocation removes 70% of the stack accesses that remain after intraprocedural allocation. Further, of the remaining stack accesses, on average, 85% are due to recursion. Even if an unlimited number of registers were available, such spills due to recursion cannot be avoided. To see the reason for this, consider a recursive function F, calling itself either directly or through mutual recursion. At the point of the call to F, some registers will be occupied. These registers have to be saved in memory so that they can be reused in the next invocation of F. That is why such registers were spilled along backward edges in an SCC in Wall's strategy, and in Steenkiste and Hennessy's method, a function spilled all the registers it occupied before making a recursive call.

Since recursion is integral to functional programming, the problem of reducing memory spills due to recursion deserves some attention. In [Sat96], we have a method that reduces memory spills due to recursion monotonically by using an increasing number of registers. But, as a consequence, the size of the code increases exponentially. In this paper, we describe a form of interprocedural register-allocation strategy that reduces memory spills due to recursion by keeping multiple versions of the same function, incurring linear code growth in the process. However, we are not able to guarantee that the number of spills will keep on decreasing monotonically with an increase in the number of registers. Instead, we show that, by having multiple versions, the performance will at least be as good as the single version case, and on the average better. The actual performance will largely depend on the runtime behavior of the program.

It is important to note that existing methods use the results of interprocedural analysis for nonrecursive calls only. Recursive calls are treated as exceptions and are handled by spilling wherever necessary. Our method does handle recursive procedures in a nontrivial fashion, using a simple interprocedural analysis.

The organization of the paper is as follows. Section 2 extends the Aho-Johnsson-Ullman algorithm [AJU77] to allocate registers for functional programs in a naive manner. Section 3 describes how we can reduce spills due to recursion by keeping multiple function versions. Section 4 discusses how *voltage graphs* and some results from number theory can be used to decide the number of meaningful versions of functions, and how certain space optimization can be done by using properties of voltage graphs. Section 5 concludes the paper.

2 Basic Register Allocation for Functional Programs

Our interprocedural register-allocation algorithm is based on a modification of the intraprocedural register-allocation algorithm by Aho, Johnson, and Ullman [AJU77]. We hereafter refer to this algorithm as the AJU algorithm. This algorithm performs register allocation and code generation for expression DAGs, whose interior nodes consist only of machine operators. In this section, we first describe the AJU algorithm, and then extend it to handle the body of a function definition. Note that we employ the AJU algorithm for its simplicity, and our interprocedural algorithm is not bound to it. As we discuss later, we could consider any other suitable algorithm as well in its place.

2.1 The Aho-Johnson-Ullman Algorithm

The algorithm assumes a fixed set of registers $\{R_0, \ldots, R_k\}$. It traverses an expression DAG in a depth-first manner from the root, and annotates each node with a register in which the value of the node is to be produced. A node is marked as visited only if all its parents have been visited. The root node gets the register R_0, and for each interior node annotated with R_i, the left child is annotated with R_i while the right child is annotated with R_{i+1}.

Code is generated in the reverse order of the visit. Code is generated for interior nodes and for leaf nodes that are left children. No code is generated for a leaf node that is a right child. This is a consequence of the assumed machine model. Let R_i be the register allocated for a leaf node. If the node is a left child, the code generated will be one of the following:

$R_i \leftarrow m$ /* *load* from location m in memory*/
$R_i \leftarrow literal$

The code for an operator node will be one of the following:

$R_i \leftarrow R_i \; op \; R_j$
$R_i \leftarrow R_i \; op \; m$
$R_i \leftarrow R_i \; op \; literal$

If a leaf node is a left child of more than one node, a load instruction is issued for each of its uses. Similarly, if an internal node is a left child of

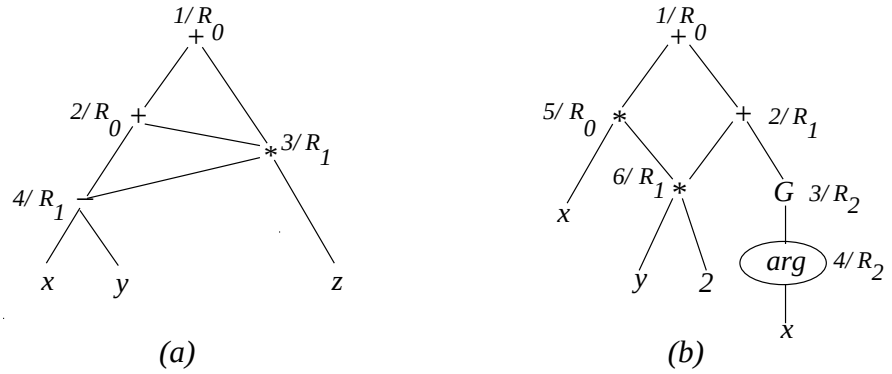

Figure 1: Using the AJU algorithm: order of visit and register allocation (a); traversal of a generalized DAG (b)

more than one node, its value has to be spilled into memory, following which it can be treated like a leaf node. Further, if an internal node gets the same register as its left parent during register assignment, it has to be spilled after evaluation. Spilling also occurs due to the unavailability of registers [AJU77].

The code generated for the example in Figure 1a is:

$R_1 \leftarrow x$
$R_1 \leftarrow R_1 - y$
$m \leftarrow R_1$ /* m is some location in memory */
$R_1 \leftarrow R_1 * z$
$R_0 \leftarrow m$
$R_0 \leftarrow R_0 + R_1$
$R_0 \leftarrow R_0 + R_1$

2.2 Extending the AJU Algorithm for Functional Programs

Before extending the AJU algorithm, we shall clearly state the problem framework. The language in consideration is an *eager first-order* functional language. Parameter passing is done through stack. Later, we discuss the possibility of passing parameters through registers and removing the first-order restriction. Since our main emphasis is on recursion, we assume that in general, no register will remain unaffected across a function call. Finally, we assume that all functions return their results in registers.

The unit of compilation is the body of a function, which can be represented as a DAG with new node types to represent if-then-else and *function calls*. For if-then-else nodes, the condition, the then, and the else branches can produce their results in the same register, which will also be the register for the final result of the if-then-else. We ignore any sharing between the then and else branches of an if-then-else expression. However, each such branch can share with the condition branch. The condition branch first evaluates its result in the desired register, and based on its contents, the then or else branch is selected for evaluation.

For a function f of arity k, let $D_1, \ldots, D_k$ be the DAGs representing the actual arguments of f, which may share nodes. These will be evaluated in registers and passed to f through the stack. The code for f will load its arguments from the stack and evaluate its result in a preassigned register, say R_m. Each of the DAGs $D_1, \ldots, D_k$ can produce its results in the same register R_m. This is because each argument, after evaluation, will be pushed to a stack, thereby freeing R_m. A function call creates a stack frame that stores the arguments, and in addition has space to store those nodes in its body for which spills may be necessary. Evaluating a call to f then amounts to evaluating the DAG for f, assuming that its arguments are in its stack frame.

2.3 Evaluating Function Calls

Extending the order of evaluation prescribed by the AJU algorithm to function calls may lead to an undesirable situation. During a call, some of the registers could be holding values of intermediate computations in the body of the caller. If the callee also needs these registers, then they have to be spilled. As an example, consider the function definition:

$$F\ x\ y\ =\ (x\ *\ (y\ *\ 2))\ +\ ((y\ *\ 2) + G\ x)$$

Figure 1b shows the DAG, the traversal, and register allocation under the AJU algorithm. It is easy to see that when G is called, the registers R_0 and R_1, containing the values of $(y\ *\ 2)$ and $(x * (y * 2))$, are live. If G uses these registers, then they have to be saved before the call to G, and restored later. If F calls itself many times due to mutual recursion through G, then the number of spills could be very high. But are such spills really necessary?

Consider another approach. First evaluate G, assuming all the registers

are available to it, and spill its result. Then the DAG of F is a simple expression DAG (all the intermediate nodes are machine operators and the leaves are in memory), and the basic AJU algorithm can be directly applied to it. Note that we have replaced multiple spills due to the call to G by a single spill. This forms the basis of our intraprocedural register-allocation scheme. The key idea is that we want to partition the DAG into regions such that each region can be independently allocated registers from a given register set. The boundaries between these regions form the points at which spills take place.

2.4 Partitioning the DAG

We first partition the DAG into its *linear parts*. The linear parts are maximal regions of the DAG that can be evaluated using the basic AJU algorithm. A node in a DAG is termed *nonlinear* if it is a function-call node or an if-then-else node; otherwise, it is called *linear*. The only nonlinear nodes in a linear part would be at its leaves. The idea is that before evaluation of a linear part, the nonlinear nodes at its leaves would already have been evaluated, and would reside in memory or in registers.

The algorithm for generating the partition is given in Figure 2. Essentially, the algorithm puts nodes in the three branches of if-then-else into different linear parts. It also ensures that the linear part is terminated at function and if-then-else nodes. The variable $nNumber$ holds the current node number, and $pNumber$ holds the number of the current linear part (the current partition number). We use list $L(pNumber)$ to collect all the function and if-then-else nodes encountered during a depth-first-like traversal of the linear part identified by the partition $pNumber$. Since the partitioning algorithm travels each node of a DAG exactly once, its complexity is linear in the number of nodes of the DAG.

Figure 3 shows an example DAG, and its partition into linear parts is shown as dotted regions. Each partition (from 0 to 6) corresponds to a linear part. It can be seen how nonlinear nodes do serve as interfaces between the linear parts. Linear parts specified by partitions 2 and 3 are empty.

2.5 Code Generation

A naive strategy for evaluating a linear part could be as follows: first, evaluate all the nonlinear nodes lying at the leaves of the linear part one by one, and

```
/* DFS does a depth-first traversal on partition pNumber and stores all nonlinear
nodes (nl nodes) at the leaves of the partition in list L(pNumber) */

Procedure DFS (node, pNumber)
  if (node.number > 0) or (for some parent p of node, p.number = 0)
  return;          /* Return if node already marked or all parents not marked */
  nNumber := nNumber + 1;
  node.number := nNumber;                                    /* mark node */
  node.pnumber := pNumber;
  case (node)
    {function: insert node in L(pNumber);                    /*store nl-node */
     if-then-else: insert node in L(pNumber);                /* store nl-node */
     operator: for each child n of node do DFS(n, pNumber);
     leaf: return; }
end-procedure
Procedure PARTITION(root)
  for all nodes n do {n.number := 0; n.pnumber := 0;}
  nNumber := 0; DFS(root, 0);          /* linear part at root is partition 0 */
  pold := 0; pnew := 0;                /* pnew is current partition number */
  for p := pold to pnew do
    {if (L(p) is nonempty)             /* nl nodes are present in partition p */
       pnew := p + 1;                          /* get new partition number */
       for each function node f in L(p) do
         for each child n of f do
           {DFS(n, pnew);                      /* each argument of a function */
            pnew := pnew + 1;}                 /* will be in a distinct partition */
       for each if-then-else node n in L(p) do
         {let c, t, and e be the cond, then and else branch of n;
            pnew := pnew + 1; DFS(t, pnew);          /* each branch of if */
            pnew := pnew + 1; DFS(e, pnew);          /* will be in a different */
            pnew := pnew + 1; DFS(c, pnew);}         /* partition */
    end-if}
end-procedure
```

Figure 2: The partitioning algorithm

spill their results; then, evaluate the rest of the linear part using the AJU algorithm, assuming that the results of all the nonlinear nodes at the leaves are in memory. Evaluation starts with the linear part at the root of the DAG

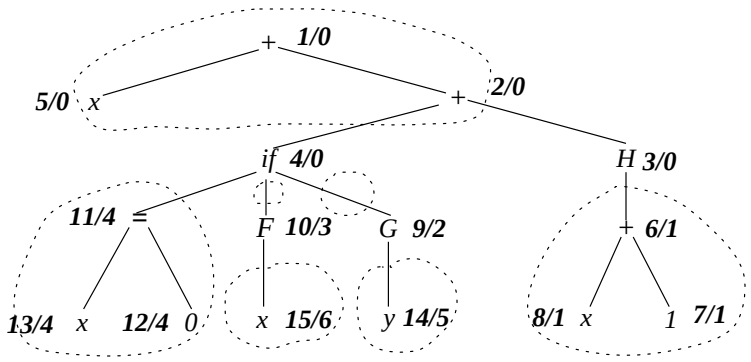

Figure 3: Numbers at each node show traversal and partition numbers, respectively

(see Figure 3), which in turn evaluates the rest of the DAG.

We now discuss the convention for spilling registers. All if-then-else and function nodes will spill their results. Besides this, some shared nodes also result in spills. The decision to spill a shared node whose parents all lie in the same linear part is dictated by the basic AJU algorithm. Otherwise, the node must necessarily be spilled.

The register-allocation and code-generation schemes described above form a starting point for handling functional programs. The bottom-up strategy discussed reduces the number of spills at the function-call boundaries. However, the scheme can only exploit as many registers as are needed to complete the allocation of the function body. For the example in Figure 3, only two registers are needed, and the number of spills will not reduce if the number of registers is increased beyond two. In the next section, we show how we can exploit additional registers to further reduce spills.

As mentioned earlier, our strategy is not bound to the AJU algorithm. For evaluating a linear part, we first want the results of nonlinear nodes at its leaves to be in memory. Once this is done, and given that a certain number of registers are available, we evaluate the rest of the linear part by using the AJU algorithm. However, we could use some other suitable algorithm as well; for instance, we could use one of the algorithms by Goodman and Hsu [GH88] to evaluate a linear part. But, since these algorithms are for pipelined processors, we must use the number of machine cycles as the metric, rather than the number of instructions.

3 Reducing Memory Spills through Multiple-Function Versions

We now present a scheme that can use extra registers profitably. To achieve this, we keep multiple versions of the same function, and give a different register assignment to each version. This will result in a reduction in spills as compared to the original program.

The main idea is that we evaluate a linear part by first evaluating the nonlinear nodes at its leaves in the registers, and *try to keep the values in the registers, instead of spilling them.* The interior of the linear part is then evaluated by using the basic AJU algorithm. Given a set of available registers N_t, we partition this set into the subsets N_l and $N_t - N_l$. We call N_l the set of *linear registers*, and $N_t - N_l$ the set of *nonlinear registers*. The set N_l contains enough registers to evaluate the interior of any of the linear parts of the program. The other set is used to hold values of nonlinear nodes, and it is this set that is of interest to us. We call this set M, and address its members as $R_0, \ldots, R_{M-1}$.

3.1 Single and Multiple Versions

Since the nonlinear nodes lying at the leaf level of a linear part can be evaluated in any order, we evaluate them from left to right. If the root of the linear part is evaluated in R_i, the nonlinear nodes at its leaves are allocated the registers from R_i onward to hold their returned values. In the process, if we cross the register with the maximum index, then, for the remaining nonlinear nodes, allocation continues from R_0 onward. Consider the example:

$$F\ x\ =\ \mathsf{if}\ (x \leq 3)\ \mathsf{then}\ 1\ \mathsf{else}\ F(x-1)\ +\ F(x-2)\ +\ F(x-3)$$

Its DAG is shown in Figure 4. We assume there are three registers, and that the result of the DAG is required in R_0. Then, the linear part at the else branch will be evaluated in R_0, and following the idea of keeping returned values in registers, the three calls to F are allocated registers R_0, R_1, and R_2 to hold their results. With a single version of each function, the only way to do this is to evaluate the first two calls from the left, spill their results, evaluate the third call, and then load the return values to the assigned registers. We call the resulting method a *single-version strategy* (SVS). In

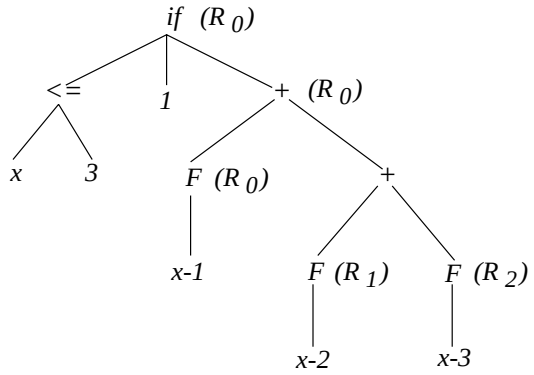

Figure 4: The initial DAG of function F

the *multiple-version strategy* (MVS) that we are about to describe, many of these spills are eliminated by keeping multiple versions of the same function.

Here we describe the MVS. For the present, we assume that the number of function versions is the same as the number of registers, M. Some of these versions may be redundant, and later we discuss ways of removing them. The initial version of F is designated F_0. Version F_i returns its result in R_i. Further, in the DAG of F_0, if a nonlinear node was a call to a function G_j returning its result in register R_j, then in the DAG of F_i, it will be a call to version $G_{(j+i) \bmod M}$, which will return its result in $R_{(j+i) \bmod M}$. The label of the register in which an if-then-else node is evaluated is similarly modified. We call this *circular invocation*.

3.2 Naive Spilling Decisions

A function version may be required to spill. Under circular invocation, a version may appear in different contexts. But we adopt a certain convention that allows us to determine the spills of each version statically. Our allocation is such that the following assumption (I_1) is always satisfied: a call to F_i can assume that only registers R_0 to R_{i-1} are live.

So F_i, in turn, spills enough registers that the assumption holds for calls inside its own body. Let there be four registers ($M = 4$), and consequently four versions of F, named F_0, F_1, F_2, and F_3. A portion of the *run-time call tree* (RTCT) for a call to F_0 is shown in Figure 5. Program control takes the else branch of each of the interior calls in the tree. In each such else branch, only the three calls to F are shown. Calls to F_0 and F_1 will not spill,

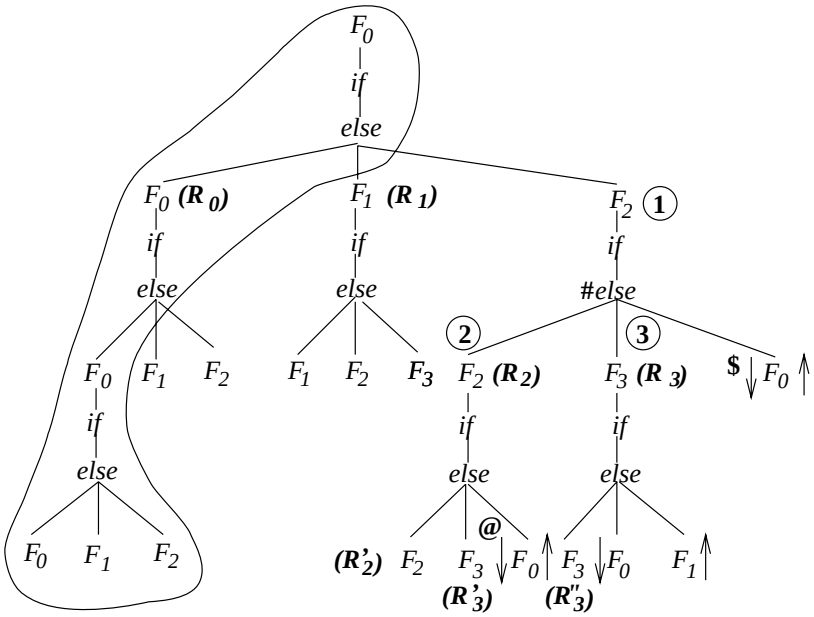

Figure 5: The RTCT for a call to F_0: downward arrows and upward arrows show places of spill and restoration; condition and then branches of if nodes are not shown

because assumption I_1 holds for the calls inside them even without spilling. However, a call to F_2, taking the else branch, will spill all the occupied registers (R_0 till R_3) before the inner call to F_0. Similarly, F_3 will spill all the occupied registers before the call to F_0, so that assumption I_1 holds for F_0. In summary, each of the calls, F_2 and F_3, will spill four times (equal to M, the number of registers) when execution takes their else branches.

If calls to F_2 and F_3 occur less often in a RTCT, then the total number of spills is reduced drastically. For instance, if execution always takes the leftmost branch (shown as the enclosed region in Figure 5), then no spill occurs. On the other hand, if execution always takes the rightmost branch, then quite a few spills are expected. However, our method ensures that even in the worst possible scenario, the number of spills remains less than the spills for the same program with a single version for each function.

3.3 Redundant Spills

The strategy just described results in many redundant spills. In Figure 5, we specially marked two calls to F_2 and a call to F_3. We refer to the two calls to F_2 as parent F_2 and child F_2. The call to parent F_2 has to spill the occupied register set $\{R_0, R_1, R_2, R_3\}$ just before the call to F_0 in its body (at position \$ in the figure). The calls whose results are held by these registers are also shown in the figure; the positions where spills take place are indicated by downward arrows. Similarly, child F_2 has to spill $\{R_0, R_1, R'_2, R'_3\}$ (at position @). R_i, R'_i, and R''_i represent the same register holding the results of different calls at different instants.

Both the marked calls to F_2 spill the register subset $\{R_0, R_1\}$ containing the same values. Observe that such values are live before the call to parent F_2, and are not used until the same call to F_2 returns. So the spilling of the above register subset could be hoisted up to the position marked #; i.e., parent F_2 will spill them as soon as control takes its else branch, and restore them after the call returns. Then child F_2 can assume that all registers are free, and need not spill these registers. Since we made the spilling behavior of the two calls to F_2 different, they have different code: this is why we give child F_2 a $*$ superscript.

For a $*$-superscripted call, assumption (I_2) holds: a $*$-superscripted call can assume that all the registers are free.

If we follow the convention that the result of a $*$-superscripted call is spilled, then F_3 at position 3 can also be given a $*$ superscript. Further, notice that by doing so the spilling obligations of parent F_2 ($\{R_0, R_1, R_2, R_3\}$) are met. Similarly, the spilling obligations of child F_2 (which are now $\{R'_2, R'_3\}$) are met by $*$ superscripting the calls to F_2 and F_3 under it. The complete spilling behavior is shown in Figure 6. Each call in the RTCT satisfies assumption I_1 or I_2, and meets its spilling obligations.

We now give rules for $*$ superscripts.

1. If F_j occurs in the body of F_i, and F_i and F_j both spill under the naive strategy, then F_j will receive a $*$ superscript. This ensures that F_j does not spill the registers already spilled by F_i.

2. If F_j occurs in the body of F_i^* and is a candidate for spilling under the naive strategy, it will receive a $*$ superscript.

3. Let n^* stand for an $*$-superscripted function call or if node (see below).

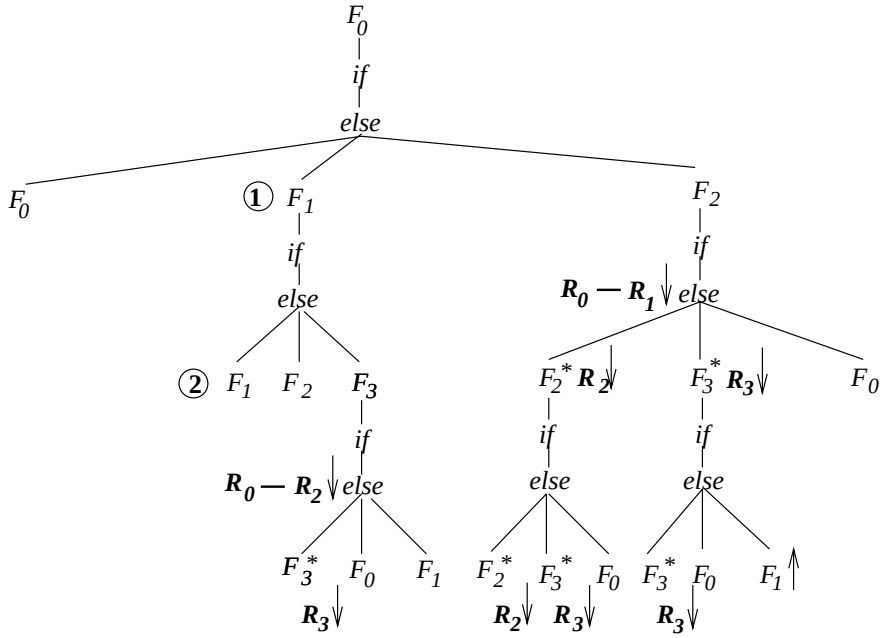

Figure 6: Complete spilling behavior for the example

If F_j and n^* occur at the leaf level of the same linear part and F_j is evaluated before n^*, then F_j receives an $*$ superscript.

Also observe that under our new scheme we need not spill the results of all if-then-else nodes. To distinguish the if-then-else nodes whose results need be spilled, we give them $*$ superscripts. The rules for doing so are:

4. If if_j and n^* occur at the leaf level of the same linear part and if_j is evaluated before n^*, then if_j receives an $*$ superscript.

5. If if_j is evaluated in the register with the maximum index R_{M-1} (i.e., $j = M-1$), and it is not the last in a sequence of nonlinear nodes lying at the leaf level of a linear part, it receives an $*$ superscript.

As an illustration of the rules above and the consequent spilling behavior, consider the example:

$$F\ x = (\ \mathsf{if}()\ \mathsf{then}\ ()\ \mathsf{else}\ (F() + F() +\ \mathsf{if}\ () + F())\ ) +\ \mathsf{if}\ () + G()$$
$$G\ x =\ \mathsf{if}\ ()\ \ \mathsf{then}\ ()\ \mathsf{else}\ (G() + G() + G())$$

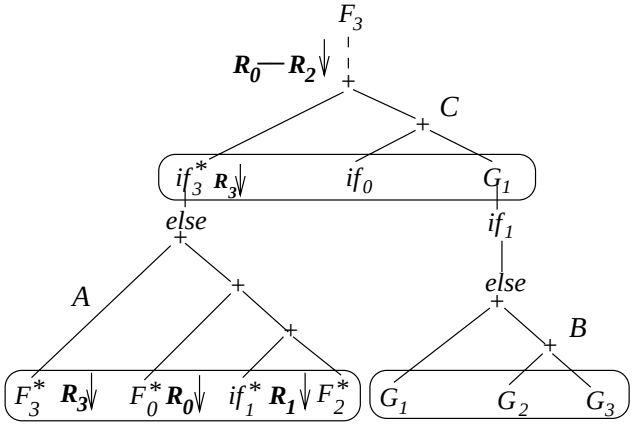

Figure 7: An example illustrating $*$-superscript rules

Parts of the functions not relevant to our discussion have been left blank within parentheses. Assume that there are four versions ($M = 4$) of each function. We show the F_3 and G_1 versions of the functions in Figure 7. Consider the sequence of nonlinear nodes $\{F_3, F_0, \mathsf{if}_1, F_2\}$ lying at the leaf level of linear part A. Calls F_3 and F_2 are $*$ superscripted owing to rule 1. This is because F_3, their caller, is a candidate for spilling. Moreover, F_0 and if_1 are $*$ superscripted because of rules 3 and 4. None of the $*$-superscript rules apply for the sequence $\{G_1, G_2, G_3\}$ lying at the leaf level of B. Finally, the node if_3 lying at the leaf level of C is $*$ superscripted because of rule 5. Notice that, unlike in the SVS, the result of if_0 does not have to be spilled.

There is one situation where the result of an $*$-superscripted call (or an $*$-superscripted if-then-else node) will not be spilled. This is illustrated by the call $F_2{}^*$ in Figure 7. Since it is the last nonlinear node lying at the leaf level of A, its evaluation can fall through the evaluation of the corresponding linear part.

For both the MVS and SVS, we now associate each spill with some function call. Since the number of function calls in both strategies remains the same, it is easier to compare the total number of spills. The simple rule for doing this is the following: with a call to a function F, we associate the spills inside its body. Since each spill occurs in the body of some function, the above rule accounts for all spills.

The method still has certain redundant spills. In Figure 6, assume that the two calls to F_1 marked 1 and 2 and the F_3s called by them all take their

else branches during execution. Then, both F_3s will spill registers R_0, R_1, and R_2, in which the spilling of R_0 is clearly redundant. To avoid such redundancy, we first have to detect the context of such call occurrences, and then introduce yet more versions of the same function. Therefore, we have decided to ignore such duplication in spills.

Redundant spills also occur due to sibling calls in the same function definition spilling the same registers. For example, in Figure 6, the calls F_2 and F_3, called within the body of F_1, will spill $\{R_0, R_1\}$ and $\{R_0, R_1, R_2\}$ when control takes their else branches. In both sets, registers R_0 and R_1 contain the same values. To avoid such spills, we have to precompute in the body of the caller F_1 whether the calls to F_2 and F_3 will take their else branches and, in such a situation, spill R_0 and R_1 there itself. Though it is feasible to do so, the approach is quite complex and may result in exponential growth in the code size [Sat96]. So we choose to ignore such spills also.

Note that each function has a number of versions equal to the number of registers used (the versions without superscripts) plus a number of $*$-superscripted versions. So far we have assumed that the number of registers used is M, the entire set of available registers. However, this may not always be profitable, since beyond a certain point, duplication in register spills will dominate, and the number of spills may even exceed the spills in the original program. We illustrate this by referring to Example 6 in Appendix A. Figure 8 depicts the complete RTCT for the call to F_0 (11) when we use five nonlinear registers (R_0–R_4). Under the single-version case, the number of spills is seven (two spills each by F_0, A_2, and A_3, and one by B_4). But observe that under the MVS, calls A_3 and B_4 will spill five registers each, out of which three registers will have the same values. The positions where these spills take place are also indicated in the figure.

Therefore, it might not be profitable to use the whole set of available registers, M, but a subset (say N registers) to decide on the number of versions. We will now describe an algorithm for choosing a value of N to guarantee that the number of spills under the MVS will always be less than the number under the SVS.

3.4 Critical Spilling Regions (CSRs)

Consider a program whose call graph is an SCC. Observe that the RTCT for any program is the same for both the MVS and SVS modulo version renaming. Under the MVS, spills take place only at some of the nonlinear

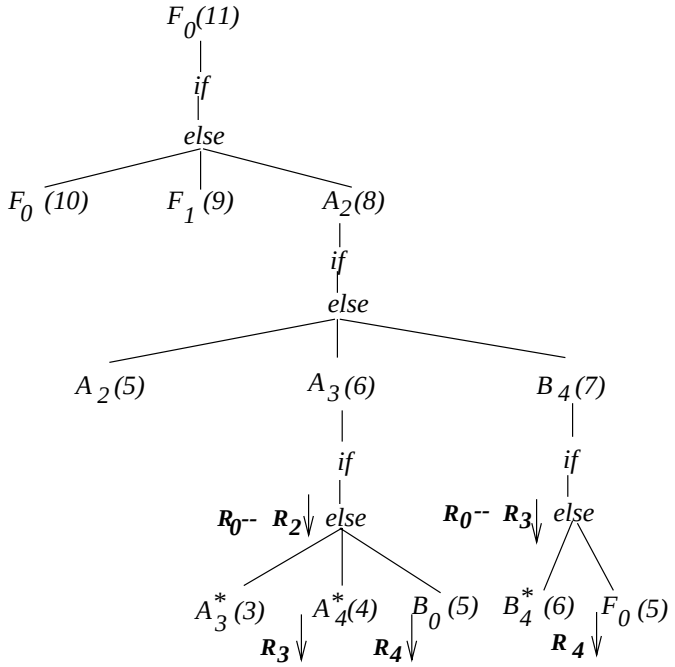

Figure 8: Duplication in spills dominating an RTCT; downward arrows show where spills take place under the MVS

nodes. Out of these nodes, only some have $*$ superscripts. As we show in the proof of Theorem 1, the $*$-superscripted nodes spill less than or an equal number of times compared to the corresponding nodes under the SVS. Therefore, around those calls in the RTCT, we demarcate regions that spill and are without $*$ superscripts. We call such a region a CSR. Now the number of versions N is chosen so that the number of spills within the CSRs never exceeds the number of spills in the corresponding regions of the RTCT under the SVS.

Formally, a CSR is a region in an RTCT that encloses adjacent function-call nodes according to the following rules:

1. Let F_i be a node in the RTCT with register index i and without an $*$ superscript. Make F_i the root of a CSR if F_i does not spill but at least one of its children spills.

2. Include in the same CSR those nodes with register index j, say G_j, such that:

i. $i < j$,

ii. F_i calls G_j, and

iii. G_j is not the root of another CSR.

We now discuss the construction of CSRs through the following example program:

$$F_0\ x\ =\ \mathsf{if}\ (x \leq 3)\ \mathsf{then}\ 1\ \mathsf{else}\ F_0(x-1) + F_1(x-2) + F_2(x-3)$$

Figure 9 shows a portion of an RTCT with a call to F_{N-5} at the root. Assume that program control takes the else branch of each of the interior call nodes in the figure. For constructing CSRs, observe that only nodes F_{N-4} and F_{N-3} satisfy rule i. We therefore have two types of CSRs: CSR_1 with F_{N-4} as the root, and CSR_2 with F_{N-3} as the root. Note that each F_{N-4} in an RTCT will be the root of a CSR of type CSR_1, and each F_{N-3} is made the root of a CSR of type CSR_2. Following rule *ii*, F_{N-2} is added to CSR_1 (see Figure 9), and both F_{N-2} and F_{N-1} are added to CSR_2.

Let us now justify the intuition behind the construction of CSRs by referring to CSR_2 in Figure 9. For constructing a CSR, we first identify a sequence of unsuperscripted sibling calls that do spill. In this case, they are F_{N-2} and F_{N-1}. To start, we include these calls in the CSR. Inside a CSR, we want the number of spills under an MVS to be smaller than under an SVS. Since under an MVS the number of spills in the CSR consisting only of F_{N-2} and F_{N-1} could be high, we offset these spills by including some unsuperscripted calls, such as F_{N-3}, in the same CSR. This in effect increases the number of versions for which an MVS would be more profitable than an SVS. Note that we cannot include F_{N-5}, because it does not appear in every context in which the sequences F_{N-2} and F_{N-1} appear. Further, we cannot include the sibling F_{N-3}, because it is the root of another CSR. So, nothing else can be added to CSR_2.

Theorem 1 *Consider a program whose call graph consists of a single SCC. If, for each CSR, the number of spills under an MVS does not exceed the number of spills under an SVS, then for any RTCT of the program, the total number of spills under an MVS will not exceed the number of spills under an SVS.*

Proof of Theorem 1 Apart from the CSRs, the $*$-superscripted nodes are the other places where there will be spills under an MVS. Clearly, the spills

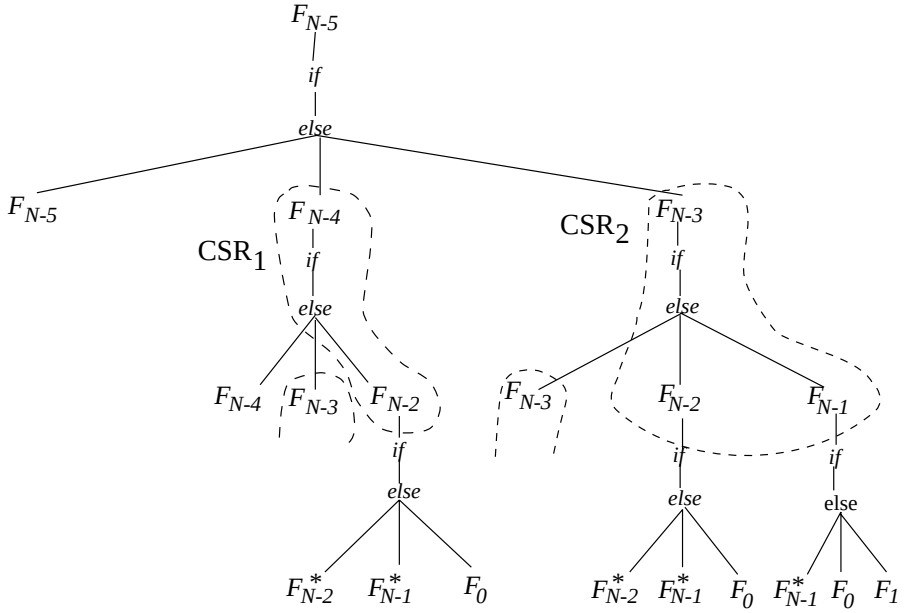

Figure 9: CSR_1 and CSR_2 are the two CSRs of function F

due to $*$-superscripted if-then-else nodes are matched by their counterparts under an SVS. Consider an $*$-superscripted call. Observe that the only spills associated with such a call are (1) spills of results of $*$-superscripted function calls in its body, and (2) spills of results of $*$-superscripted if-then-else nodes in its body. These are also matched by their counterparts under an SVS.

Proof of Theorem 1 □

To see how the above result can be used to find a profitable number of versions, consider the same function F once again. Let the number of registers for which the performance of an MVS is guaranteed to be better than an SVS be N. From the discussion above, in addition to the N versions, $F_0, \ldots, F_{N-1}$, there are two additional versions, (F^*_{N-2} and F^*_{N-1}). According to our spilling conventions, versions F_{N-2} and F_{N-1} will spill, and versions F_i for $i < N-2$ will not spill.

Consider the two CSRs, CSR_1 and CSR_2, of function F in Figure 9. Tables 1 and 2 compare the number of spills under MVS and SVS for these two CSRs. The rows show various cases, depending on whether F_{N-1} and F_{N-2} take their then or else branches. The idea is to analyze every possible

F_{N-2} takes Branch	Number of Spills in an SVS	Number of Spills in an MVS
then	2	0
else	4	N

Table 1: Number of Spills for CSR_1

F_{N-2} takes Branch	F_{N-1} takes Branch	Number of Spills in an SVS	Number of Spills in an MVS
then	then	2	0
then	else	4	N
else	then	4	N
else	else	6	$2N$

Table 2: Number of Spills for CSR_2

control path in a CSR, calculate the number of spills in terms of the variable N in both the cases, and obtain a constraint involving N. Analysis of each CSR gives rise to one such constraint, and the final value of N is obtained from all such constraints.

Analysis of Table 1 shows that for $N \leq 4$, spills under an MVS will not exceed that under an SVS in any CSR of the kind CSR_1. Table 2 shows a similar behavior for CSR_2 when $N \leq 3$. In conclusion, if $N \leq 3$, then the number of spills under an MVS will never exceed that under an SVS in either of the two CSRs. We can have up to three unsuperscripted versions of F. However, to ensure that the number of spills is less than the same in an SVS, we should have $N < 3$ (ignoring the trivial case when the number of spills in both cases is zero). In general, if CSR analysis yields a constraint $N \leq t$, it means that for $N < t$ the number of spills in any CSR under an MVS will be strictly less than the number of spills in the same region under an SVS. However, when $N = t$, then the number of spills in both the strategies could be the same. Because we gain, regardless, at other places in the RTCT, we recommend using t registers.

So far we have discussed how to bring down the number of spills in case of a self-recursive function like F above. We first identify the CSRs, and from their analysis, we find the value of N. The method of determining the value of N for a general SCC is analogous: we take each individual function in the

SCC and determine the CSRs for it. Each CSR gives rise to a constraint involving N; we determine the final value of N from this set of constraints. Since all the constraints are of the form $a * N < b$, we simply take $\lfloor b/a \rfloor$ for each such inequation, and choose the minimum value of N.

3.5 Handling General Call Graphs

We have considered call graphs consisting of a single SCC. We now consider call graphs which, in general, have more than one SCC, in addition to nonrecursive calls. For handling such a call graph, Steenkiste and Hennessy's method [SH89] involves replacing each SCC by a single compound node, and applying the bottom-up allocation to the resulting DAG. Our approach is similar, but with two important differences. First, our bottom-up allocator allocates registers from the set $N_t - N_l$ to the nodes in the DAG. The same set N_l is used to evaluate the linear parts of each node in the DAG. Second, the number of registers allocated to an SCC is the number N that we obtain for the SCC from the analysis discussed above. It is important to note that if an SCC is allocated registers R_i to R_j, then a function F in the SCC will have the versions F_i to F_j, with F_i playing the role of F_0 in the previous discussions.

Figure 10 shows a call graph with three SCCs. A and B are the two nonrecursive calls requiring one and two registers, respectively. The values of N obtained for the SCCs are as shown in the figure. The figure also shows the bottom-up allocation. Note that if a function in one SCC calls a function in another SCC, then no spill occurs at the call boundary. This is because disjoint nonlinear register sets are given to the two SCCs. SCC_1 is given registers R_5 through R_7, and its functions have versions indexed by $5, 6$, and 7. Let a function F in SCC_1 call a function G in SCC_2. This translates to a call to the version G_2, which, in SCC_2, plays the role of G_0.

In case of a shortage of registers, we adopt Steenkiste and Hennessy's strategy of spilling some registers, and continue with the bottom-up allocation.

3.6 Experimental Results

We now present the results of our experiments, which demonstrate the effectiveness of our method. Tables 3–7 in Appendix B tabulate the number of spills for Examples 1–5 in Appendix A. In each table, the column under the

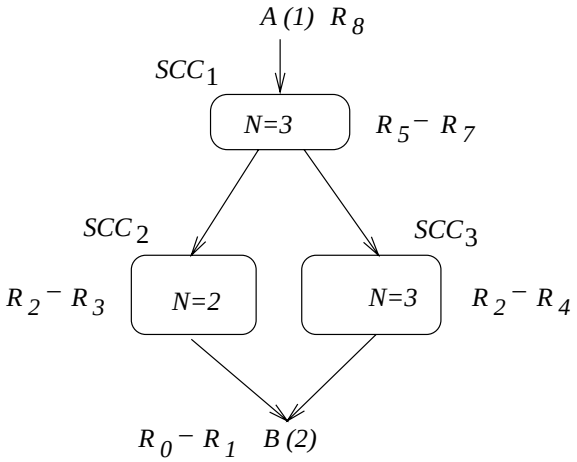

Figure 10: Bottom-up allocation for a generalized call graph

heading "Incremental Strategy" shows the number of spills when the program is evaluated following the approach by Steenkiste and Hennessy [SH89]. Since the choice of intraprocedural allocation algorithm in their method is unspecified, we used the AJU algorithm. Subsequent columns show the number of spills when the same program is evaluated under our bottom-up strategy, using single and multiple versions. The columns titled "r Registers" indicate that r nonlinear registers were used to decide on the number of versions. If A is a function in an SCC, and r is the number of nonlinear registers used, then the number of versions of A is r plus the number of $*$-superscripted versions for A. We divided the number of spills into two categories. The first category of spills is due to evaluation of nonlinear nodes, and the second to evaluation of the interior of linear parts. The second category of spills is the same for both the SVS and the MVS. For purposes of comparison, they have been ignored.

The function $S(k, n)$ in Example 1 is used in solving the *prefix problem* using Ladner and Fischer's method [LF80]. Example 2 is the *map* operation on binary trees. Since we are considering a first-order language, we take the first-order version of *map*. Example 3 is for transposing a square matrix when it is represented as a power list [Mis94]. Example 4 is an inefficient version of *quicksort*. The last example shows a case of mutual recursion. Note that the Fibonacci function falls in the category of Example 1. Many of the operations on binary trees and k-ary trees (for instance, unifying two k-ary

trees) fall in the category of Example 2. For k-ary trees, our CSR analysis gives larger values of N. Example 3 is like an operation on a 4-ary tree. For this example, CSR analysis finds the value of N to be 4. For *quicksort*, the value of N is 2, and for Example 4, the value of N is 5.

Figure 11 shows the graphical representations of the experimental results. Consider the graph for the *quicksort* example. For each case, we take a list of size 2,000. Notice that under each of the multiple-version cases, the program is executed without incurring any spills. This is because in the RTCT, the versions with higher indices (versions that spill) never carry control to those branches of the function bodies where spills take place. When the list is sorted in reverse order, we have the worst-case behavior. Here, under an incremental strategy, the number of spills is 4,000; the same under the single-version case is 2,000. Under the multiple-version strategy, we get marginally better behavior. The reason is that the RTCT is dominated by versions that spill. The figure in parentheses with each line shows the number of spills under the incremental strategy.

We mentioned earlier that the program behavior is mostly input dependent. Observe this from the graphs for various examples. Note that as long as the number of versions remains within N, we get better performance than with the single-version program.

3.7 Discussion

3.7.1 Effect on Code Growth

Keeping multiple versions results in a linear increase in code size that may affect the cache behavior. Studies conducted by Davidson and Holler [DH92] in the context of subprogram inlining reveal that code growth does not necessarily make the cache behavior worse. In their benchmark programs, though code size increased due to inlining by factors ranging from 3 to 8, they observed marginally better cache and paging behavior. Their reason is that in effect, a lesser number of instructions got executed while executing inlined programs. Moreover, inlining altered the locality pattern, which sometimes became better and sometimes became worse than the locality pattern of the original program.

We feel that our method does not result in degradation of cache behavior. The CSR analysis on various programs indicates that the number of function versions under the MVS will rarely go beyond a factor of 8. In the next

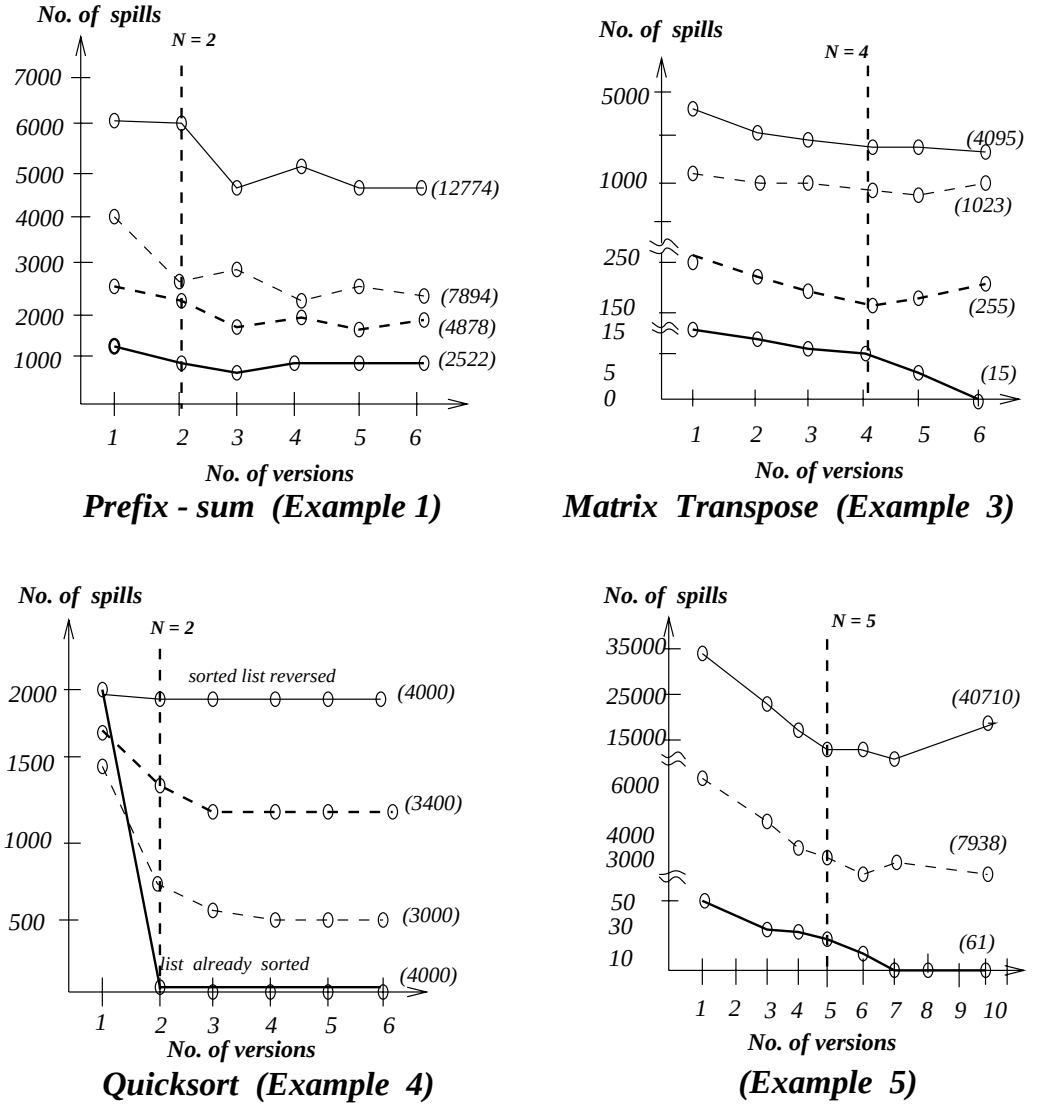

Figure 11: Graphical representation of the experimental results

section, we discuss ways of decreasing the number of versions even further without increasing spills. In addition, we can improve cache behavior by using cache prefetching techniques. When a function version is called, we know the versions likely to be called next; hence, they can be prefetched.

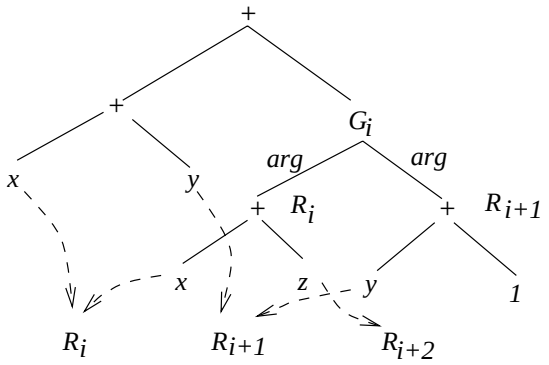

Figure 12: Passing parameters in registers

3.7.2 Passing Parameters in Registers

All along, we have assumed that parameter passing is via stack. When a function version F_i is called, registers from R_i onward are free; therefore, its parameters could be prepared in R_i, $R_{(i+1) \bmod N}, \ldots, R_{(i+k-1) \bmod N}$, where k is the arity of F. If $(i + k - 1)$ crosses the value of N, then there are several design choices. One strategy is to evaluate via stack those parameters whose register indices cross the value N. Once the parameters of F_i are in evaluated form, the DAG of F_i should be evaluated so that the result is available in R_i. We illustrate this through an example. Let Figure 12 represent the DAG of version F_i whose three arguments are available in R_i, R_{i+1}, and R_{i+2}. The call to G gets the register assignment i, and hence its two parameters are also to be prepared in R_i and R_{i+1}. The challenge is then to use the parameters of F_i in registers as much as possible while computing the parameter values of G_i. When it is no longer possible to retain the parameter values of F_i in registers, they have to be spilled into memory, and further references to them will be memory references.

Table 8 in Appendix B shows the number of register and memory operations for the Fibonacci function, when parameters are passed in registers. The table shows clearly that the improved performance of the MVS over the SVS is not affected even when arguments are passed in registers. We are currently investigating other design choices so that programs will execute with a minimal number of register and memory operations.

3.7.3 Complexity of the MVS

Our algorithm involves (1) finding the SCCs in a call graph, (2) doing CSR analysis, and (3) finding the number of meaningful versions. Algorithms for finding SCCs and obtaining a spanning tree in a graph (which are required by point 3, and are described in the next section) are of linear complexity [vL90]. For doing analysis on a CSR, we need to examine all control-flow paths inside the CSR. This can be exponential in the number of if-then-else nodes inside the CSR, but in practice, the number of such if-then-else nodes will be few and the number of control paths hardly exceeds the number 2^4 or 2^5. The remaining steps in our algorithm are of linear complexity.

4 Finding the Number of Meaningful Versions

Given an SCC, we have a method of determining N, the number of registers we can profitably use. Each function is assumed to have N versions in addition to some versions with $*$ superscripts. But some of the versions may not be *meaningful*, in the sense that they will never be called in the course of any execution. We want to avoid creating such versions to prevent unnecessary growth in code size. So, what we are interested in is the number of meaningful versions for each function. Further, we show that by manipulating the evaluation order (choosing an order other than left-to-right), it is possible to reduce the number of versions even further without introducing extra spills.

Figure 13 illustrates the notion of meaningful versions. In the figure, F and G are two functions that constitute an SCC. Versions F_0 and G_1 are as shown in the figure. Assume that the main program calls F only, and G, in turn, is called only by F or G itself. Also assume that to minimize the number of spills, we decide to keep four versions of each function. It is easy to see that under the circular invocation strategy, the versions F_1, F_3, G_0, and G_2 will never be called. So the only meaningful versions are F_0, F_2, G_1, and G_3.

We define the *distance* of a function call node, with respect to its caller, as the register number that it gets while assuming that the root of the associated DAG gets register zero (the caller returns its result in R_0). In Figure 4, the three calls to F are at distances 0, 1, and 2 with respect to the caller. We can determine the distance of each called function in a program by inspecting

Figure 13: Meaningful versions

the DAGs of the function bodies.

4.1 The Number of Meaningful Versions: Analysis

A distinct function version implies a distinct register assignment to it. Function version F_i is said to have register assignment i. To find the number of meaningful versions, we proceed as follows. First, we show how to determine the number of distinct register assignments for a single self-recursive function. This is done using Lemmas 1 and 2. Then, using voltage graphs, we show how to convert a general SCC to the case of an SCC consisting of a single self-recursive function, so that the number of register assignments to the function remains unchanged under the transformation.

We first handle the case of a function with a single recursive call to itself.

Lemma 1 *Let $N \geq 1$ be the number of registers. Let F be a recursive function; the only call to F lies at a distance d in the body of F. Then the number of distinct register assignments to F is $N/gcd(N, d)$.*

Proof of Lemma 1 Assume an initial assignment of 0 to F. Then S_1, the set of possible assignments to F, is given by $\{i * d \bmod N \mid i \geq 0\}$. This set is the same as $S_2 = \{(j * \gcd(N, d)) \bmod N \mid j \geq 0\}$. It is obvious that any element of S_1 is in S_2. Let us take an element $k = (j_0 * \gcd(N, d)) \bmod N$ in S_2. Then, from the properties of gcd [Mig92], $k = j_0(x\ N + y\ d) \bmod N$, where x and y are integers. From this we can find a non-negative value of i such that $k = i * d \bmod N$, and the number of elements in S_2 is $N/\gcd(N, d)$.

Proof of Lemma 1 □

This result can be generalized to the following lemma.

Lemma 2 *Let F be a recursive function with n calls to itself at distances $d_1, d_2, \ldots, d_n$. Then the number of distinct assignments to F is given by*

$N/gcd(N, d_1, \ldots, d_n)$, and the assignments (assuming an initial assignment of zero) can be generated from $\{k * gcd(N, d_1, \ldots, d_n) \mid k \geq 0\}$.

Proof of Lemma 2 Assuming an initial assignment of zero to F, the possible calls to F can lie at distances

$$\{(i_1 d_1 + \ldots + i_n d_n) \bmod N \mid i_1, \ldots, i_n \geq 0\}$$

where $i_1, \ldots, i_n$ are integers. This is the same as the set

$$\{k * \gcd(N, d_1, d_2, \ldots, d_n) \bmod N \mid k \geq 0\}$$

whose cardinality is $N/\gcd(N, d_1, \ldots, d_n)$. The equality of both sets can be proved as in the previous lemma.

Proof of Lemma 2 □

Lemma 3 *Every member of an SCC in a call graph receives the same number of assignments.*

Proof of Lemma 3 Consider any cycle in an SCC. Let the nodes $F_1, F_2, \ldots, F_m$ constituting the cycle ($m \geq 1$) have $k_1, k_2, \ldots, k_m$ number of assignments, respectively. Nodes F_i and F_{i+1} are the adjacent nodes for $i < m$, and F_1 is the adjacent node of F_m. Let F_1 call F_2 at distance d_{12}. Then, F_2 has a number of assignments that is at least as many as F_1; i.e., $k_1 \leq k_2$. This is so because the k_1 assignments to F_2 can simply be obtained by adding (modulo N) d_{12} to each of the assignments to F_1. Following the same argument, we have $k_1 \leq k_2 \leq k_3 \ldots k_{m-1} \leq k_m \leq k_1$, which implies $k_1 = k_2 = k_3 \ldots k_{m-1} = k_m$.

Proof of Lemma 3 □

Clearly, it is enough to concentrate on any one node of the SCC. For finding the number of assignments to such a reference node, we use voltage graphs [GT87].

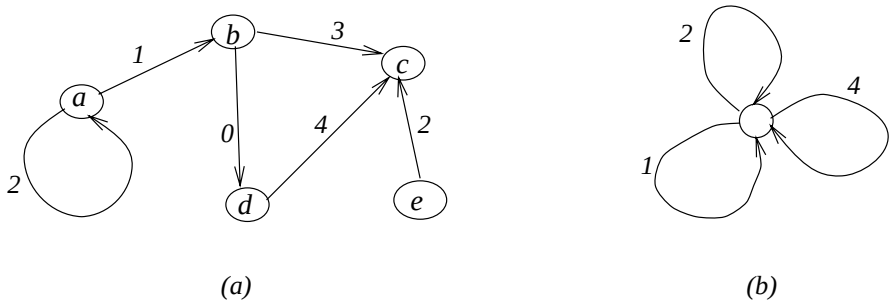

Figure 14: A voltage graph (a); and a bouquet of circles (b)

4.2 Voltage Graphs

Let G be a directed graph. Each edge in G has a plus and a minus direction (the direction of the directed edge and its reverse, respectively). Let A be a finite group. Let a function α map the *plus-directed* edges to the elements in A. Such a function is called the *voltage assignment* on G, and the pair $\langle G, \alpha \rangle$ is called a voltage graph [GT87]. The values of α are called the voltages, and A is called the *voltage group*. The original graph G is called the *base graph* of the voltage graph. If $\alpha(e^+) = v$, then $\alpha(e^-)$, the voltage associated with the *minus-directed edge*, is the *group inverse* of v. Figure 14a shows a voltage graph with $\langle \{0, 1, 2, 3, 4\}, +(mod\ 5) \rangle$ as the voltage group. Edge ab is a plus-directed edge, and ba is a minus-directed edge. $\alpha(ab) = 1$, and $\alpha(ba) = 1^{-1} = 4$. A *walk* in a voltage graph is a sequence of edges $e_1, e_2, \ldots, e_n$, where edges e_i and e_{i+1} $(1 \leq i < n)$ have an endpoint in common. The walk is said to be of length n. Each edge e_i has either a $+$ or $-$ sign. A walk is *closed* if its starting and ending vertices are the same. A *directed closed walk* is a closed walk in which there is no edge with a *minus* direction.

The *net voltage on a walk*, $W = e_1^{\sigma_1}, \ldots, e_n^{\sigma_n}$ (where $\sigma_i = +or-$), is the product (under the *group operation*) $\alpha(e_1^{\sigma_1}) \ldots \alpha(e_n^{\sigma_n})$ of voltages on the edges of W. A closed walk starting and ending at vertex u is called a *u-based closed walk*. Gross and Alpert [GT87] have shown that the set of net voltages occurring on all the u-based closed walks forms a subgroup of the voltage group. This is called the *local voltage group* at u, and is denoted by $A(u)$. In Figure 14a, the sequence of edges $\langle bc, cd, db \rangle$ is a b-based closed walk. The net voltage on this walk is $(3 + 1 + 0) (\text{mod } 5) = 4$, and $A(b)$ is the same as the voltage group.

A graph that has exactly one node and has one or more self-loops is called a *bouquet of circles*. If the base graph of a voltage graph is a bouquet of circles, then the local voltage group at the node is simply the subgroup generated by the voltages on all the loops [GT87]. Figure 14b shows a bouquet of circles as a voltage graph with $\langle\{0,1,2,3,4\}, +(\text{mod } 5)\rangle$ as the voltage group. The local voltage group at the node is the subgroup generated by voltages $1, 2$, and 4; and it is the same as the voltage group. When the base graph is not a bouquet of circles, Gross and Tucker suggest the following procedure to transform it to a bouquet of circles. The transformation is done with respect to any reference node u, and it has been shown that $A(u)$ remains unchanged under the transformation.
The steps of Gross and Tucker's procedure are as follow:

Step 1. Select an arbitrary spanning tree T; e.g., with some reference node (say u) as the root.

Step 2. Compute the T-potential for all nodes v in G. For each $v \in G$ there is a unique path in the tree from u to v. $\alpha(v, T)$, the T-potential of a vertex v, is defined as the net voltage in the path from u to v.

Step 3. Compute T-voltages for all edges e in G. If v and w are the initial and terminal vertices of e^+, then the T-voltage, $\alpha_T(e)$, is defined as the *product* $\alpha(v, T).\alpha(e).\alpha(w, T)^{-1}$.

Step 4. If there is an edge with group identity as the T-voltage, then the two vertices connected by this edge have the same potential. Merge the endpoints of all such edges, and remove such edges from the graph. The transformed graph is then a bouquet of circles.

4.3 Voltage Graphs for Finding the Meaningful Versions

We claim that Gross and Tucker's algorithm can be adapted to find the meaningful number of versions of each function in an SCC. The correspondence is as follows. If N is the number of registers, then $< \{0, 1, \ldots, N-1\}, +_N >$ is our voltage group. $+_N$, i.e., addition modulo N, is our group operation, and an SCC in a call graph is our base graph. The distances in the SCC are the voltage assignments; however, the only meaningful walks in our case are directed walks. We show that if any SCC can be converted into a bouquet

of circles by the method of Gross and Tucker, the set of net voltages on all directed u-based closed walks (u is the reference node) remain unchanged. In fact, the elements of this set are the meaningful versions for the function represented by u (assuming the existence of version number *zero*). Then, by using Lemma 2, we can calculate the meaningful versions for the function.

It may seem that we cannot directly use the procedure described above because it allows walks over *minus*-directed edges, whereas we are constrained to consider walks only over *plus*-directed edges. We claim that since the graphs considered for our purpose are always strongly connected, the above procedure is still applicable.

Theorem 2 *Let G be an SCC, and α be some voltage assignment to G. Let T be a directed spanning tree. Following the procedure due to Gross and Tucker, let the graph be transformed to a bouquet of circles with u as the reference node. Then the set of net voltages at u due to all the u-based directed closed walks remains unchanged under the transformation.*

Proof of Theorem 2 Since the choice of a spanning tree in the procedure is arbitrary, we can choose a directed spanning tree (while traversing from the root, we never encounter a reverse edge). After assigning T-voltages to edges, the local voltage group at u remains unchanged [GT87], which means that the set of net voltages on all the u-based closed walks at node u remains unchanged. This also implies that the set of net voltages on all the u-based directed closed walks remains unchanged. In their transformation, Gross and Tucker are able to merge the endpoints of an edge with group identity as the voltage, because the local voltage groups at such endpoints are the same. The only thing we then must show is that we too can merge the endpoints of an edge having zero (the group identity) T-voltage.

In Figure 15, the directed edge e from v to w has voltage zero. Since the graph is an SCC, there must be a directed path from w to v with net voltage, say t. Let V and W be the sets of net voltages of all directed closed walks at v and w, respectively. Let $w_1 \in W$. Consider a v-based directed closed walk as follows. Start from v; go to w along e; traverse all edges of the w-based closed walk that resulted in the net voltage of w_1; and then come back to v along the same set of edges that make a net voltage of t, mentioned above. Since t may not be equal to zero, traverse the directed cyclic path comprising e and the path having a net voltage of t, $N-1$ more times to finally reach v. The directed closed walk so described has a net voltage of w_1. It implies

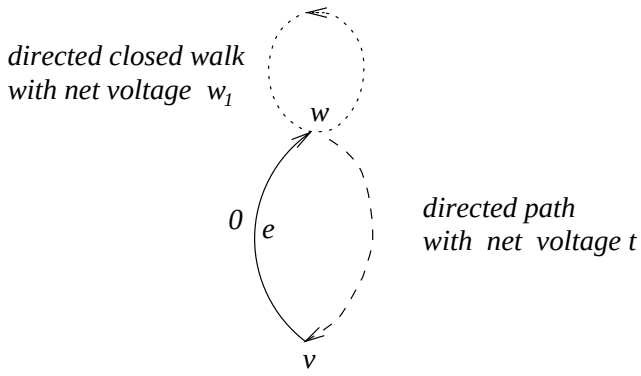

Figure 15: In an SCC, w and v have identical register assignments

that $w_1 \in V$. We can show, similarly, that if $v_1 \in V$ then $v_1 \in W$; therefore $V = W$. Thus we can merge the vertices v and w and remove the edge e without affecting the net voltages at other points in the graph.

Proof of Theorem 2 □

After merging the endpoints of all edges with zero voltage, the call graph is a bouquet of circles. As discussed earlier, by using Lemmas 2 and 3, what we find is the number of versions for each function in the SCC. But we are required to find the number of actual assignments to each function. For this we take a function, say F, in the SCC as the reference node. The set of register assignments to F, S_F, is obtained using the procedure described (it is assumed that F has an assignment of zero). Then, take any other function, say G, in the SCC. Find a directed path from F to G and let the sum of distances in the path be d_{FG}. The set of register assignments to G is given by $\{d \ +_N \ d_{FG} \mid d \ \in \ S_F\}$.

4.3.1 An Example

Figure 16a shows the SCC involving four functions F, G, H, and I, whose definitions are given in Example 7 in Appendix A. This SCC is the base graph of the voltage graph that we construct. The numbers along the edges are the distances between the functions in the SCC. These distances could be obtained by inspecting the DAGs of the functions in the SCC. Let us assume that we have 16 registers to decide on the number of versions; i.e., the value of N is 16. We take $< \{0, 1, \ldots, 15\}, +_{16} >$ as our voltage group.

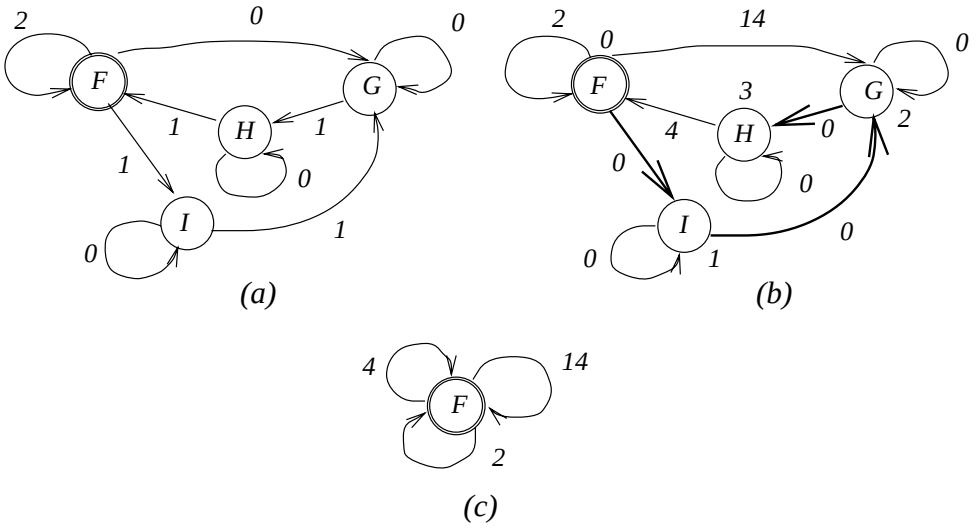

Figure 16: Transformation of an SCC to a bouquet of circles

To apply the algorithm of Gross and Tucker, we select F as our reference node. In Figure 16b, the bold lines show the chosen directed spanning tree T, with F as the root. The T-potentials of the nodes and the T-voltages of the edges of the SCC are as shown in the figure. Figure 16c shows the corresponding bouquet of circles.

By Lemma 2, the number of meaningful versions for F is 8, and by Lemma 3, the number of meaningful versions for functions G, H, and I is also 8. Without any analysis we would have taken 16 versions for each of the functions.

4.4 Voltage-Graph-Based Optimizations

Viewing an SCC as a voltage graph allows us to do other optimizations. If the distances along the n self-loops of a bouquet are $r_1, r_2, \ldots, r_n$ (called *generators*), then the number of assignments to the reference node is $q = N/\gcd(N, r_1, \ldots, r_n)$. This number q may be large, and we would prefer a smaller number. In the expression for q, by changing a certain distance, say r_i to r_i', we may reduce its value. But what is the implication? Observe that each self-loop in the bouquet uniquely corresponds to some directed edge in the original SCC. In fact, all of our analysis remains valid if we do the same modification to the corresponding edge in the original SCC.

Lemma 4 *Let an SCC in a call graph be transformed to a bouquet of circles. Modifying a generator by Δ in the bouquet amounts to modifying the distance of the corresponding edge in the original SCC by Δ ($\Delta \in A$, the voltage group).*

Proof of Lemma 4 No self-loop in the bouquet corresponds to any edge in the spanning tree T, e.g., which T-potentials and T-voltages are calculated. So we are not modifying the T-potentials. Let e^+ be the edge from vertex v to w in the call graph which is not in T, and $\alpha(e^+)$ be its assigned voltage. The T-potentials of v and w are $\alpha(v,T)$ and $\alpha(w,T)$, respectively. This edge will become a self-loop in the bouquet of circles, and its transformed voltage will be a generator that is $\alpha(v,T).\alpha(e^+).\alpha(w,T)^{-1}$. Our voltage group is $(\{0,1,\ldots,(N-1)\},+_N)$. It is also an abelian group. Adding voltage Δ to this generator amounts to

$$\alpha(v,T).\alpha(e^+).\alpha(w,T)^{-1}.\Delta \quad \text{i.e.,}$$

$$\alpha(v,T).(\alpha(e^+).\Delta).\alpha(w,T)^{-1}$$

Proof of Lemma 4 □

For the example in Figure 16, we have seen that the number of meaningful versions for each member of the SCC is 8 when the number of registers is 16. If we change the voltage of one self-loop from 14 to 16 (i.e., to 0 with $\Delta = 2$) and of another from 2 to 0 ($\Delta = 14$), then the number of meaningful versions for F (and hence for each other member of the SCC) will be 4. For this we must change the distance in the SCC between F to G from 0 to 2 ($0 +_{16} 2 = 2$) and the distance for the self-loop of F from 2 to 0 ($2 +_{16} 14 = 0$). This amounts to just switching the evaluation order of calls G and F in the body of F, and as we have discussed earlier, this change is safe.

Based on the last lemma, we now define an algorithm to find the values of Δ so that the number of versions is reduced. Essentially, we find a new evaluation order of the function calls inside a function DAG. Let us refer to the original SCC, in which edges are labeled with distances, as the OSCC (Figure 16a). After choosing a spanning tree T in the SCC, the nodes and edges are labeled with T-potentials and T-voltages. Let us call it TSCC (Figure 16b). The TSCC is then transformed to a bouquet (Figure 16c). Normally we call the voltages along the edges in the bouquet generators,

Let $r_1, r_2, \ldots, r_q$ be the generators in the bouquet.
$g = \gcd(N, r_1, \ldots, r_q)$, and
$n_1, \ldots, n_p$ are prime factors of N/g.
Step 1
Select the next n_i $(i = 1, \ldots, p)$
If $i > p$ return-message(failure in finding a new evaluation order);
Step 2
For each function node F in TSCC do
Let the generators of F be $r_1, \ldots, r_t$;
Find the corresponding distances $d_1, \ldots, d_t$ in OSCC;
Select a permutation of $d_1, \ldots, d_t$ such that when
distances $d_1, \ldots, d_t$ are replaced by the permutation,
each of the new generators $r'_1, \ldots, r'_t$ in TSCC is divisible by $g * n_i$;
/* $r', \ldots, r'_t$ are the new generators calculated, e.g., the permutation */
if such a permutation does not exist goto **Step 1**.
endfor
Step 3
Let the permutations replace the corresponding distances in the OSCC;
The present distances in the OSCC define the new evaluation order.

Figure 17: Procedure for finding an efficient evaluation order

but we now refer to the voltages along all edges in the TSCC, excepting the ones in the spanning tree T, as generators. We also refer to the voltages along those edges in the TSCC, which originate from node F and are not in the spanning tree, as the generators of F. With this background, Figure 17 describes the algorithm.

In Figure 5, $g = 2$ and the prime factors of N/g are $n_1 = 2$, $n_2 = 2$, and $n_3 = 2$. In the bouquet, the generators of F are 2 and 14. The corresponding distances of such generators in the OSCC are 2 and 0. When we take the permutation $[0, 2]$ of such distances and calculate the new generators of F in the bouquet, they are 4 and 0, respectively. Both are divisible by $n_1 * g = 4$; so a desired permutation exists. Similarly for nodes G, H, and I, we can find identity permutations as the desired permutations. Therefore $[0, 2]$ should define the new evaluation order in F. This essentially means that in the body of F, while evaluating the else branch, first evaluate the call to F, then evaluate the call to I, and finally evaluate the call to G. Identity permutations for G, H, and I imply that the evaluation order for

such functions remains unchanged. With this changed evaluation order, the number of versions for each function will be 4, and the number of spills remains the same as in the earlier case.

In our algorithm, once we find a set of desired permutations so that we can reduce the number of versions by a factor of n_i, we stop. The reason is that once we find one set of permutations, the next time we are unlikely to succeed.

5 Conclusion

In this paper, we have presented a method that keeps multiple versions of the same function to reduce memory spills in the presence of recursion. It is based on a bottom-up intraprocedural register allocation which itself, in the presence of recursion, eliminates many spills that would otherwise occur in existing methods. Keeping multiple versions reduces the number of spills further. We have also presented a method for determining the number of versions, for which the multiple-version strategy is guaranteed to be profitable. Further, we have solved the problem of determining the number of meaningful versions through the use of voltage graphs, and discussed other voltage-graph-based optimizations.

There are two obvious extensions to this work. If we extend our language to include higher order functions, the following issues will arise. In the DAG of a function, an occurrence of a higher order parameter will be a nonlinear node, and it will receive a register assignment, say i. Then the function that will be bound to the higher order parameter, say h, should have a version i. Essentially, we have the problem of transferring the information to the called function that registers R_0 till R_{i-1} are live at the point of the call. Therefore we need to know all possible bindings to h at compile time. For a higher order language, we cannot determine all such bindings during compile time. However, if we place restrictions on the language so that functions are prohibited from returning partial applications, i.e., the only higher order object that can be returned is a manifest function, then we can determine all bindings to h statically. Our analysis can then easily be extended to such a higher order language.

An inefficient method to deal with a generalized higher order language could be to spill all the occupied registers at the point of call to a higher order function, spill the result of the higher order function, and restore the

spilled registers after the function returns.

A related extension is to consider lazy languages. Here the problem is that since variables occuring in a function body may be bound to unevaluated expressions, or closures, the register requirement of a function cannot be determined by a mere examination of its body.

Acknowledgments

We thank A. A. Diwan for suggesting the use of voltage graphs for finding the number of meaningful versions. Thanks are also due to the two anonymous referees for suggestions that led to improvements in the presentation of the paper.

Appendix A

Example 1 /* from [LF80] */

$$S(k,n) = \begin{array}{l}\textsf{if } (k=0 \textsf{ and } n \le 2) \textsf{ then } S(1,\lceil n/2\rceil) + S(0,\lfloor n/2\rfloor) + \lfloor n/2\rfloor \\ \quad \textsf{else if } (k \ge 1 \textsf{ and } \mathrm{even}(n) \textsf{ and } n \ge 2) \\ \quad\quad \textsf{then } \ S(k-1,\lceil n/2\rceil) + (n-1) \\ \quad \textsf{else if } (k \ge 1 \textsf{ and } \mathrm{odd}(n) \textsf{ and } n \ge 3) \\ \quad\quad \textsf{then } S(k-1,\lceil n/2\rceil) + (n-2) \\ \quad \textsf{else if } (k \ge 0 \textsf{ and } n = 1) \textsf{ then } 0\end{array}$$

Example 2 /* map operation on binary trees */

$$\begin{array}{l} \mathit{fmap}\ [\,] = [\,] \\ \mathit{fmap}\ \mathit{Tnode}\ x\ \mathit{ltree}\ \mathit{rtree}\ =\ \mathit{Tnode}\ f(x)\ (\mathit{fmap}\ \mathit{ltree})\ (\mathit{fmap}\ \mathit{rtree}) \\ f\ x = x+1 \end{array}$$

Example 3 /* transpose of a square matrix represented as a powerlist */

$$\begin{array}{l} \mathit{transpose}\ \langle x\rangle = \langle x\rangle \\ \mathit{transpose}\ (p \mid q)\ |'\ (u \mid v) = \mathit{transpose}\ p\ |'\ \mathit{transpose}\ q) \\ \qquad\qquad\qquad\qquad\qquad\quad \mid (\mathit{transpose}\ u\ |'\ \mathit{transpose}\ v) \end{array}$$

Here, if A and B are similar matrices, then $A \mid B$ is the concatenation of A and B by rows; whereas $A\,|'\,B$ is their concatenation by columns.

Example 4 /* quicksort */

$$
\begin{aligned}
&qs\ [\,] = [\,]\\
&qs\ (x : xs) = append\ (qs\ (filter_l\ \ x\ xs))\\
&\qquad\qquad\qquad\qquad (append\ (filter_e\ x\ xs)\ (qs\ \ (filter_g\ x\ xs)))\\
&filter_g\ x\ [\,] = [\,]\\
&filter_g\ x\ (y : ys) = \mathsf{if}\ (x < y)\ \mathsf{then}\ y : (filter_g\ x\ ys)\ \mathsf{else}\ filter_g\ x\ ys\\
&filter_l\ x\ [\,] = [\,]\\
&filter_l\ x\ (y : ys) = \mathsf{if}\ (x > y)\ \mathsf{then}\ y : (filter_l\ x\ ys)\ \mathsf{else}\ filter_l\ x\ ys\\
&filter_e\ x\ [\,] = x : [\,]\\
&filter_e\ x\ (y : ys) = \mathsf{if}\ (x = y)\ \mathsf{then}\ y : (filter_e\ x\ ys)\ \mathsf{else}\ filter_e\ x\ ys
\end{aligned}
$$

Example 5 /* example involving mutual recursion */

$$
\begin{aligned}
&A(x, y) = \mathsf{if}\ (x <= 0)\ \mathsf{then}\ (1 + y + B(y))\ \mathsf{else}\ (x + y)\ +\ B(x - 1)\ +\ C(x - 2)\\
&B(p) = \mathsf{if}\ (p <= 0)\ \mathsf{then}\ 1\ \mathsf{else}\ B(p - 1)\ +\ C(p - 1)\\
&C(i) = \mathsf{if}\ (i <= 4)\ \mathsf{then}\ 1\ \mathsf{else}\ (2 + i)\ +\ A(i - 1, i - 2)\ +\ C(i - 1)\ +\ C(i - 2)\\
&\qquad\quad +C(i - 3)\ +\ C(i - 4)
\end{aligned}
$$

Example 6 /* illustrating duplication of spills */

$$
\begin{aligned}
&F(x) = \mathsf{if}\ (x \leq 10)\ \mathsf{then}\ 1\ \mathsf{else}\ F(x - 1) + F(x - 2) + A(x - 3)\\
&A(y) = \mathsf{if}\ (y \leq 5)\ \mathsf{then}\ 1\ \mathsf{else}\ A(x - 3) + A(x - 2) + B(x - 1)\\
&B(z) = \mathsf{if}\ (z \leq 6)\ \mathsf{then}\ 1\ \mathsf{else}\ B(z - 1) + F(z - 2)\\
&MAIN\colon\ F(11)
\end{aligned}
$$

Example 7 /* illustrating the application of voltage graphs */

$$
\begin{aligned}
&F(x) = \mathsf{if}\ (x \leq 1)\ \mathsf{then}\ 1\ \mathsf{else}\ G(x - 1) + I(x - 2) + F(x - 3)\\
&G(z) = \mathsf{if}\ (z \leq 1)\ \mathsf{then}\ 1\ \mathsf{else}\ G(z - 1) + H(z - 2)\\
&H(p) = \mathsf{if}\ (p \leq 2)\ \mathsf{then}\ 1\ \mathsf{else}\ H(p - 1) + F(p - 2)\\
&I(q) = \mathsf{if}\ (q \leq 2)\ \mathsf{then}\ 1\ \mathsf{else}\ I(q - 1) + G(q - 2)
\end{aligned}
$$

Appendix B

Arg k	Arg n	Incremental Strategy	Bottom-Up Strategy (Multiple Versions)				
			1 Register	2 Registers	3 Registers	4 Registers	5 Registers
1	10,000	894	447	376	310	332	280
1	30,000	1,794	897	808	670	688	552
1	50,000	2,522	1,261	986	832	868	857
1	80,000	3,794	1,897	1,287	1,312	1,017	1,075
1	500,000	12,774	6,010	6,010	4,593	5,213	4,609
0	500,000	20,670	10,335	6,764	7,665	5,965	6,254
2	500,000	7,894	3,947	2,583	2,913	2,276	2,484
3	500,000	4,878	2,439	2,295	1,780	1,988	1,647

Table 3: Number of Spills for Example 1 in Appendix A (Value of $N = 2$)

Nature of the Tree	Incremental Strategy	Bottom-Up Strategy (Multiple Versions)					
		1 Register	2 Registers	3 Registers	4 Registers	5 Registers	6 Registers
(Left chain)	2,000	1,000	0	0	0	0	0
(Right chain)	2,000	1,000	998	999	1,000	1,000	996
Centered	2,000	1,000	502	501	500	500	498
Centered	2,000	1,000	704	500	504	498	408
Centered	2,000	1,000	701	500	701	701	500
Centered	2,000	1,000	600	599	600	600	596
Centered	2,000	1,000	400	399	400	400	396
Left skewed	2,000	1,000	152	99	100	150	96

Table 4: Number of Spills for Example 2 in Appendix A (Value of $N = 2$)

Array Dimension	Array Size	Incremental strategy	Bottom-Up Strategy (Multiple Versions)					
			1 Register	2 Registers	4 Registers	5 Registers	6 Registers	7 Registers
1	2^2	3	3	2	0	0	0	0
2	2^4	15	15	14	12	10	6	0
3	2^6	63	63	58	46	44	48	42
4	2^8	255	255	234	182	167	173	207
5	2^{10}	1,023	1,023	938	726	681	623	675
6	2^{12}	4,095	4,095	3,754	2,902	2,736	2,572	2,347
8	2^{16}	65,535	65,535	60,074	46,422	43,692	41,975	41,170

Table 5: Number of Spills for Example 3 in Appendix A (Value of $N = 4$)

Nature of Input List	Incremental Strategy	Bottom-Up strategy (Multiple Versions)					
		1 Register	2 Registers	3 Registers	4 Registers	5 Registers	6 Registers
Sorted in reverse	4,000	2,000	1,950	1,950	1,948	1,950	1,950
Sorted	4,000	2,000	0	0	0	0	0
	4,000	2,000	50	48	48	50	48
	4,000	2,000	1,000	999	1,000	1,000	996
	3,400	1,700	1,401	1,197	1,200	1,200	1,194
	3,400	1,700	701	501	500	500	498
	3,000	1,500	500	501	500	500	498

Table 6: Number of Spills for Quicksort Example in Appendix A (Value of $N = 2$)

Arg Values	Incremental Strategy	Bottom-Up Strategy (Multiple Versions)					
		1 Register	3 Registers	4 Registers	5 Registers	6 Registers	7 Registers
8 8	61	50	29	18	21	12	0
10 10	299	244	150	132	109	99	82
12 12	1,541	1,256	801	680	576	502	610
14 14	7,938	6,467	4,512	3,683	2,810	2,633	2,808
16 16	40,710	33,165	22,027	18,684	13,378	13,274	12,678
18 18	209,044	170,297	108,456	96,571	70,496	67,319	62,860
20 20	1,073,040	874,147	577,948	496,215	382,665	352,998	333,641

Table 7: Number of Spills for Example 5 in Appendix A (Value of $N = 5$)

Arg i	Multiple Version Strategy							
	1 Register		2 Registers		3 Registers		4 Registers	
	r-op	m-op	r-op	m-op	r-op	m-op	r-op	m-op
10	326	216	326	180	326	188	326	142
14	2,258	1,504	2,258	1,272	2,258	1,174	2,258	1,340
18	15,500	10,332	15,500	8,736	15,500	8,138	15,500	7,982
20	40,586	27,056	40,586	22,876	40,586	21,724	40,586	19,492
22	106,262	70,840	106,262	59,892	106,262	56,714	106,262	50,866
24	278,204	185,468	278,204	156,812	278,204	147,074	278,204	138,844
26	728,354	485,568	728,354	410,544	728,354	383,934	728,354	378,952

Table 8: Memory (m-op) and Register (r-op) Operations when Parameters are Passed in Registers: The Fibonacci Example

References

[AJ76] A. V. Aho and S. C Johnson. Optimal code generation for expression trees. *Journal of the Association of Computing Machinery*, 23(3):488–501, 1976.

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the Association of Computing Machinery*, 24(1):146–160, 1977.

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[AS92] A. W. Appel and Z. Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5(3):191–221, 1992.

[Cha82] G. J. Chaitin. Register allocation via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, New York, 1982. ACM.

[Cho88] F. C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, New York, 1988. ACM.

[CMCH92] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, 1992.

[DH92] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effect on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.

[GH88] J. R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the International Conference on Supercomputing*, New York, 1988. ACM Press.

[GT87] J. L. Gross and J. L. Tucker. *Topological Graph Theory.* New York, 1987. Wiley Interscience.

[JW96] S. Jagannathan and A. Wright. Flow-directed inlining. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, 1996.

[LF80] R. E. Ladner and J. Fischer. Parallel prefix computation. *Journal of the Association of Computing Machinery*, 27(4):831–837, 1980.

[Mig92] M. Mignotte. *Mathematics for Computer Algebra*. New York, 1992. Springer-Verlag.

[Mis94] J. Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, 1994.

[Mur91] Thomas P. Murtagh. An improved storage management scheme for block structured languages. *ACM Transactions on Programming Languages and Systems*, 13(3):372–398, 1991.

[PJ87] S. L. Peyton-Jones. *Implementation of Functional Programming Languages*. Englewood Cliffs, NJ, 1987. Prentice-Hall.

[Sat96] M. Satpathy. Improved garbage collection and register usage for functional language implementations. PhD Thesis, Department of Computer Science and Engineering, I.I.T. Bombay, 1996.

[Ses87] P. Sestoft. Automatic call unfolding in a partial evaluator. In *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, Amsterdam, 1987. North-Holland.

[SH89] P. A. Steenkiste and J. L. Hennessy. A simple interprocedural register allocation and its effectiveness for Lisp. *ACM Transactions on Programming Languages and Systems*, 1989.

[SO90] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, New York, 1990. ACM.

[vL90] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A. 1990. Elsevier.

[Wal86] D. W. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*. New York, 1986. ACM.

[WJW75] W. Wulf, R. Johnsson, and C. Weinstock. *The Design of an Optimizing Compiler*, volume 2 of *Programming Language Series*. 1975. Elsevier.