UC Irvine
ICS Technical Reports

**Title**
An algorithm for minimizing spill code in loops

**Permalink**
https://escholarship.org/uc/item/7bh1r086

**Authors**
Kolson, David J.
Nicolau, Alexandru
Kennedy, Ken

**Publication Date**
1994-10-16

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

SLBAR
Z
699
C3
no. 94-43

# An Algorithm for Minimizing Spill Code in Loops*

David J. Kolson Alexandru Nicolau
Dept. of Info. and Comp. Science
University of California, Irvine
Irvine, CA 92717-3425

Ken Kennedy
Dept. of Comp. Science
Rice University
Houston, TX 77005

Technical Report #94-43

16 October 1994

**Abstract**

Loops are the main source of parallelism in applications. The issue of finding an optimal register allocation to loops has been an open issue for some time. In this case optimal refers to the minimization of spills from registers to memory. In this paper we address this issue and present an optimal, but exponential algorithm which allocates registers to loop bodies such that the spill code is minimal. We also show heuristic modifications to the algorithm that performs as well as the exponential approach on typical loops from scientific code. Finally, we examine this algorithm's feasibility in production compilers.

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

*This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

# 1 Introduction

Modern high-performance architectures rely on the exploitation of temporal and spatial parallelism in order to achieve high throughputs. Advanced compiler techniques such as basic block enlargement and trace scheduling [12, 11] provide the longer instruction streams necessary to keep these machines operating at or near peak performance. In fact, recent research indicates that basic block enlargement is a key component in achieving high degrees of parallelism in superscalar machines [20]. Due to the increased basic block size effective register allocation for these instruction streams becomes more complex.

Due to the importance of register allocation, *optimal* solutions have been extensively studied [15, 17, 7, 16]. Because register allocation is an NP-Hard problem in the general case [1], attempts at optimal solutions have either simplified the problem to index registers [15], or have used a brute-force exponential method for general-purpose registers with heuristics to prune the search space [7, 8, 16].

This approach does yield an effective strategy for straight-line code, but the key to increased performance in applications comes from speeding up loop bodies. Hence, we wish to minimize the number of loads and stores, due to spill code, that will be repeatedly executed within the loop. In order to extend the methods in [7, 8, 16, 15] to handle loops we must overcome the fundamental difficulty in dealing with loops—that of matching loop entry and exit register usage. In order for the loop body code to remain correct, the register usage at the loop's entry and exit must be equivalent so that correct results are obtained. While a previous paper [17] provided a technique for handling simple loops, optimality was lost in the process.

The focus of this paper is to show the viability of extending the techniques in [16, 17] to deal with loops by incorporating loop unrolling techniques into the algorithm. Previously it was not known whether optimal register allocation for a loop could be accomplished, regardless of efficiency of the algorithm. The difficulty was due to the fact that in order to ensure optimality for the overall loop, matching of registers at the top and bottom of the loop body may require additional spills. To optimally minimize these spills, loop unwinding with different register allocation in each unwound iteration may be needed. Furthermore, it was not known whether any limited unwinding can be guaranteed to converge and result in an optimal allocation.

In this paper we demonstrate that the algorithms for register allocation to basic blocks given in [16, 17] can be extended to allocate registers to simple loops. We also present a heuristic which, in practice, seems to perform as well as the exponential algorithm. In Section 2 we discuss

related work and the framework for our work. In Section 3 we present our algorithm and Section 4 details the theoretical aspects of our algorithm. Section 5 presents heuristics and Section 6 gives results. Finally, we discuss the possibility of using our heuristic technique in production compilers in Section 7.

## 2 Related Work

The general problem of register allocation is inherently NP-Hard and, as such, heuristic algorithms have been commonly utilized to determine some sub-optimal "acceptable" solution. One of these heuristic approaches is graph coloring [4, 7, 6]. In allocating registers by graph coloring, the live ranges of the variables in a basic block are examined. When two variable's lifetimes overlap they are said to interfere. A graph is then constructed wherein the nodes represent the variables and the edges joining nodes represent the interference of the two particular nodes being joined. The task is then to "color" the graph nodes with the same number of colors as physical registers. If no coloring of the graph is found, some variable is heuristically selected and spilled. As the key to good register allocation in this scheme is the selection of a particular variable to spill, heuristics for selection have received attention [5, 13] along with methods of coloring the graph [8, 2, 3]. Once the original code has been updated with the spill code, a new graph is then constructed to reflect the new interferences. This process is then repeated until some colorable graph is found.

However, many researchers have felt that for particularly critical code segments, such as the innermost loops of time-sensitive applications, an optimal allocation is necessary. In a seminal work on register allocation by Horwitz *et al.* [15], a method is presented for obtaining an optimal register allocation to index registers which minimizes the number of loads and stores. Further work was done which improved the efficiency of the Horwitz algorithm [19]. Further enhancements, including extension of the basic algorithm to deal with simple loops was proposed by Kennedy in [17]. More recent research has extended the basic idea in Horwitz's algorithm to include register allocation for general purpose registers [16].

The algorithm presented in [16] in its most general form, which we shall call BB-OPT, explores every possible register allocation at each virtual register access point in a basic block and produces an allocation which contains the minimal memory traffic. That is, the allocation produced is optimal with respect to the cost of memory loads and stores due to spill code.

Briefly the algorithm is as follows. The register access pattern is the sequence of virtual register

reads and writes found in the basic block (BB) code. A virtual register read is denoted by the virtual register name (e.g. VR1 denotes a read of virtual register one) and a virtual register write is denoted by concatenating the virtual register name with '*' (e.g. VR2* denotes a write to virtual register two). Further, the register access pattern reflects the semantics of the instruction, so that if the instruction is **VR1 = VR2 + VR3** the corresponding register access pattern is VR2, VR3, VR1* due to the fact that the arguments for an instruction are read before the write to the destination occurs[1]. This leads to the following definition:

**Definition 2.1** *A* **register access pattern** *is a sequence of virtual register reads and writes found in some code segment. A virtual register read is denoted by the virtual register name and a virtual register write is denoted by the virtual register name concatenated with '*'.*

Once the register access pattern is constructed, the initial register configuration is determined. A configuration is a mapping of virtual to real registers. Thus,

**Definition 2.2** *A* **register configuration** *or* **register mapping** *is a binding of virtual to real registers and represents the contents of the real registers at some point in computation.*

If allocation starts at the beginning of the program, all real registers are assumed free; otherwise the exiting configuration from the previous BB is used as the initial configuration for the current BB. After this information has been derived, we can invoke **Function** BB-OPT which will determine the optimal register allocation for the particular basic block under consideration[2].

When BB-OPT terminates, the new_state set will contain all possible exit virtual-to-real register mappings for the block. The set can then be examined for the lowest cost configuration and the virtual-to-real register allocation can then be determined by tracing the path back from the minimal configuration to the root of the allocation tree. A heuristic is presented in [16] which serves to restrict the growth of the allocation tree and is shown to produce good results with enlarged BB's, additionally further refinements given in [17] can be used.

This algorithm will indeed find the register allocation for a basic block with lowest spill code cost because it exhaustively explores every possible register configuration at each step. As it is, however, this basic algorithm is not adequate to allocate registers to loop bodies[3]. The virtual-to-real register configurations at the loop body (LB) exit must match the initial register assignments

[1] In [16], it is stated that this register access pattern is not realistic in the sense that it does not allow multiple register reads and writes to be considered concurrently. A simple extension, however, allows multi-register reads and writes in a single instruction.

[2] Refer to Appendix B for the pseudo-code to this function.

[3] For simplicity of exposition the loop body does not contain conditionals. Although extensions exist to handle this, they are beyond the scope of this paper.

at the top of the next iteration body. However, the loop body has two preceding basic blocks—the loop entry and loop exit. This considerably complicates matters as it is now necessary, when allocating registers to a loop body, to have the register configurations at LB exit match those at the LB entry. If the register configurations, resulting from the optimal basic block algorithm in [16, 17], are not the same at these two points, the computation performed by the resultant code is not correct unless register moves and/or spills are generated to enforce a match. However, since the cost of this additional spill code may vary greatly from configuration to configuration and would vary further by unrolling the loop some number of times, BB-OPT's results, which ignore this effect, cannot be optimal.

As an initial method for finding an allocation to an LB where the entry and exit configurations are equivalent, we might try the following: use BB-OPT to find an allocation for the LB and then insert spill code (and possibly copy operations—register to register moves) at the end of the LB, to make the values at LB entry match those at LB exit. Figure 1 illustrates this method using two real registers for allocation.

In this example, VR1 denotes virtual register one and R1 denotes real register one. Also, as can be seen by the code fragment in (a), VR3, VR5, VR6, and VR7 are constants. When allocating with BB-OPT the values present in the real registers at LB entry are VR3 and VR5. Therefore, the allocation to the loop body started with VR3 and VR5 occupying the two real registers R1 and R2. An important consequence of this is that it becomes necessary to re-load these values at the end of the LB so that the loop code remains correct in (b). Notice that the spill cost for the loop body in (b) is eight (resulting from BB-OPT) for a single sequential execution, and a cost of three was added in order to make it possible to iterate over this LB (i.e. so that the "backedge" values match the loop entry values).

Flow information can be used to further improve the code. The code segment in (c) has been adjusted accordingly and results in a cost of ten. Note that even though the register-to-register move does not read or write main memory, it is still counted as it is a necessary consequence of the generated spill code. This is another extension necessary in adapting the BB-OPT algorithm to the handling of loops. Unfortunately, even with this optimization, this method does not yield the minimal spill cost per iteration.

Figure 2 illustrates an allocation where the cost per iteration is lower. This lower cost of nine was found as a result of unrolling[4] the loop one iteration. When a match was found between

[4] When using this technique, we can remove the intermediate exit tests of the loop if the number of unwindings

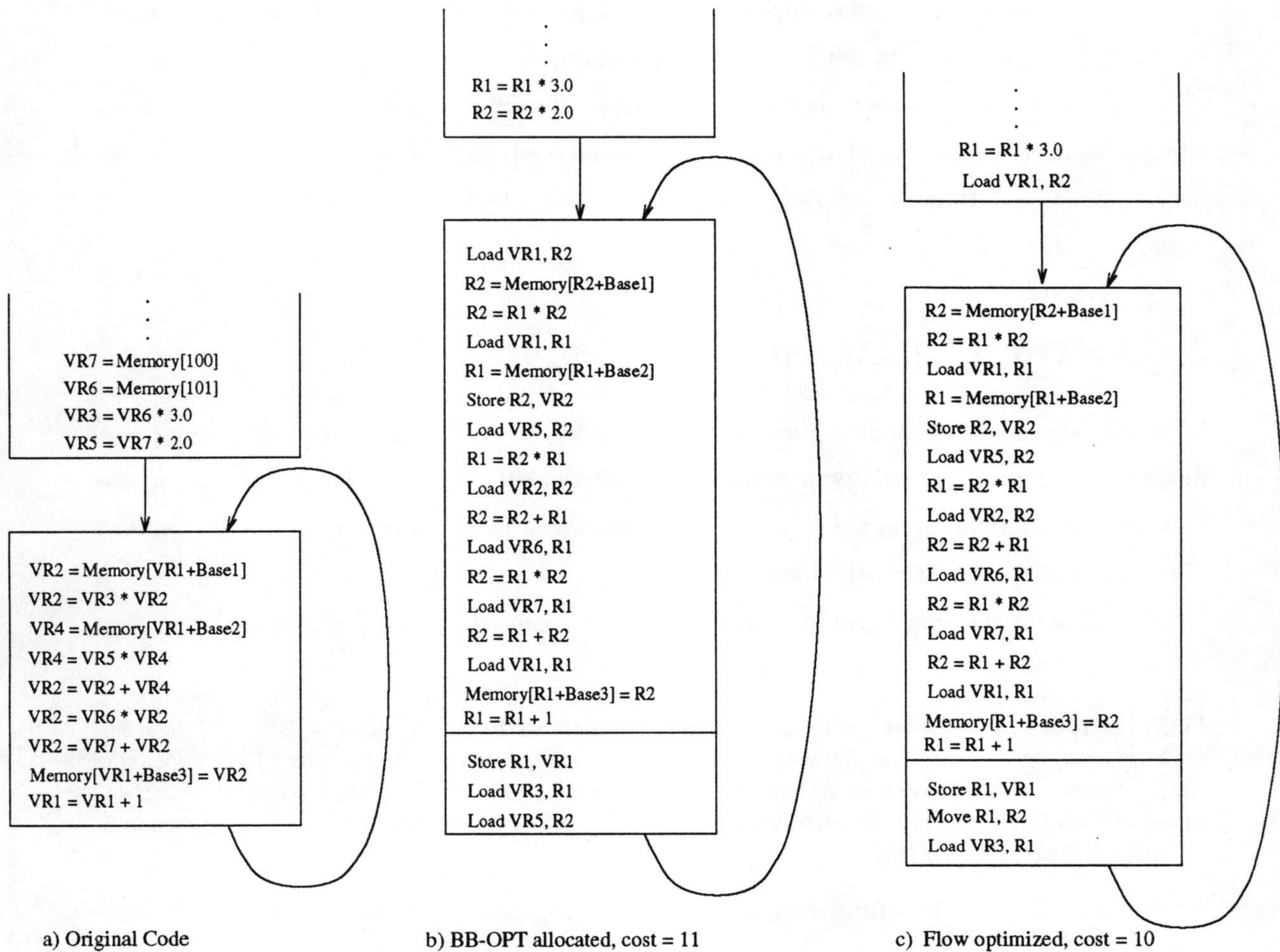



Figure 1: A loop basic block allocated with BB-OPT.

some loop entry and exit register configurations, flow of control was directed back to that point. The resultant code is minimal and correct as the allocation for the body of the second iteration resulted in an exit configuration that is minimal *and* matches the exit configuration of the first iteration as well. Thus correctness is preserved and a lower spill cost per iteration is found at the expense of a larger object code size.

It is not immediately obvious why only two iterations suffice to produce an allocation which results in less spill code. (In general, the optimal allocation may take longer to emerge and the resulting loop body may span more than one iteration.) In fact, this is why this problem has been an open issue. If the process of unwinding the loop and applying BB-OPT is continued, the cost *may* be decreased. However, previously it was not known whether this unrolling and allocation would terminate.

# 3 An Optimal Algorithm

By iteratively unrolling one loop iteration and applying BB-OPT to the resulting code, we can find a new loop body, potentially spanning several iterations of the original loop, such that: a) the cost of spills per iteration in the loop body is minimal; and b) the entry and exit configurations of the new loop match. The algorithm in Figure 3 called BB-OPT-LOOP performs this.

The following terms are used in conjunction with the exposition of the BB-OPT-LOOP algorithm.

**Definition 3.1** *An* **allocation tree** *is the tree produced by the application of BB-OPT to some register configuration and register access pattern. The root of the tree is the starting (given) register configuration and the leaves of the tree are register configurations which represent the virtual-to-real register bindings at the completion of the code segment. The allocation tree may contain many iterations of the original loop.*

**Definition 3.2** *An* **exit configuration** *is a particular register configuration that is a leaf in some allocation tree.*

**Definition 3.3** *An* **allocation path** *is a path through some allocation tree from the root to some leaf. This path defines a (unique) virtual-to-real register binding for the code segment.*

**Definition 3.4** *An* **iteration ancestor** *of register configuration $X$ is a register configuration $Y$ which lies on the allocation path from the initial register configuration to $X$ and the parent of $Y$ and $Y$ belong to iterations $i$ and $i+1$ for some $i$, respectively.*

is a multiple of the number of executed iterations. If this is not the case, we may simply leave the exit test in. In general, an adjustment to the code speculatively executed before the exit may be required.

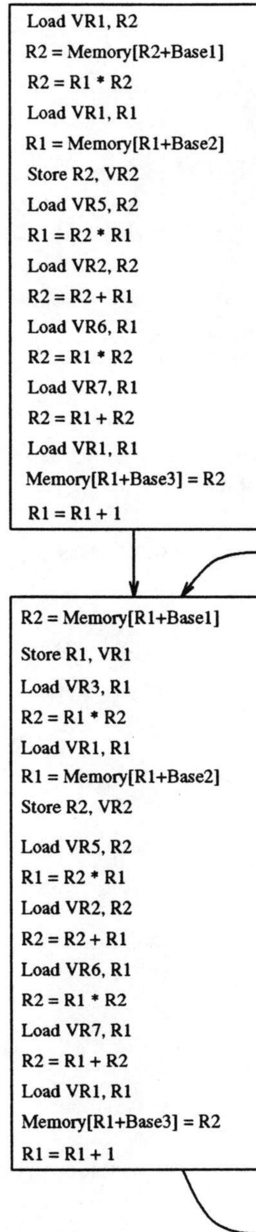



Figure 2: A BB-OPT-LOOP allocated loop basic block with cost nine.

```
Procedure BB-OPT-LOOP ( REGS : Initial register configuration;
                        RA : Register access pattern;
                        K : number of iterations)
Begin
    Set MIN to an empty configuration with ∞ average cost
    Set i to 0
    Set current state to REGS
    Loop
        Set save_state_set to null
        Foreach state S in the current register state set do
            new state set = BB-OPT(S, RA)
            Foreach state N in new state set do
                If N has an iteration ancestor A with same configuration then
                    Direct N to A
                    Delete N from new set state
                    AverageCost(N) = (Cost(N) - Cost(A)) / (Iter(N) - Iter(A))
                    If AverageCost(MIN) > AverageCost(N) then
                        MIN = N
                    Endif
                Endif
            Enddo
            Set save_state_set to save_set_state ∪ new_set_state
        Enddo
        Set i to i + 1
        Set current register state set to save_state_set
    Until i = K
    Return MIN
End BB-OPT-LOOP
```

Figure 3: A loop register allocation algorithm.

BB-OPT-LOOP works by iteratively using BB-OPT to compute the allocation tree for each unrolling of the loop body. BB-OPT-LOOP starts with the given initial register allocation and applies BB-OPT to the given loop body yielding a new state set. For each state in that new state-set all iteration ancestor configurations are examined to see if the new state and its ancestor match. If these two states match, then the average iteration cost for execution of this path is determined and compared with the current minimum, saving the new state if it becomes the new minimum. This only finds a local minimum, however this minimum is "global" over the number of iterations unrolled so far ($K$). Hence, if the loop were fully unrolled, this would indeed be a global minimum.

All of the configurations at the new unrolled depth are "collected" until all of the nodes from the previous depth have been processed. This is done with the save_state_set. What this accomplishes is a breadth-first expansion of the allocation tree, where each possible exit configuration from an allocation tree becomes a "root" of an allocation tree for the subsequent unrolling of the loop body. Once the bounded number of iterations ($K$) has been reached, the minimum average cost loop is returned. Note that this algorithm must always get an average cost less than or equal to what BB-OPT would get because we deal strictly with the costs calculated by BB-OPT, and add nothing more—beyond unrolling.

# 4 Convergence and Optimality

For simplicity of exposition we assume that no register has been allocated to a global value which is not used within the loop body. This is in no sense restrictive—good register allocation methodology would make all registers free and available to the innermost loops; globals would then be allocated afterward to the relevant portions of the code segments. Thus, the register access pattern cannot change for a given loop body.

Due to the exponential nature of BB-OPT and the fact that the register access pattern does not change, we can generate all possibilities for the exit configurations after one application of the algorithm. Then the costs associated with going from one configuration to each other exit configuration can be found. This requires an application of BB-OPT to each of the exit configurations found after the initial application using the initial loop configuration. (We illustrate this by example shortly.)

Why is it the case that all configurations are "reachable" from any other? This is true because

at each virtual register access all possibilities for spills are examined. Since the register access pattern for the loop body contains only those virtual registers accessed within the loop, there will be some point when register configurations are generated and are identical to those in other generation trees. Hence, the same set of exit configurations is derived by the application of BB-OPT to any one of the first iteration exit configurations and the register access pattern for the loop—differing only in the cost of the allocation path.

Basically this is a combinatorial observation. If there are $r$ real registers in the target machine and $v$ virtual registers in the register access pattern, there can be no more than $\binom{v}{r}$ possible register configurations. However, not all of those configurations will be legal as exit configurations, as the last virtual register accessed must be present to result in a legal allocation.

Now the costs associated with changing from each possible exit configuration to any other exit configuration have been determined. A graph can be constructed wherein each node represents one of the possible exit configurations and the edges between nodes represent the spill cost in going from some configuration to another configuration. Note that these edges are directed as it may not be possible to go from configuration $x$ to configuration $y$ for the same cost as $y$ to $x$. The result will then be a fully-connected, directed graph. We call this graph the *configuration graph*. There are an exponential number of nodes, hence this graph is exponential in size. We do not suggest (nor is it the case) that this graph need be built—we use it for the purposes of illustrating certain properties of this algorithm.

Figure 4 illustrates the method of building the configuration graph. For this example we have assumed that we are at the second iteration of unrolling and we have two physical registers. If we assume that both registers are initially free then the starting configuration for allocation to the loop body for the access pattern in (a) is $\{ \emptyset, \emptyset \}$. The first iteration will build an allocation tree in which the root is $\{\emptyset, \emptyset\}$ and the exit configurations are {**VR1, VR2***}, {**VR1, VR3**}, and {**VR1, VR4***}. BB-OPT may then be applied to each of these configurations and the allocations trees in (b) can be obtained.

The leaves have been labelled so that we may associate a label which is used in the configuration graph with its actual configuration in the allocation tree. As was explained previously, we have the costs associated with going from the allocation tree's initial configuration to all other possible exit configurations. In the case of the same configurations we take the lowest cost one, breaking ties arbitrarily.

The configuration graph can be constructed from the allocation trees in (b). Traversing a path

through the first allocation tree from the initial configuration labelled A, to VR1VR3, which has been labelled B, there are two possible paths which both terminate at a node with cost of two. Therefore, a directed edge from A to B with cost two is added to the configuration graph. Other edges are added similarly, one for each distinct lowest cost exit configuration in that allocation tree. We then repeat this procedure for each of the allocation trees in (b). Thus, a fully-connected graph results, as shown in (c). To find the optimal allocation, this graph is searched for the lowest cost cycle.

In this example, there are four virtual registers and two real registers for a total of $\binom{4}{2} = 6$ possible configurations. Notice that the allocation trees have only generated three configurations as exit configurations. The reason that there are less exit configurations lies in the virtual register access pattern. Note that all of the exit configurations contain **VR1**. This must be the case as a legal allocation must have **VR1** bound to a real register at this point for correct computations to take place.

## 4.1 Convergence

The question of convergence of the algorithm presented earlier can now be addressed. When an unrolling of the loop body and allocation to that iteration is performed, all of the edges in the graph from the initial configuration to all of the exit configurations are obtained. If the allocation algorithm is again applied to each of these nodes (e.g. unroll the loop body for another iteration), directed edges from each of the possible exit configurations to one another are obtained. Hence, the completely connected graph is incrementally built. In order to guarantee that the algorithm converges, it must shown that by unrolling, exit configurations that do not lie on the allocation path to the current exit configuration are not indefinitely generated.

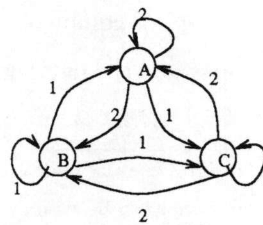
In terms of the constructed graph this is equivalent to finding a path from some starting node which eventually returns to that same node (i.e. a cycle must be found). However, we know that one must exist as the graph is completely connected[5]. Intuitively this converges because both the number of configurations and the number of transitions from one configuration to another is finite *and independent of (fixed w.r.t.) the number of iterations the loop executes*, although exponential.

[5] In some sense this is trivial as every node is directly connected to itself. However, the cost edge associated with this connection is not guaranteed to be a minimum. Indeed, in most cases, this edge tends to have a higher than average cost.

b) Allocation Trees

vr1vr2*vr3vr4*vr1

a) Register Access Pattern




c) Configuration Graph

Figure 4: Building a configuration graph from the allocation tree.

## 4.2 Optimality

The question of optimality can also be addressed with the notion of the configuration graph. An optimal allocation is one in which the memory traffic is minimized. When the loop body is unrolled, an optimal allocation becomes the allocation which has minimal memory traffic or spill cost *over the iterations* that are contained within the unrolled loop. Thus, in the optimal allocation, the ratio of the spill cost for the new unrolled loop body to the number of iterations it contains, is minimized. In the configuration graph this corresponds to the ratio of the total cost of a cycle to the number of nodes in that cycle.

**Definition 4.1** *The* **minimal average spill cost** *for a loop is*

$$\min \ \langle 1 \leq i \leq n \rangle \quad \frac{\sum_{j=1}^{i} Cost_j}{i}$$

*where $Cost_j$ is the cost associated with edge $j$ in some cycle of length $i$.*

That is, the optimal allocation is found by examining the average cost of all possible cycles in the configuration graph and taking the minimum.

Note that this does not simply correspond to the minimal cycle of length one in the graph. (A cycle of length one would imply that some allocation to the loop body is minimal *and* its initial configuration naturally (i.e. without spills or moves) matches its exit configuration.) In the worst-case it is possible that the optimal cycle must make a complete tour of the graph.

While the above can become quite expensive, BB-OPT-LOOP can be invoked with some $K$ significantly lower than the (expected) loop bound. In this case, the allocation returned by BB-OPT-LOOP will be optimal for the given initial configuration and number of unrollings ($K$), since the algorithm will explore all possible allocations with the given initial configuration. In the graph this is equivalent to finding the minimal average cycle with a bound ($K$) on the length of the cycle, and the start node equal to the node in the graph which corresponds to the initial configuration. If a lower average cost cycle exists, it must be farther away from the given initial configuration than $K$, or it may be reachable within length $K$ from some other configuration. Hence, the allocation returned must be minimal for the given parameters. In addition, all the pruning optimizations proposed in [15, 17] would still apply.

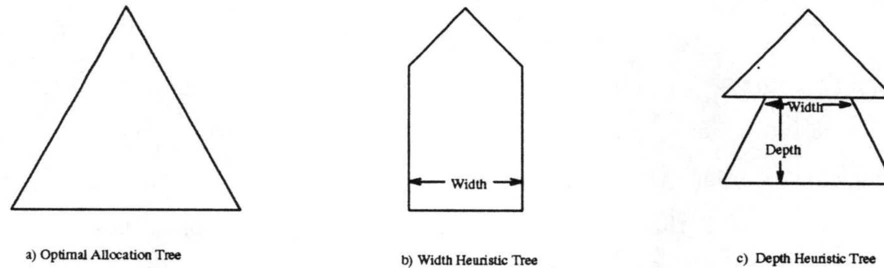



Figure 5: Pictorial representation of allocation algorithm.

# 5 Heuristic Pruning

The algorithm given above will compute the optimal register allocation for a loop. However, even for moderately long loops, the algorithm may be computationally prohibitive. Therefore, it becomes impractical to obtain an optimal allocation for large loop bodies even with the enhancements proposed for basic blocks in [16, 15, 17]. However, the optimal algorithm does provide a strong starting point for determining good heuristics.

The computational complexity in this algorithm arises from the replacement of each physical register in the current configuration when a read or write miss occurs. One way of reducing the complexity is to introduce heuristics to prune the search tree. One pruning method is to select only one of the $r$ physical registers in the current configuration for replacement. The heuristic proposed in [16] does this and prunes a significant amount of the search tree by looking at the tradeoffs between replacing the most distant clean and dirty[6] registers.

Since this scheme only expands a node once, the heuristic function must be a very close approximation to the true optimal, if results are to be good. The heuristic presented in [16] performs well for basic blocks. However, in allocating registers in loops, the difference in cost between the true optimal and the heuristic allocations is even more critical, as slight differences can have cumulative effects over large numbers of iterations, and thus can have a significant impact on overall performance.

We have utilized two strategies for heuristic pruning. The first restricts breadth (width) of the search, while the second strategy restricts the depth between successive width restrictions. Figure 5 is a pictorial representation of the size of the optimal and heuristic allocation trees.

[6] Clean refers to the case when the virtual registers memory location is consistent with the value in its assigned register, while dirty refers to an inconsistency.

### 5.1 Width Restriction

The first approach to pruning the search tree is to expand each node in the current set and then keep only the best $m$ of those expanded states. That is, if a virtual register access causes a read/write miss in a node in the current set under examination, we replace each virtual register in that node with the virtual register causing the miss. Then, after all nodes have been examined we keep only the $m$ lowest cost nodes in the new state set. The number of nodes which result from expanding the current state-set is bounded by $mr$. After this expansion step we retain only the $m$ best of the $mr$ new nodes.

This heuristic serves to capture the intuition that at certain stages in the search tree there is some set of nodes that appear to be better candidates than others for expansion and can be thought of as a local greedy approach. One or more nodes might provide better solutions if they are fully expanded so that possibly lower cost paths might have eventually been generated. Therefore, we have the ability to follow the most promising $m$ nodes through the search tree.

### 5.2 Depth Restriction

Another method for heuristic pruning of the allocation tree is depth restriction and refers to the amount of look-ahead into the allocation tree before the width heuristic is applied. Using the width restriction alone effectively yields a look-ahead of one, as it only locally selects nodes for future expansion. With the addition of depth restriction, the selection of nodes by the width heuristic is deferred until the desired depth has been reached. When the depth restriction parameter has been reached the width heuristic is applied (hopefully) selecting nodes that are closer to the minimal allocation.

Depth restriction allows the deficiency found in a local-greedy approach to be partially alleviated. Nodes which appear to be "good candidates" will be expanded along with other nodes that would have been pruned. This allows a sort of "recovery" mechanism within the depth size window in the selection of nodes for expansion.

## 6 Results

We took the Livermore loops in C source and unrolled them three times to obtain larger code segments and live ranges. The unrolled code was then compiled with the GNU C Compiler into

SPARC assembly code. The register access pattern was then derived from the resultant assembly code.

With these register access patterns, several experiments were conducted. Results of the BB-OPT[7] algorithm (which was applied on the basic blocks of the loop) on the Livermore Loops are presented in subsection 6.1. Results of forcing the entry and exit register mappings, as suggested by [17], and "patching" (the use of moves and spills to match register usage) are detailed in subsection 6.2. In subsections 6.3 and 6.4 results are presented for BB-OPT-LOOP optimal and heuristic algorithms, respectively. Lastly, the results of a comparison of BB-OPT-LOOP with a graph coloring algorithm are presented in subsection 6.5.

## 6.1 BB-OPT

For each Livermore Loop, we calculated the optimal allocation, using BB-OPT, for the pre-loop basic block and used the exit configuration for that allocation as the initial register bindings for the loop to determine the initial values for the loop entry (i.e. the allocation of registers before the loop is entered).

Our results (Table 2, column "BB-OPT") reflect the dynamic spill cost. This was computed by determining the number of iterations that a loop executes and then multiplying by the spill cost. In the case of BB-OPT, unwound for three iterations, the spill cost was determined for the unwound loop body and then multiplied by the number of executions of that loop body (i.e. the initial number of iterations divided by three).

## 6.2 Forcing Convergence

One idea presented in [17] as an extension to Horwitz's algorithm to deal with simple loops is to "force" the register configurations to be equal to one another at the loop entry and exit. In this scheme only one iteration of the loop body is examined. Some state is chosen as both the initial and final configuration with the allocation proceeding from that state. When the end of the loop code is reached, register mismatches may be found as all values are present but are not in the same physical registers as at loop entry. Mismatches can be dealt with by introducing register-to-register copy operations at this point. To find the minimal allocation for the loop in this manner requires that all possible initial configurations be tried.

[7] We actually unwound the loop three times before applying BB-OPT, to create better opportunity for BB-OPT to do well.

We have used this idea in conjunction with BB-OPT. Since it is not practical to apply BB-OPT to every possible initial configuration for the loop, what we have done is to assume that all of the registers at loop entry are free. BB-OPT is then applied to the register access pattern, and at the point when all free registers have been taken, that particular configuration is noted. Allocation then continues until the entire allocation tree is built. At this point the leaves of the allocation tree are searched for the least cost configuration which matches the one previously noted (we refer to this as the "Force Method"). This results in an allocation for the loop code in which the values at the end of the loop will be in the correct registers for the subsequent iterations. Another approach is to find the absolute minimum leaf in the allocation tree and introduce spill code to match it with the noted initial configuration (we refer to this as a "Patch Method"). In both cases the entry and exit register mappings have been forced equal. Table 1 shows the results for both of these methods for two and four registers as the true optimal is computationally infeasible to calculate in all cases.

Somewhat surprisingly, in many cases it is more cost-effective to introduce spill code at the loop exit than to force the register mappings to be equal. Since certain values are "anticipated" as being live for a long time (e.g., specific values are forced into the allocation at the loop exit), more spill code than need be is created because non-exit values are "favored" for spill code. Thus, an artificial increase in register pressure results at the subsequent virtual accesses. Also, because we are dealing with a small number of real registers the spill cost has a small upper bound. For instance, in the case of two real registers, if both virtual registers need to be saved and then two new virtual registers need to be loaded we have a maximum spill cost of four. Since we have picked the node with smallest cost to direct back to loop entry and since that direction is bounded by a small replacement cost, intuitively it seems that this is a better decision (since we have started with an allocation which is optimal w.r.t. this block and have imposed a small additional cost).

### 6.3 BB-OPT-LOOP, Optimal Results

In Table 2 we have obtained the true optimal results for BB-OPT-LOOP. As discussed earlier, that for a given initial register configuration, a minimum found after unrolling a specified number of times is optimal w.r.t. that initial configuration. (Our method for determining that initial configuration was outlined earlier in subsection 6.2.) Although it is possible that a lower cost exists, which can be found by examining all conceivable entry mappings, no lower cost can be found for these code segments with the particular initial configuration used. The column labelled

| | BB-OPT | | |
|---|---|---|---|
| Program | Number of Registers | Force Method | Patch Method |
| | 2 | 13,734 | 13,334 |
| LL1 | 4 | 6,668 | 5,868 |
| | 2 | 4,736 | 4,670 |
| LL2 | 4 | 1,934 | 1,467 |
| | 2 | 11,334 | 10,334 |
| LL3 | 4 | 5,334 | 4,000 |
| | 2 | 3,920 | 3,752 |
| LL4 | 4 | 2,296 | 1,904 |
| | 2 | 17,316 | 17,427 |
| LL5 | 4 | 5,772 | 5,994 |
| | 2 | 17,649 | 17,760 |
| LL6 | 4 | 7,326 | 6,438 |
| | 2 | 6,060 | 5,760 |
| LL7 | 4 | – | – |
| | 2 | – | – |
| LL8 | 4 | – | – |
| | 2 | 5,338 | 5,202 |
| LL9 | 4 | 1,360 | 1,224 |
| | 2 | 6,494 | 6,732 |
| LL10 | 4 | – | – |
| | 2 | 12,987 | 13,653 |
| LL11 | 4 | 6,660 | 5,661 |
| | 2 | 12,987 | 13,653 |
| LL12 | 4 | 6,660 | 5,661 |

Table 1: Spill costs for the two methods of matching register maps.

| Program | Number of Registers | BB-OPT | | | BB-OPT-LOOP | |
|---|---|---|---|---|---|---|
| | | Method | Dynamic Spill Cost | Spills per Iteration | Dynamic Spill Cost | Spills per Iteration |
| LL1 | 2 | Patch | 13,334 | 33.3 | 12,800 | 32 |
| | 4 | Patch | 5,868 | 14.7 | 4,800 | 12 |
| LL2 | 2 | Patch | 10,000 | 50 | 9,000 | 45 |
| | 4 | Patch | 1,467 | 7.3 | 1,400 | 7 |
| LL3 | 2 | Patch | 10,334 | 10.3 | 9,000 | 9 |
| | 4 | Patch | 4,000 | 4 | 2,001 | 2 |
| LL4 | 2 | Patch | 3,752 | 22.3 | 3,528 | 21 |
| | 4 | Patch | 1,904 | 11.3 | 1,514 | 9 |
| LL5 | 2 | Force | 17,316 | 52 | 17,264 | 52 |
| | 4 | Force | 5,772 | 17.3 | 5,312 | 16 |
| LL6 | 2 | Force | 17,649 | 53 | 17,649 | 53 |
| | 4 | Patch | 6,438 | 19.3 | 5,661 | 17 |
| LL7 | 2 | Patch | 5,760 | 48 | 5,760 | 48 |
| | 4 | – | – | – | – | – |
| LL8 | 2 | – | – | – | – | – |
| | 4 | – | – | – | – | – |
| LL9 | 2 | Patch | 5,202 | 51 | 5,000 | 50 |
| | 4 | Patch | 1,224 | 12 | 1,100 | 11 |
| LL10 | 2 | Force | 6,494 | 63.7 | 6,200 | 62 |
| | 4 | – | – | – | – | – |
| LL11 | 2 | Force | 12,987 | 13 | 12,974 | 13 |
| | 4 | Patch | 5,661 | 5.7 | 4,991 | 4 |
| LL12 | 2 | Force | 12,987 | 13 | 12,974 | 13 |
| | 4 | Patch | 5,661 | 5.7 | 4,991 | 4 |

Table 2: True optimal results of loop allocation algorithm (as discussed in text).

BB-OPT contains the results of using BB-OPT *only* on a triply unwound loop for a given program while the BB-OPT-LOOP column contains the results of unwinding the loop the indicated number of iterations. The column labelled "method" indicates the method selected from Table 1 which provided the least spill cost. Finally, the spills per iteration column shows the number of spills in a single iteration. Note that these are not necessarily whole numbers for the BB-OPT column as the spill code was allocated for a trace of three iterations, so the cost here represents an amortized cost.

From Table 2, we see that the savings in spill cost uniformly increases as the number of real registers available increases. When we only have two real registers, the maximal difference between the two methods is bounded by four (in the worst case both registers have to be spilled and then loaded.). As the number of registers increases, this bound also increases. Thus, BB-OPT-LOOP

will have an increased performance advantage over BB-OPT with larger numbers of registers (if the code has high register pressure). Of course, since these techniques are expensive, some of the longer loops and higher number of registers were too time consuming to compute.

### 6.4 BB-OPT-LOOP, Heuristic Results

In Table 3, the results for the heuristic BB-OPT-LOOP can be found. Because our heuristic bounds the width of the search tree, the first column labelled "Width = 1" represents the case where the minimum is followed through the allocation tree. This strategy yields results very similar to those obtained by allocation via BB-OPT with entry/exit point matching.

Also, as the width is increased, closer approximation of the optimal numbers occurs. With very few exceptions, increasing the width of the tree yields improved results. In the cases where the heuristic broke down, a node generated children which locally appeared to be better choices (i.e. they had lower costs than their siblings) and served to "knock" other nodes out of the expansion set. As the width of the tree is expanded, beyond that which is shown, this phenomena diminishes rapidly. Evidently, at each stage in the allocation tree, there are only a few nodes which are good candidates and as the tree expands, it becomes evident which of those candidates will eventually lead to the minimum.

Of course, as the number of available registers increases beyond the number of overlapping lifetimes, there is no spill code needed. Table 3 demonstrates some cases with eight registers where no spill code is necessary.

Tables 4 and 5 show results for our algorithm with the depth heuristic and the width heuristic working in conjunction. In both cases, heuristic widths of 1, 2, and 5 were used. In Table 4 a depth of 2 was used; while in Table 5 a depth of 3 was used. These results show that the spill cost per original iteration decreases as the width increases with very few exceptions, particularly as the depth increases.

### 6.5 Comparison with Graph Coloring

The register allocation algorithm currently most widely used is the graph coloring algorithm mentioned earlier. The Gnu Standard Distribution C Compiler (gcc) implements a graph coloring scheme in allocating registers. Further, the code produced by this compiler is generally accepted to be of high quality [14]. We therefore have used this compiler as a metric of code produced by a graph coloring algorithm on our benchmarks.

| Program | Number of Registers | Width = 1 Cost | Iters | Cost / Iter | Width = 2 Cost | Iters | Cost / Iter | Width = 5 Cost | Iters | Cost / Iter |
|---|---|---|---|---|---|---|---|---|---|---|
| LL1 | 2 | 13,200 | 3 | 33 | 12,800 | 3 | 32 | 12,800 | 3 | 32 |
| | 4 | 5,600 | 3 | 14 | 5,600 | 3 | 14 | 6,400 | 3 | 16 |
| | 8 | 1,200 | 3 | 3 | 1,200 | 3 | 3 | 1,200 | 3 | 3 |
| LL2 | 2 | 10,000 | 3 | 50 | 9,000 | 3 | 45 | 9,000 | 3 | 45 |
| | 4 | 1,600 | 4 | 8 | 1,600 | 4 | 8 | 1,400 | 5 | 7 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL3 | 2 | 10,000 | 3 | 10 | 9,000 | 3 | 9 | 9,000 | 3 | 9 |
| | 4 | 2,000 | 4 | 2 | 2,000 | 4 | 2 | 2,000 | 4 | 2 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL4 | 2 | 4,032 | 3 | 24 | 3,528 | 3 | 21 | 3,528 | 3 | 21 |
| | 4 | 1,680 | 5 | 10 | 1,680 | 4 | 10 | 1,680 | 4 | 10 |
| | 8 | 168 | 3 | 1 | 168 | 3 | 1 | 168 | 3 | 1 |
| LL5 | 2 | 19,256 | 3 | 58 | 17,264 | 3 | 52 | 17,264 | 3 | 52 |
| | 4 | 5,312 | 4 | 16 | 5,312 | 4 | 16 | 5,312 | 4 | 16 |
| | 8 | 0 | 3 | 0 | 0 | 3 | 0 | 0 | 3 | 0 |
| LL6 | 2 | 19,647 | 3 | 59 | 17,649 | 3 | 53 | 17,649 | 3 | 53 |
| | 4 | 5,661 | 4 | 17 | 5,661 | 4 | 17 | 5,661 | 3 | 17 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL7 | 2 | 5,760 | 3 | 48 | 5,760 | 3 | 48 | 5,760 | 3 | 48 |
| | 4 | 2,040 | 4 | 17 | 2,160 | 4 | 18 | 2,040 | 3 | 17 |
| | 8 | 360 | 4 | 3 | 360 | 4 | 3 | 360 | 3 | 3 |
| LL8 | 2 | 5,624 | 3 | 296 | 4,788 | 3 | 252 | 4,617 | 3 | 243 |
| | 4 | 2,128 | 4 | 112 | 2,109 | 4 | 111 | 2,014 | 4 | 106 |
| | 8 | 798 | 3 | 42 | 855 | 3 | 45 | 779 | 3 | 41 |
| LL9 | 2 | 5,100 | 3 | 51 | 5,000 | 3 | 50 | 5,000 | 3 | 50 |
| | 4 | 1,300 | 4 | 13 | 1,200 | 4 | 12 | 1,200 | 3 | 12 |
| | 8 | 100 | 3 | 1 | 100 | 3 | 1 | 100 | 3 | 1 |
| LL10 | 2 | 6,200 | 3 | 62 | 6,200 | 3 | 62 | 6,200 | 3 | 62 |
| | 4 | 2,200 | 4 | 22 | 2,100 | 4 | 21 | 2,400 | 3 | 24 |
| | 8 | 600 | 3 | 6 | 500 | 3 | 5 | 500 | 3 | 5 |
| LL11 | 2 | 14,970 | 3 | 15 | 12,974 | 3 | 13 | 12,974 | 3 | 13 |
| | 4 | 3,992 | 4 | 4 | 3,992 | 4 | 4 | 3,992 | 3 | 4 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL12 | 2 | 14,970 | 3 | 15 | 12,974 | 3 | 13 | 12,974 | 3 | 13 |
| | 4 | 3,992 | 4 | 4 | 3,992 | 4 | 4 | 3,992 | 3 | 4 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |

Table 3: Results of heuristic width restriction only.

| Program | Number of Registers | Depth = 2 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Width = 1 | | | Width = 2 | | | Width = 5 | | |
| | | Cost | Iters | Cost / Iter | Cost | Iters | Cost / Iter | Cost | Iters | Cost / Iter |
| LL1 | 2 | 13,200 | 3 | 33 | 12,800 | 3 | 32 | 12,800 | 3 | 32 |
| | 4 | 7,200 | 3 | 18 | 5,200 | 3 | 13 | 4,800 | 3 | 12 |
| | 8 | 1,200 | 2 | 3 | 1,200 | 2 | 3 | 1,200 | 2 | 3 |
| LL2 | 2 | 10,400 | 3 | 52 | 9,200 | 3 | 46 | 9,200 | 3 | 46 |
| | 4 | 1,600 | 3 | 8 | 1,400 | 3 | 7 | 1,400 | 3 | 7 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL3 | 2 | 10,000 | 3 | 10 | 9,000 | 3 | 9 | 9,000 | 3 | 9 |
| | 4 | 2,000 | 2 | 2 | 2,000 | 2 | 2 | 2,000 | 2 | 2 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL4 | 2 | 4,032 | 3 | 24 | 3,528 | 3 | 21 | 3,528 | 3 | 21 |
| | 4 | 1,680 | 4 | 10 | 1,680 | 3 | 10 | 1,680 | 3 | 10 |
| | 8 | 168 | 2 | 1 | 168 | 2 | 1 | 168 | 2 | 1 |
| LL5 | 2 | 19,256 | 3 | 58 | 17,264 | 3 | 52 | 17,264 | 3 | 52 |
| | 4 | 6,640 | 3 | 20 | 5,312 | 3 | 16 | 5,312 | 3 | 16 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL6 | 2 | 19,647 | 3 | 59 | 17,649 | 3 | 53 | 17,649 | 3 | 53 |
| | 4 | 5,661 | 4 | 17 | 5,661 | 4 | 17 | 5,661 | 3 | 17 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL7 | 2 | 5,760 | 3 | 48 | 5,760 | 3 | 48 | 5,760 | 3 | 48 |
| | 4 | 2,040 | 4 | 17 | 2,160 | 4 | 18 | 2,040 | 3 | 17 |
| | 8 | 360 | 2 | 3 | 360 | 2 | 3 | 360 | 2 | 3 |
| LL8 | 2 | 5,605 | 3 | 295 | 4,731 | 3 | 249 | 4,598 | 3 | 242 |
| | 4 | 2,090 | 3 | 110 | 1,995 | 3 | 105 | 1,995 | 3 | 105 |
| | 8 | 836 | 3 | 44 | 836 | 3 | 44 | 779 | 3 | 41 |
| LL9 | 2 | 5,100 | 3 | 51 | 5,100 | 3 | 51 | 5,000 | 3 | 50 |
| | 4 | 1,600 | 3 | 16 | 1,200 | 3 | 12 | 1,200 | 3 | 12 |
| | 8 | 100 | 2 | 1 | 100 | 2 | 1 | 100 | 2 | 1 |
| LL10 | 2 | 6,200 | 3 | 62 | 6,200 | 3 | 62 | 6,200 | 3 | 62 |
| | 4 | 3,700 | 3 | 37 | 2,400 | 3 | 24 | 2,400 | 3 | 24 |
| | 8 | 600 | 3 | 6 | 500 | 3 | 5 | 400 | 3 | 4 |
| LL11 | 2 | 13,972 | 3 | 14 | 12,974 | 3 | 13 | 12,974 | 3 | 13 |
| | 4 | 3,992 | 3 | 4 | 3,992 | 3 | 4 | 3,992 | 3 | 4 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| LL12 | 2 | 13,972 | 3 | 14 | 12,974 | 3 | 13 | 12,974 | 3 | 13 |
| | 4 | 3,992 | 3 | 4 | 3,992 | 3 | 4 | 3,992 | 3 | 4 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |

Table 4: Results of depth restriction = 2.

| Program | Number of Registers | Depth = 3 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Width = 1 | | | Width = 2 | | | Width = 5 | | |
| | | Cost | Iters | Cost / Iter | Cost | Iters | Cost / Iter | Cost | Iters | Cost / Iter |
| | 2 | 13,200 | 3 | 33 | 12,800 | 3 | 32 | 12,800 | 3 | 32 |
| LL1 | 4 | 6,000 | 3 | 15 | 4,800 | 3 | 12 | 4,800 | 3 | 12 |
| | 8 | 1,200 | 2 | 3 | 1,200 | 2 | 3 | 1,200 | 2 | 3 |
| | 2 | 10,000 | 3 | 50 | 9,000 | 3 | 45 | 9,000 | 3 | 45 |
| LL2 | 4 | 2,200 | 3 | 11 | 1,600 | 3 | 8 | 1,400 | 3 | 7 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| | 2 | 10,000 | 3 | 10 | 9,000 | 3 | 9 | 9,000 | 3 | 9 |
| LL3 | 4 | 2,000 | 2 | 2 | 2,000 | 2 | 2 | 2,000 | 2 | 2 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| | 2 | 3,528 | 3 | 21 | 3,696 | 3 | 22 | 3,528 | 3 | 21 |
| LL4 | 4 | 1,680 | 4 | 10 | 1,680 | 3 | 10 | 1,680 | 3 | 10 |
| | 8 | 168 | 2 | 1 | 168 | 2 | 1 | 168 | 2 | 1 |
| | 2 | 19,256 | 3 | 58 | 17,264 | 3 | 52 | 17,264 | 3 | 52 |
| LL5 | 4 | 5,312 | 4 | 16 | 5,312 | 4 | 16 | 5,312 | 4 | 16 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| | 2 | 18,981 | 3 | 57 | 17,649 | 3 | 53 | 17,649 | 3 | 53 |
| LL6 | 4 | 5,994 | 3 | 18 | 5,661 | 3 | 17 | 5,661 | 3 | 17 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| | 2 | 5,760 | 3 | 48 | 5,760 | 3 | 48 | 5,760 | 3 | 48 |
| LL7 | 4 | 2,040 | 3 | 17 | 2,040 | 3 | 17 | 2,040 | 3 | 17 |
| | 8 | 360 | 2 | 3 | 360 | 2 | 3 | 360 | 2 | 3 |
| | 2 | 4,655 | 3 | 245 | 4,636 | 3 | 244 | 4,522 | 3 | 238 |
| LL8 | 4 | 2,033 | 3 | 107 | 1,957 | | 103 | 1,881 | 3 | 99 |
| | 8 | 817 | 3 | 43 | 760 | 3 | 40 | 760 | 3 | 40 |
| | 2 | 5,000 | 3 | 50 | 5,000 | 3 | 50 | 5,000 | 3 | 50 |
| LL9 | 4 | 1,300 | 3 | 13 | 1,200 | 3 | 12 | 1,200 | 3 | 12 |
| | 8 | 100 | 2 | 1 | 100 | 2 | 1 | 100 | 2 | 1 |
| | 2 | 6,200 | 3 | 62 | 6,200 | 3 | 62 | 6,200 | 3 | 62 |
| LL10 | 4 | 2,200 | 4 | 22 | 2,100 | 4 | 21 | 2,400 | 3 | 24 |
| | 8 | 500 | 3 | 5 | 500 | 3 | 5 | 400 | 3 | 4 |
| | 2 | 12,974 | 3 | 13 | 12,974 | 3 | 13 | 12,974 | 3 | 13 |
| LL11 | 4 | 6,986 | 3 | 7 | 3,992 | 3 | 4 | 3,992 | 3 | 4 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |
| | 2 | 12,974 | 3 | 13 | 12,974 | 3 | 13 | 12,974 | 3 | 13 |
| LL12 | 4 | 6,986 | 3 | 7 | 3,992 | 3 | 4 | 3,992 | 3 | 4 |
| | 8 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 |

Table 5: Results of Depth Restriction = 3.

| Program | Number of Registers | Gnu gcc | | Heuristic BB-OPT-LOOP Depth = 1, Width = 2 | |
|---|---|---|---|---|---|
| | | Dynamic Spill Cost | Spills per Iteration | Dynamic Spill Cost | Spills per Iteration |
| LL1 | 4 | 7,600 | 19 | 5,600 | 14 |
| | 8 | 4,800 | 12 | 1,200 | 3 |
| LL2 | 4 | 4000 | 20 | 1,600 | 8 |
| | 8 | 2,600 | 13 | 0 | 0 |
| LL3 | 4 | 8,000 | 8 | 2,000 | 2 |
| | 8 | 8,000 | 8 | 0 | 0 |
| LL4 | 4 | 2016 | 12 | 1,680 | 10 |
| | 8 | 1,344 | 8 | 168 | 1 |
| LL5 | 4 | 9,628 | 29 | 5,312 | 16 |
| | 8 | 5,644 | 17 | 0 | 0 |
| LL6 | 4 | 8,991 | 27 | 5,661 | 17 |
| | 8 | 6,327 | 19 | 0 | 0 |
| LL7 | 4 | 4,680 | 39 | 2,160 | 18 |
| | 8 | 2,160 | 18 | 360 | 3 |
| LL8 | 4 | 2,907 | 153 | 2,109 | 111 |
| | 8 | 1,444 | 76 | 855 | 45 |
| LL9 | 4 | 2,700 | 27 | 1,200 | 12 |
| | 8 | 1,600 | 16 | 100 | 1 |
| LL10 | 4 | 5,700 | 57 | 2,100 | 21 |
| | 8 | 2,100 | 21 | 500 | 5 |
| LL11 | 4 | 6,986 | 7 | 3,992 | 4 |
| | 8 | 5,988 | 6 | 0 | 0 |
| LL12 | 4 | 6,986 | 7 | 3,992 | 4 |
| | 8 | 5,988 | 6 | 0 | 0 |

Table 6: Comparison of results between heuristic BB-OPT-LOOP with $depth = 1, width = 2$ and Gnu C.

Gcc was configured to produce code for the SPARC architecture. Further, the register allocation module was modified so that gcc would produce code for four and eight physical registers[8]. Table 6 summarizes the results of the code produced by gcc as well as a comparison with our heuristic algorithm. We have arbitrarily selected to compare gcc with our heuristic parameters of $depth = 1$ and $width = 2$ due to its acceptable performance/execution-time trade-off.

[8] Gcc produced an internal compiler error when the real register count was set to two.

| Method | CPU Time | | | Average Number of Spills Per Iteration | |
|---|---|---|---|---|---|
| | Ave. | Min. | Max. | 4 Regs. | 8 Regs. |
| Graph Coloring Gcc | 0.06 secs | 0.04 secs | 0.11 secs | 33.8 | 17.7 |
| BB-OPT | 600.0 secs | 345.0 secs | 2700.0 secs | 10.8 | — |
| BB-OPT-LOOP | 1800.0 secs | 480.0 secs | 10 hrs. | 9.3 | — |
| Heuristic BB-OPT-LOOP | | | | | |
| Depth=1, Width=1 | 0.08 secs | 0.06 secs | 0.34 secs | 9.8 | 4.7 |
| Depth=1, Width=2 | 0.13 secs | 0.07 secs | 0.48 secs | 9.7 | 4.8 |
| Depth=1, Width=5 | 0.34 secs | 0.11 secs | 0.57 secs | 9.8 | 4.5 |
| Depth=2, Width=1 | 0.11 secs | 0.08 secs | 0.54 secs | 11.0 | 4.8 |
| Depth=2, Width=2 | 0.23 secs | 0.13 secs | 0.65 secs | 9.4 | 4.7 |
| Depth=2, Width=5 | 0.39 secs | 0.15 secs | 0.70 secs | 9.3 | 4.4 |
| Depth=3, Width=1 | 0.24 secs | 0.16 secs | 0.72 secs | 11.0 | 4.7 |
| Depth=3, Width=2 | 0.47 secs | 0.23 secs | 0.89 secs | 9.4 | 4.4 |
| Depth=3, Width=5 | 0.93 secs | 0.33 secs | 1.12 secs | 9.3 | 4.3 |

Table 7: Execution times of the various methods.

# 7 Use in a Production Compiler

The allocation model used here is realistic in the sense that virtual registers can be created for the constants, local and global variables, and temporary expressions found in the intermediate code and later assigned (or allocated) to real, physical registers in a separate manipulation of the intermediate code representation. Because this model has been shown to be successfully implemented [18, 21, 10, 9], the integration of our register allocation strategy into the real-to-virtual register binding phase is straightforward. Therefore, it is possible to incorporate this algorithm into a production compiler. As such, the optimization level parameter to the compiler can control the heuristic width and depth used. In order to assess this possibility, we have noted the average execution times for various heuristic width and depth combinations which were investigated on the stated benchmarks. Table 7 summarizes execution times for the various allocation schemes. To provide a fair comparison, results for Livermore loops seven, eight and ten were omitted from the heuristic averages for four real registers as results for these loops were not available in all cases. The results indicate that while the brute-force approach, even for basic blocks is likely to be too expensive, the heuristic BB-OPT-LOOP is efficient enough to be practical.

On average, the execution time of our heuristic algorithm is twice that of the Gnu gcc algorithm. However, the quality in the spill code produced by our method results in a savings of a factor of five in memory references due to spill code. Similar results are evident in the comparison of the

quality of the spill code produced with $depth = 1$ and $width = 1$ and Gcc spill code, while the average execution time is comparable to the execution time of Gcc's allocator. As the search space is expanded, the allocation derived from BB-OPT-LOOP improves with some increase in the running time (which is still likely to be within acceptable limits).

# 8 Conclusion

The problem of register allocation has long been a subject of much research. Good register allocation serves to minimize the memory traffic during the execution of a program, so it is quite natural to devote a moderate amount of computation by a compiler to determine some such allocation. The earliest work in determining the optimal allocation of registers to data items focused on registers which were used only as index registers [15]. Work was done later which served to extend those ideas to allocating index registers to loops [17]. More recently, research was done that extended the original model in [15] to that of general purpose registers [16]. This paper continues in that tradition by extending the work in [15, 16, 17] to allocation of general purpose registers to loops and answers the long standing question of whether it is possible to, in principle, achieve optimal (minimal) spill code in loops.

We have presented an algorithm which demonstrates that it is possible to determine an optimal allocation of registers to data items in loops. However, this algorithm is computationally infeasible for all but the shortest loops, especially as the current trend in high-speed architecture is to execute enlarged basic blocks. In order to reduce the computational complexity of the algorithm, we identified a heuristic which serves to restrict the width of the allocation tree, and demonstrated it for some Livermore Kernels. Further, the fast execution time of our heuristic is practical for production compilers, while its results are superior.

# A Pseudo-code for Heuristic BB-OPT and BB-OPT-LOOP

```
Function BB-OPT-HEUR ( REGS : Initial register configuration;
                       RA : Register access pattern;
                       K : number of iterations;
                       W : heuristic width of tree;
                       D : heuristic look-ahead depth)
Begin
   Set current register state to REGS
   Set curr_depth to 0
   Foreach virtual register access V in RA do
      Foreach state N in the current register state set do
         If V is in state N then
            Add N to new_state set
         Otherwise
            Foreach real register R in N do
               Create N', an exact duplicate of N
               Replace virtual register, V', currently in R with V
               Cost(N') = Load-Cost(V) + Store-Cost(V') + Cost(N)
               Note that N is the parent of N'
               Add N' to new state set
            Enddo
         Endif
      Enddo
      Set current register state set to new_state set
      Set curr_depth to curr_depth + 1
      If curr_depth = D then
         Set current register state set to
         Set curr_depth to 0
      Endif
   Enddo
   Return new_state set
End BB-OPT-HEUR
```

## References


[1] A. H. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley. Reading, MA., 1974.

[2] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill Code Minimization Techniques for Optimizing Compilers. *Proceedings of SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, January 1989.

[3] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(4), April 1979.

[4] P. Briggs. *Register Coloring via Graph Coloring.* PhD thesis, Rice University, April 1992.

[5] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *Proceedings of SIGPLAN Conf. on Prog. Lang.*, June 1989. Portland, Oregon.

[6] G. Chaitin. Register Allocation and Spilling Via Graph Coloring. *Proc. of SIGPLAN Symposium on Compiler Construction*, 17(6), June 1982.

[7] G. Chaitin, M. Auslander, A. Chandra, J. Coocke, M. Hopkins, and P. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6, January 1981.

[8] F. Chow and J. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.

[9] J. W. Davidson and C. W. Fraser. Register Allocation and Exhaustive Peephole Optimization. *Software—Practice and Experience*, 14(9), September 1984.

[10] W. H. E. Day. Compiler Assignment of Data Items to Registers. *IBM Systems Journal*, 9(4), 1970.

[11] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985.

[12] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Comp.*, C-30(7), July 1981.

[13] R. A. Freiburghouse. Register Allocation Via Usage Counts. *Communications of the ACM*, 17(11), November 1974.

[14] T. Granlund and R. Kenner. Eliminating Branches using a Superoptimizer and the GNU C Compiler. *Proceedings of SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, 27(7), July 1992.

[15] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index Register Allocation. *Journal of the ACM*, 13(1), January 1966.

[16] W. Hsu, C. Fischer, and J. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, 15(10), October 1989.

[17] K. Kennedy. Index Register Allocation in Straight Line Code and Simple Loops. In R. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[18] B. W. Leverett. *Register Allocation in Optimizing Compilers.* PhD thesis, Carnegie-Mellon University, February 1981.

[19] F. Luccio. A Comment on Index Register Allocation. *Communications of the ACM*, 10(9), September 1967.

[20] S. Melvin and Y. Patt. Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques. *Proc. of the 18th Annual Int. Symp. on Comp. Arch.*, 19(3), May 1991. Toronto, Canada.

[21] R. M. Stallman. Using and Porting Gnu CC. *Free Software Foundation*, November 1992.