Abstract—Embedded systems are under heavy memory constraints. In order to lower the memory footprint of programs, we propose to compact the stack by overlaying variables. Variables that don’t interfere with each other are potential candidates for an overlaying. Our heuristics consist of two parts. The first heuristic chooses a next variable for allocation, the second heuristic then chooses an actual memory location for it. We derive our heuristics mostly from an already well-researched field, register optimization, and have tuned them towards memory compaction.

We can either overlay all variables—except certain variables declared in recursive subprograms—in a globally allocated memory pool, or we can overlay local variables within the subprogram’s activation record. The advantage of the first approach is a broader choice of candidates for an overlaying, and the disadvantage is the conservative approximation of the call graph, which introduces conflicts between variables of different subprograms that may not exist at run-time. For the second approach no approximation of the call graph is necessary. The dynamic creation and destruction of activation records (including the compacted variables) during run-time ensure that only required variables are held in memory.

I. INTRODUCTION

The vast majority of all microprocessors produced every year are built into embedded systems. Cell phones, digital cameras, washing machines, and other products with embedded systems are mass produced in hundreds of thousands, and the cost of a single microprocessor becomes relevant.

Nowadays 8-, 16-, 32-, and sometimes even 64-bit microprocessors are being used in embedded systems. However it is appealing to use just 8- or 16-bit microprocessors for their better price performance ratio. Increasing demands for performance, and memory size lay the burden on the compiler to make this feasible.

Limited memory, for instance, challenges compiler writers to deal with memory issues. Conventional compilers allocate for each variable a separate memory location. We propose to overlay variables whose live ranges do not overlap to compact the stack.

We derive our approaches from an already well-researched field, register allocation. Register allocation can be seen as a graph coloring problem. A coloring of a graph is defined as an assignment of different colors to adjacent nodes. Usually one is interested in either a \( k \) coloring or a minimal coloring. In this analogy, visualized in Fig. [1], nodes refer to variables and available colors refer to available registers.

Variables with interfering live ranges can’t be assigned to the same register and must be connected in this analogy by an edge. On the other hand, variables with non-interfering live ranges can be safely assigned to the same register.

Finding a \( k \) coloring for a \( k \)-register machine is known to be NP-complete [1], and thus not practically realizable. In the compiler literature one can find two major approaches for coloring graphs “nearly optimal”: Chaitin’s decomposition algorithm [2] [3] and Chow’s priority based algorithm [4] [5].

Memory allocation can not only be seen in the conventional way as an assignment of variables to disjoint memory locations, say in order of declaration, but also as a graph coloring problem. Variables that don’t interfere with each other can be assigned to the same memory location in order to minimize the necessary space.

The objective of register allocation is to find a \( k \) coloring for a \( k \)-register machine. The objective of memory allocation might be to either find a minimal coloring or a \( k \) coloring for a \( k \)-byte memory machine.

For embedded systems with only one program in execution, one might be satisfied with a \( k \) coloring, but for more general computing systems with several programs simultaneously in execution and competing for resources like memory, a minimal coloring might be more appropriate. We discuss in this paper how to find a minimal coloring for memory compaction. While register allocation colors variables of uniform size, the constraints for memory allocation are many-fold. Variables of different sizes, ranging from one byte characters to large arrays, need to be colored for machines with alignment restrictions.

Lia et al. proposed in [6] and [7] to make more use of auto-increment and decrememnt instructions by changing the memory layout. Variables referenced sequentially in time should be placed in consecutive memory locations. Zhuang et al. [8] extended this approach by also overlaying variables. Their overlaying strategy is tailored to maximize the utilization of auto increment and decrement instructions. We, on the other hand, tailor our heuristics to overlay variables as aggressively as possible.

We present in the next section Chow’s work on
register allocation \cite{4} \cite{5}. We then add proposals how to apply this technique for memory compaction. Most register allocation heuristics only choose a next node to color and leave the choice of the particular color open. We developed some heuristics to address this issue. Section \[III\] discusses differences between global and local overlaying strategies, e.g. overlayings of all variables—except certain variables declared in recursive subprograms—in a global memory pool or overlayings of local variables within the subprogram’s activation record. Section \[IV\] discusses results of our implementation, and section \[V\] concludes this paper.

II. PRIORITY BASED GRAPH COLORING

The first implementation of a graph coloring register allocator based on graph decomposition was presented by Chaitin et al in 1981 \cite{3}. Some refinements have been published later by Chaitin alone \cite{2} and Briggs suggested further improvements \cite{9}.

An alternative graph coloring register allocator is based on a priority queue and was proposed by Chow in \cite{4} and \cite{5}.

We introduce in subsection \[II-A\] Chow’s register allocator. In subsection \[II-B\] we apply Chow’s ideas to memory compaction and in subsection \[II-C\] we propose allocation heuristics for assigning proper colors to nodes.

For the sake of brevity, we don’t discuss Chaitin’s register allocator nor do we discuss Chaitin based memory compaction.

A. Graph Coloring for Registers

All variables are partitioned into two sets. The constrained set contains all variables with a degree higher than the number of available registers, and the unconstrained set contains all remaining variables.

Colors can be always found for variables in the unconstrained set, since the number of adjacent nodes is smaller than the number of available colors. It’s better to color these variables after all constrained variables are colored so that the constrained variables aren’t constrained by any unconstrained ones. On the other hand, constrained variables might be also colorable if enough adjacent nodes share the same color.

The algorithm orders the constrained variables by the expected savings if a variable resides in a register instead in memory. The highest prioritized variable is to be first assigned some available color. If no color is available, then the variable’s live range is split. Splitting might change the degree of other variables, and both sets must be recomputed.

The expected savings are calculated by a priority function that computes the penalties of moving a variable between register and memory, weighted by the estimated execution frequency to speed up the execution. This does not directly apply to memory compaction, but the general idea of prioritizing variables can still be used.

B. Graph Coloring for Memory

This section applies Chow’s ideas to memory compaction. We developed three heuristics for prioritizing nodes. The node with the highest priority is selected next for coloring, and then an allocation heuristic from section \[II-C\] assigns to this node a specific color. To make this section independent from section \[II-C\] we have chosen most of the examples so that there is never a choice of two or more colors for a node.

We assume an initial memory pool of a certain size $k_{max} \geq 0$. $k_{max}$ can be initialized with a pessimistic guess of the memory requirements, or even with zero. If a variable doesn’t fit in the available memory, we have to adjust the current memory size $k_{max}$.

1) Prioritizing by Degree: We can prioritize highly constrained variables, e.g. we prioritize variables by their degree. It is hard to find space for highly constrained variables. We try to resolve this by coloring them first. Afterwards it shouldn’t be hard to find free space for the less constrained variables.

There exist no nontrivial loop-free graph for which all nodes have different degrees.\footnote{Proof by contradiction: If \( n \) is the highest degree, then the corresponding node must have \( n \) adjacent nodes and the graph has at least \( n+1 \) nodes. But since \( n \) is the highest degree, there can be at last \( n-1 \) further nodes with degrees between \( 1 \) and \( n \).} This makes prioritizing by degree in any case nondeterministic, and the algorithm must choose an order among nodes of equal degree, say by applying another priority heuristic from this subsection. The conflict graph in Fig. 2 has two nodes of degree two and two nodes of degree one.

To see the advantage of prioritizing by degree, we first give variables with lower degrees a higher priority:
Fig. 2. Prioritizing by Degree

removing first A and D and then B and C results in the left overlaying of Fig. 2. By starting with A and D we merge the two variables into one variable of degree two. This joint variable now has also a high degree, conflicts with many variables and limits the choice of the other highly constrained variables B and C, which need to be placed on separate space.

On the other hand, if we first allocate the highly constrained nodes B and C, then A and D remain independent with low degrees, and because of this they can be placed, as seen in the right overlaying of Fig. 2, on already allocated memory.

2) Prioritizing by Size: We can also give larger variables higher priorities. An order among variables of equal size must be chosen, say by applying another priority heuristic from this subsection.

Fig. 3. Prioritizing by Size

In the example of Fig. 3, variables A, B, C, D, and E are already allocated, and variables S and L are still to color. The conflict graph shows just the conflicts involving S and L. Allocating first the smaller variable S between A and B would force L to be allocated over B and E and significantly increasing the size of the memory pool (Fig. 4 top). On the other hand, allocating first L over C would allow S to be allocated to the right, also increasing the size of the memory pool (Fig. 4 bottom). In both scenarios a variable forces another to increase the available memory pool. However the impact on memory size of a larger variable forcing a smaller variable to increase the available memory is on average less than the other way around.

After all larger variables are allocated we hope to find for the smaller variables niches between the larger ones.

3) Prioritizing by Free Space: A more sophisticated heuristic is derived from [10]. Hendren et al color specially formed live range intervals (“cyclic interval graphs”) for register allocation. The less colors available for an interval, the sooner a color should be assigned to it. A color is available for a certain interval, only if it hasn’t already been used for any other overlapping interval.

For memory compaction, we have to compute for each variable the number of non-conflicting locations. This number is anti-proportional to the priority of the heuristic, e.g. the lower the number of non-conflicting locations the higher the priority. An order must be chosen among variables of equal non-conflicting locations, say by applying another priority heuristic from this subsection.

We want to emphasize that Hendren et al color temporal values, e.g. time intervals of live ranges, but we, on the other hand, color geometrical values, e.g. sizes of variables in memory. This heuristic is attractive and very beautiful in conjunction with the allocation heuristic from subsection II-C.2.

Fig. 4. Prioritizing by Size

Fig. 5. Prioritizing by Free Space
and \( C \) fits into two locations (the indices distinguish all possible allocations). Further allocations will limit conflict-free locations. We can probably delay the allocation of \( A \), and to a certain extent also delay the allocation of \( B \), but we should not delay the allocation of \( C \) because \( C \) fits into the fewest memory locations.

### C. Allocation Heuristics

Chow addresses which node to choose next, but not which color to assign. We developed some heuristics for finding a color and present them in the next subsections.

1) First, Next, Best, and Worst Fit: Operating systems need to assign memory to processes. This similarity makes it worth looking at the standard allocation strategies of operating systems: first, next, best, and worst fit.

It turns out that only first fit can be used due to a fundamental difference. For memory compaction, the conflict-free memory locations change dynamically. Each variable is in conflict with different variables and is therefore allocatable at different locations. After the allocation of some variable, the next variable has a totally different view of conflict and conflict-free memory locations. Only first fit is (essentially) invariant to this difference. It always allocates the first available memory block to the next variable, e.g. the one starting at the lowest address. It doesn't really matter that different variables "see" different first free memory blocks.

The need for heuristics tailored to memory compaction becomes necessary. The next subsection discusses such a heuristic.

2) Least Pressured Fit: We use a register allocation heuristic from [10] and apply it to memory compaction. Hendren et al choose for register allocation the register that can be used for the fewest yet unallocated variables.

To make this plausible we continue with the example from subsection II-B.3 (Fig. 5). As before, we suppose we still have to color variables \( A \), \( B \), and \( C \), and the next candidate is \( C \). We see that \( C_1 \) would eliminate one allocation (A4), and that \( C_2 \) would eliminate two allocations (A1 and B1). The memory block under \( C_1 \) can be used for the fewest potential further assignments, and is therefore selected for \( C \).

The idea behind this heuristic is to prefer memory locations eliminating as few possible allocations of further variables as possible.

### III. GLOBAL CONFLICT GRAPH — LOCAL CONFLICT GRAPHS

One has the choice of generating one conflict graph over the whole program or one conflict graph per subprogram.

A global conflict graph will result in one globally allocated memory pool for compacting all variables. Local conflict graphs will result in a separate compaction of local variables within each activation record.

Variables of recursive subprograms cannot be overlaid with other variables if they are alive at the recursive call, because several instances will exist at run-time. These variables must be compacted locally within the activation record, even in the global scheme. We denote these variables as recursive variables and all other variables as non-recursive variables.

#### A. Global Conflict Graph

A conflict graph can be described by a conflict matrix, as in Table [I] and Table [II] and describes the conflicts, e.g. overlapping live ranges, among variables.

We observe that some recursive variables are not in conflict with variables from other subprograms, or are in conflict with global variables. By using this conflict matrix a coloring routine might (mistakenly) choose exactly such a pair of variables for an overlaying. To OR the conflict matrix with the bit-matrix of Table [II] prevents such a choice. This bit-matrix introduces artificial conflicts between recursive variables and all others. A coloring routine working on the resulting bit-matrix will not be able to overlay recursive variables anymore.

In the later code generation phase, the compiler has to check each variable for being recursive or non-recursive. Non-recursive variables get addresses from the global address space, and recursive variables get addresses within the activation record.

#### B. Local Conflict Graphs

Another possibility is to OR the conflict matrix with the bit-matrix from Table [II] This bit-matrix prevents all overlayings of local variables from different subprograms and only allows overlayings within subprograms. The same coloring routine can still be used.

The idea of this approach is to keep the natural overlaying mechanism of the program’s run-time environment. Upon entering a subprogram, an activation record (including the compacted local variables) is created. After leaving the subprogram, the activation record (again, including the compacted local variables) will be destroyed.

### IV. PERFORMANCE

We have implemented the Chow-based heuristics from section II in the Bauhaus environment [11] as a C to C translator. As a consequence, we do not consider temporaries generated by the compiler. Because compiler-generated temporaries are of short life, it is easy to overlay them with other variables.

Table [III] and Table [IV] show our results. Table [III] contains static measurements. The size field denotes the sum of all variablesizes and the compaction field denotes the sum of all variablesizes after compaction. Table [IV] contains dynamic measurements at run-time. The size field denotes the maximum heap size of the uncompressed program and the compaction field...
TABLE I

<table>
<thead>
<tr>
<th>Overlaying Size</th>
<th>Compaction Method No 32 bytes</th>
<th>27%</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bytes</td>
<td>32 bytes</td>
<td>27%</td>
</tr>
<tr>
<td>27%</td>
<td>18%</td>
<td>27%</td>
</tr>
<tr>
<td>11.2%</td>
<td>11.2%</td>
<td>11.2%</td>
</tr>
<tr>
<td>45.5%</td>
<td>45.5%</td>
<td>45.5%</td>
</tr>
<tr>
<td>16.3%</td>
<td>16.3%</td>
<td>16.3%</td>
</tr>
<tr>
<td>115 bytes</td>
<td>115 bytes</td>
<td>115 bytes</td>
</tr>
<tr>
<td>61.7%</td>
<td>61.7%</td>
<td>61.7%</td>
</tr>
<tr>
<td>187 bytes</td>
<td>187 bytes</td>
<td>187 bytes</td>
</tr>
<tr>
<td>44 bytes</td>
<td>44 bytes</td>
<td>44 bytes</td>
</tr>
<tr>
<td>9.5%</td>
<td>9.5%</td>
<td>9.5%</td>
</tr>
<tr>
<td>152 bytes</td>
<td>152 bytes</td>
<td>152 bytes</td>
</tr>
<tr>
<td>61.7%</td>
<td>61.7%</td>
<td>61.7%</td>
</tr>
</tbody>
</table>

g1: global variables; a1 and b1: local variables from two non-recursive subprograms; r1 and s1: local variables from one recursive subprogram (r1 are recursive and s1 are non-recursive).

TABLE II

<table>
<thead>
<tr>
<th>Overlaying Size</th>
<th>Compaction Method No 102 bytes</th>
<th>20.3%</th>
</tr>
</thead>
<tbody>
<tr>
<td>102 bytes</td>
<td>102 bytes</td>
<td>20.3%</td>
</tr>
<tr>
<td>4.9%</td>
<td>4.9%</td>
<td>4.9%</td>
</tr>
<tr>
<td>615 bytes</td>
<td>615 bytes</td>
<td>615 bytes</td>
</tr>
<tr>
<td>45.5%</td>
<td>45.5%</td>
<td>45.5%</td>
</tr>
<tr>
<td>18%</td>
<td>18%</td>
<td>18%</td>
</tr>
<tr>
<td>152 bytes</td>
<td>152 bytes</td>
<td>152 bytes</td>
</tr>
<tr>
<td>9.5%</td>
<td>9.5%</td>
<td>9.5%</td>
</tr>
<tr>
<td>152 bytes</td>
<td>152 bytes</td>
<td>152 bytes</td>
</tr>
<tr>
<td>61.7%</td>
<td>61.7%</td>
<td>61.7%</td>
</tr>
</tbody>
</table>

g1: global variables; a1 and b1: local variables from two non-recursive subprograms; r1 and s1: local variables from one recursive subprogram (r1 are recursive and s1 are non-recursive).

TABLE III

<table>
<thead>
<tr>
<th>Static Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>pipeline</td>
</tr>
<tr>
<td>pipeline2</td>
</tr>
<tr>
<td>gradclass</td>
</tr>
</tbody>
</table>

Compaction method 1: prioritizing size + first fit; compaction method 2: prioritizing degree + first fit; compaction method 3: prioritizing free space + first fit; compaction method 4: prioritizing size + least pressured fit; compaction method 5: prioritizing degree + least pressured fit; compaction method 6: prioritizing free space + least pressured fit.

TABLE IV

<table>
<thead>
<tr>
<th>Dynamic Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>pipeline</td>
</tr>
<tr>
<td>pipeline2</td>
</tr>
</tbody>
</table>

Compaction method 1: prioritizing size + first fit; compaction method 2: prioritizing degree + first fit; compaction method 3: prioritizing free space + first fit; compaction method 4: prioritizing size + least pressured fit; compaction method 5: prioritizing degree + least pressured fit; compaction method 6: prioritizing free space + least pressured fit.
denotes the maximum heap size of the compacted program.

The larger programs, pipeline (68 variables, 21 subprograms including three recursive subprograms), and pipeline2 (46 variables, nine subprograms including two recursive subprograms), are two hardware pipeline simulation programs with rather sparse conflict matrices. Due to the sparseness, overlaying choices are trivial, and all overlaying methods are equally good. gradclass is a small program from an introductory programming book with only a few variables and with many conflicts in its conflict matrix. It is difficult to make the right choices and the performance of the individual methods differ. Prioritizing size appears to be inferior due to the fact that most variables are of equal size, and that the next variable for coloring is randomly chosen among them in our implementation. With our heuristics, the stack can be reduced by 43% to 61.7% statically and 3.7% to 20.3% dynamically. The big discrepancy for pipeline in Tables IV and V is due to the fact that pipeline contains many helper functions. Local variables of these helper functions make perfect candidates for overlayings and are responsible for the high compaction rate in Table IV. On the other hand, these helper functions are called sequentially in time and contribute almost nothing to the heap size in Table V. Better results can be expected when overlaying individual live ranges as our C to C translator overlays only whole variables.

![Call Graph](image)

We discussed in section III two different overlaying schemes, global overlaying and local overlaying. The disadvantage of the global scheme is the conservative approximation of the call graph. Fig. 6 gives an example. The line style distinguishes different execution paths. One path is established by the calling sequence f1→f2→f3, and the other path is established by the calling sequence f1→f2→f4. As we can see, f3 calls f4 only when called directly from f1, but the conflict graph, which is generated from the static control flow graph, also suggests an execution path from f1 over f2 to f3 and f4. This renders conflicts between the local variables of f2 and f4.

In the worst case, when there is no overlaying possible, the global variable space from the global scheme will be the sum of all variables. The local scheme would also yield no compaction within the activation records, but the (dynamic) execution of the program has the effect that at some points, there are just the activation records of f1 and f4 or f1, f2, and f3 allocated, but never the activation records altogether. The required maximum space is just the space of three but not four activation records.

On the other hand, the global scheme performs quite well if most paths in the call graph are actually executed, or if the conflict graph is rather sparse. The lack of conflicts increases the number of potential candidates, and overlayings over subprogram boundaries become possible.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have described how to efficiently compact memory based on register allocation techniques. We described and implemented heuristics to compact memory by a priority queue. All heuristics have been proven valuable, and the stack can be reduced by 43% to 61.7% statically and 3.7% to 20.3% dynamically. We have shown that a global compaction of all non-recursive variables does not always result in a better compaction than a local compaction of variables within the subprogram’s activation record.

In the future, we would like to collect data for standard benchmarks to fine-tune our heuristics. It may also be of interest to combine two or more heuristics into one super heuristic by calculating the priority of nodes as the (weighted) product of the priorities of the individual heuristics.

REFERENCES