International Journal of Parallel Programming, Vol. 32, No. 6, December 2004 (© 2004)

A Spill Code Minimization Technique—Application in the Metrowerks StarCore C Compiler

Virgil Palanciuc[1,2] and Dragoş Badea[1]

Graph coloring algorithms have been shown to be an efficient and effective means of performing register allocation. The power of these algorithms lies in their strong coloring heuristics and their ability to abstract away disparate allocation problems such as data-flow constraints, conforming to calling conventions, and target machine restrictions. However, even optimal algorithms cannot color every graph, and often some live ranges must be spilled to memory to make room for others. In this paper, we present a new approach of reducing spill code, which can be used to complement virtually any register allocation algorithm, and provides a good support to implement cheaper spill methods like spilling to another register (from a different class) and rematerialization (reloading the register from a constant or expression). This algorithm was partially implemented into the Metrowerks StarCore C compiler where it has proven its efficiency in terms of both cycle count and code size.

KEY WORDS: Optimization; register allocation; spill code; rematerialization.

1. INTRODUCTION

Global register allocation and spilling using graph-coloring techniques has been a topic of practical interest to compiler designers for a number of years. In all compilers that use such a technique, some sort of conflict graph is built whose vertices correspond to the variables and whose

[1] Motorola/Metrowerks Romania, Union Tower II Business Center, 5th floor, 010113, Bucharest 1, Romania. E-mail: {virgil.palanciuc; dragos.badea}@freescale.com
[2] To whom correspondence should be addressed.

0885-7458/04/1200-0475/0

edges represent the interference between the live areas of variables. The coloring of the vertices of this graph corresponds to an assignment of the variables to real machine registers. When the number of colors (i.e., real registers) is not sufficient, additional register-to/from-memory statements must be inserted, and these extra statements are referred to as spill code., Both problems (that of deciding whether a graph can be colored with the given number of registers and that of minimizing the amount of spill code) are computationally intractable. This paper presents an heuristic spill reduction algorithm that can be used to complement virtually any register allocation algorithm in order to reduce the amount of generated spill code and thus to produce more efficient (smaller and faster) compiled code. This algorithm has been developed in the context of the Metrowerks StarCore C Compiler, a highly optimizing compiler that currently targets the Motorola SC100 family: SC110, SC120, SC140 and SC140E DSP cores and also the platforms MSC8101 (single SC140 core) and MSC8102 ($4 \times$ SC140 cores). Consequently, the algorithm presented in this paper assumes an architecture with explicit parallelism, in which a set of atomic operations can be executed in parallel. Examples of such architectures are VLIW, VLES, EPIC architectures.

StarCore is based on a variable length execution set (VLES) model; in each cycle, an eight-word instruction set (called *fetch set*) is fetched from memory, and the hardware detects the portion of the fetch set (called *execution set*) that contains the actual set of instructions to be executed in parallel (actually, in the worst case an execution set may span over two fetch sets). Note that the parallel execution of several instructions has to be specified explicitly by the compiler/assembly programmer—that is, the execution sets are not formed at runtime, instead they are encoded directly in the object file.

There are two main register files in the SC architecture: one is the DALU register file that contains 16 40-bit data registers (D-class registers) for integer and fractional data operand storage; the other one is the AGU register file, which contains 16 32-bit address registers (R-class registers). Besides, StarCore also has available four address-offset registers and four modulo registers, one predication bit for conditional execution, delay slots, and hardware loop mechanisms. For a complete understanding of the StarCore features, you may check the SC140 reference manual.[1]

One of the most important decisions the register allocator has to take is which node should be spilled; many different heuristics are used to take this decision, but the idea behind all is the same—the node should not be used frequently, and spilling it to the stack should break as many conflicts as possible. Some register allocators insert some spill code to split the live range of a node into several live ranges, and then spill only one

live range of the node. Anyway, once the register allocator has determined which live ranges will be spilled, it must now decide where to place the spill code. The simplest and roughest technique is to insert a store after every definition of the live range and a load before every use. Although this spill-everywhere technique works, it usually generates much more spill code than is necessary. The existing register allocation methods have (usually local) spill elimination heuristics, to improve the quality of the code. Our spill code elimination heuristic is a global heuristic; it can be used for virtually any register allocation algorithm; it has a very good support for rematerialization; it requires a relatively small implementation effort and it generally produces very good results.

2. RELATED WORK

2.1. Chaitin's Spilling Heuristic

Chaitin's spilling heuristic [2] mentions several refinements to the simple spill-everywhere approach that can reduce the amount of spill code inserted for a given spill decision. These include recomputation of register values that can be done easily instead of transferring them to and back from memory and only inserting spill loads at the deaths of other live ranges where register resources are being made available.

2.2. Bernstein *et al.'s* Spill-Almost-Everywhere Heuristic

Bernstein's spill-almost-everywhere heuristic limits even more the amount of spill code inserted for a live range in a given basic block. With Bernstein's heuristic the number of loads and stores inserted into a basic block is at most one load/store per live range.[3] This in effect renames the live range for each basic block in which it is referenced, whereas Chaitin's heuristic may produce multiple new names in a given block.

2.3. Bergner's Interference Region Spilling

To enable the spilling of limited portions of a live range, Bergner introduces a new concept called the interference region (see Ref. 4). For two interfering live ranges, their interference region is defined to be the portion of the program where they are live simultaneously. By eliminating (spilling) this region from one of the live ranges through the addition of spill code, they will no longer be live simultaneously anywhere in the program, thus they will no longer interfere. In his PhD thesis, Bergner also presents a new live range splitting technique called interference region splitting, which goes further in the attempt to minimize the spill code.

3. OUR APPROACH

In this paper, we present another efficient approach to spill elimination. It was partially implemented on a hierarchical register allocator (which is a refinement of the algorithm presented by Callahan and Koblenz.[5] We argue that having a separate spill elimination step is generally a good idea, since it provides provisions for a cleaner implementation and one can design a spill elimination algorithm that has the same or better performance as the ones that are currently used in the register allocators. The results are presented in Section 4 and come to support this idea. In the remaining part of the paper, we will first introduce the basic ideas of the register allocation algorithm we used, and we will present our spill elimination algorithm, plus a few ideas on how to improve the results obtained by the register allocator.

3.1. Motivation

We implemented our spill elimination algorithm on top of a hierarchical register allocator—a slightly modified version of the algorithm presented by Callahan and Koblenz.[5]

Hierarchical register allocation is a graph-coloring algorithm that is sensitive to program flow structure and thus tries to place spill code in less frequently executed portions of the program. The choice of variables to spill is based on usage patterns between the spills and the reloads rather than usage patterns over the entire program. The method allows a variable to be assigned to one register over a portion of the program, memory in a second portion, and a different register in yet a third portion.

Profiling information can be trivially incorporated to improve the selection of spilled variables and the location of the spill code because all analysis is based on the probability of executing a particular basic block or flowing along a particular control flow edge.

The main idea is to represent the program's loop and conditional structure by a tree of tiles. Tiles are visited in a bottom-up fashion and a local interference graph is created and colored (using pseudoregisters) for each tile. A tile's local spill decisions are made based on local usage and coloring decisions are propagated upward into the tile tree. After the bottom-up pass has allocated variables to pseudoregisters for the entire tree, a top-down walk binds pseudoregisters to physical registers and introduces spill code where desirable and required, but not necessarily where the decision to spill was made. In order to minimize the amount of fix-up code between tiles, we propagate individual information on virtual register allocation upward and downward in the tile tree. In addition to better

spill code placement, this hierarchical approach also allows smaller conflict graphs to be constructed.

Although this register allocation algorithm has several important advantages, it has at least one important disadvantage: although it does split the live ranges at tile boundary, it does not split them inside a tile, thus occasionally ending up with very poor code, especially for large loops with high register pressure.

To fix this problem, we had the choice of adding an intra-tile live range splitting algorithm or adding a post-coloring spill reduction step. We did both and we noticed that (even though these algorithms sometimes complement each other very effectively), the spill reduction step brought far more performance than the intra-tile live range splitting algorithm, basically due to its two main features:

1. It has the ability to correct some of the mistakes made by the register allocation (wrong registers chosen for spilling and for live range splitting).
2. It has a strong support for replacing stack slots with cheaper spill locations (other registers, recomputing the value instead of reloading it etc.).

This observation lead us to the conclusion that the tile-based approach is generally enough for live range splitting, with only one notable exception that we will present in Section 3.5.

In order to use the spill reduction algorithm after doing hierarchical register allocation (or any other type of register allocation), one small change in the register allocator was made: a new class of 'special' virtual registers was created—we call them spill registers. The register allocation algorithm remains unchanged, except that now, when a virtual register is spilled, instead of inserting moves from and to memory, it inserts transfer instructions from and to the associated virtual spill register.

The spill reduction algorithm is activated after the allocator has colored all registers, the code has been changed to physical registers, and the only virtual registers remaining in the flow-graph are the registers in the spill class. The elimination of unnecessary spill code is done in two phases:

1. Gathering data and elimination of spill code.
2. Coloring spill registers.

The first step is actually an additional analysis step that computes equivalence information for all the program points (i.e., it detects the registers/immediate values/expressions that have the same value as a virtual spill register at any execution set of a program); based on this information, we can decide that some load/store instructions are useless because they do not have a significant impact on this equivalence information.

The second step attempts to 'color' the virtual spill registers—that is, to assign them to different registers. Note that even though we may occasionally have available registers from the same class (due to encoding constraints), generally these register will be registers from a different class and we will be able to use them only as spill locations (and not to perform the actual computation). If no registers are available, we may decide to use the equivalence information provided by the first analysis step in order to recreate the value. Finally, if we have no equivalence information we will end up creating a stack slot and replacing the spill register with that stack slot. Moreover, if stack consumption is a big issue, one may decide that several spill registers can share the same stack slot—but care should be taken while doing this, because although it does save stack space (and usually code size, too), it tends to increase the cycle count. This will be discussed in more detail in Section 3.5.

In the next sections, we will describe in detail the spill reduction algorithm, and we will provide a number of examples to illustrate how it works.

3.2. Phase One—Analysis and Spill Code Elimination

As mentioned earlier, in the following sections we are going to view the spill locations as a special class of virtual registers that will be referred as spill registers. We are going to talk about their lifetimes, about their preferences to get a certain color, and finally we are going to allocate them real registers. The difference between spill registers and usual virtual registers is that, in case no physical register is available, we are not going to further create other spill locations for them but just associate a stack slot to each spill register. In addition, if the spill registers' coloring is not successful we do not have to further insert spill code.

The first phase of the spill reduction is based on the computation of a dataflow analysis problem—we called it 'equivalence propagation'. It analyzes the content equivalence of each spill location with each physical register, with a constant or with a simple expression at any point in program. By 'simple expression', we mean only those expressions that can be represented by a single machine instruction. We impose this condition because keeping complex expressions would be impractical in terms of computational complexity; also it will usually not bring benefit, since we are interested in rematerialization of registers at small cost, no larger than that of a reload from a stack slot.

Based on this information provided by the analysis, we will be able to eliminate some unnecessary *load* and *store* instructions right from the first phase and provide enough information to support the second phase of the

algorithm in making decisions about coloring or rematerializing the spill registers at smaller costs.

This data analysis step that we are performing is actually a combination of three dataflow problems similar with *constant propagation*, *copy propagation*, and a somehow more sophisticated *available expressions* analysis. The difference is that by performing these three analysis steps together, we are able to gather more information as one could infer in one analysis based on the information provided by the other (the simplest example is the code sequence $r0 = 3\ r1 = 3$, where we can decide that $r0 = r1$ although there was no explicit assignment from $r1$ to $r0$.). Actually, the technique used in this first step can also be considered to be a particular combination of analyses and optimizations, and thus it could be described by combining the analysis frameworks, as described in Ref. 6.

The lattice we are working on has as elements arrays of sets. For each physical and each spill register we will reserve a position in the array; on this position, we keep a set representing the set of 'objects' equivalent to that particular register (where an object may be another register, a constant value, or an expression). Actually, we do not need to keep a set—we can represent, for a register *Reg* this set as a structure containing:

- the number that is the value in *Reg* register (of course, we must be able to assign two special values to that number—'NOT_A_CONST' and 'ANY_CONST', corresponding to the *bottom* and *top* elements of the 'integer constant propagation' (ICP) lattice, as defined in Ref. 7)
- a bitset representing the registers that are equivalent with *Reg* register
- a list containing the expressions equivalent with *Reg* register. As we already said, it is not useful (and it is practically impossible) to have any expression in this equivalence list—we considered that it is enough to keep the 'simple expressions'—that is expressions that can be recomputed using a single instruction, and thus can actually be represented by the instruction itself.[3] Note that here we will also have to keep two special values—ANY_EXPRESSION and NO_EXPRESSION corresponding to the *bottom* and *top* elements of the lattice. We could represent ANY_EXPRESSION by a NULL list, and NO_EXPRESSION by an empty list.

[3] One could keep any expression as the tree representation of it, but proving two expressions equivalent would imply very complex symbolic calculations, and it will still have a limited generality. Besides that, we are not interested in very complex expressions since the rematerialization cost would be too high and we will end up preferring the reload from stack, anyway.

On this lattice, we do a classical iterative dataflow analysis as presented in Ref. 7 with the values initialized to *bottom* (except the values for *entry*, which are initialized at *top*).

```
IterativeAnalysis(FlowGraph FG)
  {
    foreach basic_block B in FG do
    {
      if B!= Entry(FG) then
        Initialize(B.equiv_info);
      else
        InitializeEntry(B.equiv_info);
      AddElement(B, WorkList);
    }
  repeat
  {
      Initialize(temp);
      B= ExtractElement(WorkList);
      foreach basic_block P in Predecessors(B) do
        temp = Meet(temp, BlockFlowFunction(P,false));
      if ( temp != B.equiv_info) then
      {
        B.equiv_info = temp;
        foreach basic_block S in Successors(B) do
          AddElement(S,WorkList);
    }
  }untilEmpty(WorkList);
  } /* End of IterativeAnalysis */
```

```
EquivInfo Meet( EquivInfo e1, EquivInfo e2)
{
  foreach register R do
  {
  Result.Reg[R].Const= meet_constants(e1.Reg[R].Const, e2.Reg[R].Const);
  Result.Reg[r].Regs = BitsetAnd(e1.Reg[R].Regs, e2.Reg[R].Regs);
  }
  foreach register R do
    Result.Reg[R].Exp = meet_expressions(e1.Reg[R].Exp, e2.Reg[R].Exp);
  return Result;
}
```

```
Initialize(EquivInfo e)
  {
    foreach regsiter R do{
      e.Reg[R].Const = ANY_CONST;
      BitsetFill(1,e.Reg[R].Regs);
      e.Reg[R].Exp= ANY_EXPRESSION;
  }
}
InitializeEntry(EquivInfo e)
{
  foreach regsiter R do{
    e.Reg[R].Const = NOT_A_CONST;
    BitsetFill(0,e.Reg[R].Regs);
    e.Reg[R].Exp= NO_EXPRESSION;
  }
}
```

Notice that during the analysis we call the block flow function with a second parameter 'false'. This is because we use the same function to eliminate the useless spill instructions with the second parameter set to 'true'.

The meet operator we used relies on the classical meet operators for ICP and for bit vectors. In the case of expressions, lattice and the meet operator is somehow similar with the one used for constant propagation—here we have $e1 \wedge e2 == e1$ if $e1$ and $e2$ produce the same result (assuming neither $e1$ or $e2$ have the special values of *top* or *bottom*).

The equivalence propagation analysis mostly provides us information about which load instructions are to be removed and then we can decide to remove store instructions that are useless (they actually become dead code). This happens because usually load instructions can exist in equivalence points, where $r_j == s_k$. (s denotes a spill register and r denotes a physical register). Notice that stores usually come either after definitions of physical registers (which invalidate possible equivalence relations existent up to that point) or after redefinitions that update the old value. Thus, it is unlikely to directly discover useless store instructions—but here it is an example (Fig. 1) in which both the redefinition (mac) and the store instruction can be removed.

Anyway, this is an exception—store instructions are usually seen as necessary in the first place and they may eventually become useless after removing some reloads (if we manage to remove all the loads exposed to a store).

Consider the example in Fig. 2, in which variable x is spilled at a certain point in register allocator. The temporaries $t1$, $t2$, and $t3$ are created

```
add r1,r2,r3
store r3,s1
[...]
load s1,r4
mac r5,r6,r4 /* equiv_info:{(r5==0)}*/
store r4,s1
[...]
```

Fig. 1. Both redefinition and store can be removed.

def x use x use x	def t1 store t1, s1 load s1, t2 use t2 load s1, t3 use t3	i1. def r1 i2. store r1, s1 i3. load s1, r2 i4. use r2 /*no redef of r2*/ i5. load s1, r2 /* unnecessary load */ i6. use r2

Fig. 2. Code sequence containing a useless load.

and let us assume that real registers can be assigned to them. If $t2$ and $t3$ get the same physical register ($r2$) and there is no other rewrite instruction on $r2$ between instructions $i3$ and $i6$, then the values in $s1$ and $r2$ are identical before instruction $i6$. As result, we can optimize this piece of code and eliminate the load in $i5$.

In Fig. 3, we have an example in which store instructions can be eliminated. Assume variable x is dead at block exit. If $t3$ and $t4$ were given the same color ($r3$), the dataflow analysis will see $s_1 == r_3$ at point $i8$ in program. The algorithm goes on with removal of the load $i8$, which used to

def x use x use x & def x use x /*last use of x*/	def t1 store t1, s1 load s1, t2 use t2 load s1, t3 use t3 & write t3 store t3, s1 load s1, t4 /*last use of s1*/ use t4	i1. def r1 i2. store r1, s1 i3. load s1, r2 i4. use r2 i5. load s1,r3 i6. use r3 & def r3 i7. store r3,s1 /* dead code after removing i8 */ .....no def of r3 i8. load s1, r3 /* last use of s1*/ i9. use r3

Fig. 3. Dead code store (instruction $i7$) due to load elimination (instruction $i8$).

be the last instruction using the spill register $s1$. Spill registers are analyzed as usual virtual registers and $s1$ is going to be seen as *dead* after program point $i7$. This means that its definition in $i7$ is dead code as well and can be removed.

Next, we present the remaining routines from the first stage—*BlockFlowFunction* (which is the 'core' of the analysis algorithm) and *FirstPhase*, which is the entry point in this phase:

```
void FirstPhase(FlowGraph fg){
  IterativeAnalysis(FG);
  foreach basic_block B in FG do
    BlockFlowFunction(B,true);
  DeadCodeElimination(FG);
}
```

```
EquivalenceInfo BlockFlowFunction(Block b, bool elim)
pp_equiv_info = b.equiv_info
foreach ES in b do
  foreach instruction I in ES (traversed so that data dependencies
  are fulfilled) do
  {
    if (I is a call instruction) then
      foreach register r destroyed by callee do
      delete_equiv(pp_equiv_info, r);
    if(I does not change any register) then continue;
    if(I reloads register r from memory ) then
      delete_equiv(pp_equiv_info, r);
    if(elim & & redundant(pp_equiv_info, I)) then
      remove_inst(I)
    if(! redundant(pp_equiv_info,I)) then
    {
      delete_equiv(pp_equiv_info, dst(I));
      add_equiv (pp_equiv_info,I, dst(I));
    }
  }
  return pp_equiv_info;
}
```

The heart of the analysis algorithm is the function *BlockFlowFunction*, which produces the EquivalenceInfo at the end of a basic block, given the EquivalenceInfo at the beginning of the basic block. In this paper, we have presented a simplified version of it, that does not deal with issues such as predicated execution or unpairable execution sets

(i.e., execution sets with circular data dependences—for example the 'swap' execution set: tfr $d0$, $d1$ and tfr $d1$, $d0$). The analysis over one basic block is done as follows: for each instruction that changes at least one register[4] we first check to see whether the instruction is redundant (i.e., writes a value that was already existing in the register) and if it is, then we can safely remove it (if the value of *elim* parameter is 'true'). If it is not redundant, we delete all equivalences of that register, and recreate a new set of equivalences. We do this by using three functions: *delete_equiv*, *add_equiv* and *redundant*.

The function *redundant* checks to see whether the computation of an instruction was not already performed. It can be easily implemented using the *add_equiv* function—it simply checks whether this instruction writes any register; if it does, then it is redundant if it would bring no new equivalence information for that register.

The function *delete_equiv* removes the equivalence information related to a register—that is, it initializes the information for that register to *bottom* and removes all other equivalence information related to it (i.e., if we destroy $r1$, we will need to destroy the equivalences $r2 = r1$ and $r3 = r1 + 1$).

The function *add_equiv* is the most complex function used in this stage. It contains an iterative inference engine that first extracts the obvious equivalence information provided by the instruction (i.e., if we have tfr $d1$, $d2$, we know that after this instruction $d1$ will be equal to $d2$) and then updates all equivalence information where the register is involved, considering the existing equivalence information. For example, in the case of a tfr instruction from $d1$ to $d2$, the first step is to mark the fact that $d1$ is equal to $d2$ and $d2$ is equal to $d1$; but after that, we also need to extend the existing equivalence information (i.e., if we knew that $d1$ is constant and has the value '3', we must mark the fact that $d2$ equals 3; further, if we knew that $d3$ is $d1 + d6$, we must add the equalities $d3 = d2 + d6$, $d2 = d3 - d6$, $d6 = d3 - d2$). This function should also take into account the arithmetic properties of an operation (e.g., commutability).

Note that, besides removing redundant instructions, we could also replace spill registers with expressions, thus providing rematerialization. However, doing this from the first phase may lead to wrong rematerialization decisions which could adversely impact on the second stage, so it is a good idea to delay rematerialization until the second phase.

[4] Here by register we also mean spill register; also note that call instructions are considered to destroy all physical registers that are not callee saved (according to ABI), but no spill registers.

3.3. Phase Two—Rematerialization of Spill Registers

In the second phase, spill registers are bound to physical registers, constants/expressions (when the value may be rematerialized, and this has not been done in the previous step), or memory locations, according to the designated preferences and priorities.

```
SecondPhase(FlowGraph FG, EquivalenceInfo EI){
   Webs = ComputeWebs(FG);
   foreach web W in Webs do
      RematerializeAll(W, EI, false); //(1)
   IGraph = ComputeInterferenceGraph(Webs);
   Pri = ComputePriorities(Webs, IGraph);
   foreach web W from Webs, sorted by Pri do
   {
      RematerializeAll(W, EI, true);//(2)
      if(no remaining uses for W) then continue;
      Color = ChooseBestColor(W,IGraph, EI);
      foreach rematerialized use from W do
      if (reload from Color is better than current rematerialization
      instruction) then
         Change the rematerialization instruction to a reload instruction;
      Adjust EI to reflect destroyed equivalences;
   }
   DeadCodeElimination(FG);
}
```

Note that in the coloring stage, we do not color virtual spill registers, but instead we color webs (roughly speaking, a web is a distinct lifetime of a register—for a more precise definition see Ref. 7). This is because a spill register is likely to have several lifetimes (and this likelihood is further increased by the first spill elimination stage). Consider the example in Fig. 4.

In this example, the spilled variable creates a spill register with two distinct lifetimes, which can, and actually should be colored individually. This case is quite frequent in hierarchical register allocation, where the 'large portion of the program' may be an inner loop where x was allocated to a physical register, and outside the loop x was spilled. In this case, we would have fix-up load and store instructions that would follow exactly the pattern presented in this example.

Let us go back to the coloring algorithm—notice that immediately after building the webs, we make a first call to *RematerializeAll*. This procedure uses the equivalence information to replace all possible reloads

```
def x
store x,s0
.....
load s0,x

[ a large portion of the program that uses x ]

store x,s0
.....
load s0,x
use x
```

Fig. 4. Spill register with two lifetimes.

```
def  r1
def r2
add  r1,r2,r3
store r3, s0
def r3 // destroys the old value from r3
[...region with high R-class register pressure...]
[...region with high D-class register pressure...]
(*)load  s0, r3
```

Fig. 5. Rematerialization would stretch lifetimes of $r1$ and $r2$.

with equivalent rematerialization instructions (constants, transfers from an equivalent register, recomputation). However, we do not want all the values to be rematerialized from the very beginning—otherwise we would have done this from the first stage. Instead, at the first call to *RematerializeAll* (denoted with (1) in the pseudocode) we only rematerialize the values that do not stretch any lifetime, which means that additional conflicts between physical and spill registers are avoided. Consider the example in Fig. 5. In this example, we have a region with high R register pressure, which makes us spill $r3$ and reuse it for different purposes in that code region. Assuming that we have another region with high D register pressure, we may not want to rematerialize $r3$ in the point (*), because doing so would stretch the lifetimes of $r1$ and $r2$ and thus would prevent these two registers from being used as spill locations for the D-class registers in the region where we have a high D-class register pressure. It is better to delay rematerialization decisions that stretch lifetimes until the point (2) in the algorithm, where we have to assign a color for the spill register (and at that point we know that this spill register is the most important one which was not already colored, thus rematerialization is acceptable even if it stretches lifetimes).

After the first rematerialization step, we build the interference graph and we compute the priority of each spill register (actually, for each web). The register with a higher priority will be the first to be colored, as it has a higher impact on the performance of the resulting code. The priority is computed with a heuristic merit function similar to a function that computes spill cost during register allocation. Different merit functions may be used depending on the compilation options: (e.g., compilation for improved cycle-count or code size).

Having the interference graph built and the priority computed, one can start the actual coloring. First, we attempt to rematerialize as many register-uses as possible from the current web. If there is no register-use left to be colored, the job is done and we may continue with the next web. Otherwise, re-evaluate the conflicts (if some uses were rematerialized, there may be a smaller web with less conflicts) and select the best color available. A good heuristic for the 'best' color is that color which could be used by the fewest neighbors (we are only interested in the neighbors that are spill registers) and which destroys as few rematerialization opportunities as possible. For example, if we have a physical register Rx that is in conflict with all the neighbors and it cannot be used to rematerialize any neighbor, then Rx is likely to be the perfect choice for a spill location. Note that we may also have special preferences for some registers (e.g., if we somehow manage to get a register from the same class, we will have smaller register-to-register transfer costs).

After having chosen a register for the spill location (or a stack slot, if no register was available) we make an additional check to see whether it would not be profitable to replace some of the rematerialized instructions with reloads without losing performance and without creating additional conflicts. Replacing rematerialization instructions with a register that was chosen as spill location is useful because it is usually cheaper—not only for speed, but also for code size. For example—we may have no speed penalty when materializing register from a constant, but it is likely that we will have a code size penalty. In addition, reloading from the register that was chosen as spill location could avoid stretching the lifetime of several other physical registers.

Assuming that a register was chosen as spill location, we must take care of the possible inconsistency that may occur in the equivalence information provided by the first step. For example, in the case presented in Fig. 5, if we chose $r1$ as spill location for a D register in the region of high D-class register pressure, we must know that we will not be able to rematerialize $r3$ anymore in point (*).

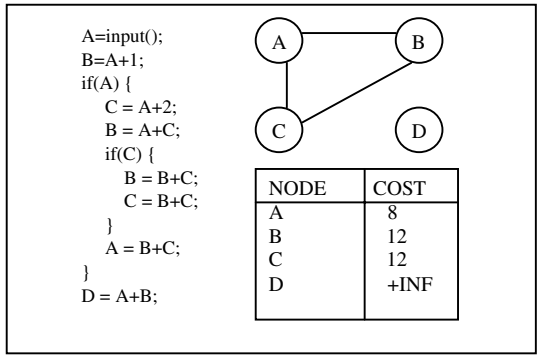

Fig. 6. Code example, with corresponding interference graph and spill costs.

Finally, in the end of the second stage of the algorithm we have to call dead code elimination again, since generally this coloring stage leaves a lot of dead code behind.

3.4. Example

We will now examine a somehow more elaborate example that proves the power of this spill elimination algorithm (this is in fact exactly the example presented by Bergner[4]).

Assume that we have the code in Fig. 6 (with the corresponding spill costs) and that there are only two available registers left. It is obvious that this graph cannot be colored, and we will have to spill the variable A.

A regular register allocator, using Chaitin's local heuristics for spill reduction, would generate the code presented in Fig. 7(a), which has two store instructions and two load instructions. As described in Ref. 4, the interference region spilling heuristic would produce the code shown in Fig. 7(b) that has only one load instruction and one store instruction. Our method would give the optimal result that is shown in Fig. 7(c) (which would actually be the code generated by using interference region splitting; but remember that our method is not a live range splitting algorithm, it is a spill reduction algorithm and thus can complement any register allocation algorithm already implemented).

Let us explain in short why our method produces the result in Fig. 7(c): after doing the first spill reduction phase, we will end up with a code very similar to the one produced by IR spilling (the last load would be eliminated because the value of A already exists in $r0$; the last store will be eliminated as it is dead code).

```
r0=input();
Store r0;
r1=r0+1;
if (r0){
  Load r1;
  r0=r1+2;
  r1=r1+r0;
  if (r0){
    r1=r1+r0;
    r0=r1+r0;
  }
  r0=r1+r0;
  Store r0;
}
Load r0;
r0=r0+r1;
```

```
r0=input();
Store r0;
r1=r0+1;
if (r0){
  Load r1;
  r0=r1+2;
  r1=r1+r0;
  if (r0){
    r1=r1+r0;
    r0=r1+r0;
  }
  r0=r1+r0;

}

r0=r0+r1;
```

```
r0=input();


r1=r0+1;
if (r0){
  r1=r0;
  r0=r1+2;
  r1=r1+r0;
  if (r0){
    r1=r1+r0;
    r0=r1+r0;
  }
  r0=r1+r0;

}

r0=r0+r1;
```

Fig. 7. Results of spill reduction using: (a) Chaitin's spilling heuristic; (b) Bergner's IR spilling heuristic; (c) Our approach.

But as opposed to other spill reduction approaches, instead of spilling directly to the stack, we are now trying to 'color' the virtual spill register (eventually, if we find no color or no way to rematerialize its value, an available stack slot will be assigned to this spill register). Looking at the code, we see that at the point where we have the first load instruction, we have the value already available in $r0$, so we can 'reload' the value directly from $r0$. Thus, the first store will become redundant and be eliminated as dead code.

Note that in this example that we are actually facing a special case of the problem of rematerialization: we could have decided from the very first step that it is possible to remove the first load instruction and replace it with a register-to-register transfer. In this particular case, the results would be the same, but in general, taking this decision in the first step would stretch the lifetime of $r0$ before the 'coloring' step. In a different example, we may have preferred to reload A from stack, and to use the $r0$ register as a spill location for another (more important) variable.

3.5. Support for the Proposed Algorithm in a Production Compiler

In this section, we will talk about the improvements that can be made (generally in other optimization steps) to achieve better results with this spill elimination algorithm.

First—it is quite obvious that it would be best if the spill elimination algorithm had nothing to optimize—which means that in order to

achieve good register allocation, it is very important to have a lifetime-sensitive software pipelining algorithm (examples of such algorithms are Slack Modulo Scheduling[8] and Swing Modulo Scheduling[9]—but depending on the architecture, other algorithms may prove to be better). In addition, if the register allocator follows an instruction-scheduling step—it is best to have a bi-directional, lifetime-sensitive instruction scheduler as the first instruction-scheduling step.

Another very important change that should be made in order to achieve good register allocation is restriction graph splitting, which is actually a kind of 'architecture-dependent live range splitting'. By this, we mean that if the architecture has encoding constraints that force several registers to be colored in a single step, the register allocator should build a restriction graph and should check from that graph whether all registers are colorable. If a register is determined to be uncolorable from the very beginning, we should split its live range to avoid spilling it. For example, assume the following code sequence[5]:

$$
\begin{array}{l}
[\\
max\,vd0, vd1 \\
max\,vd0, vd2 \\
]
\end{array}
$$

In this case, even assuming we have plenty of physical registers available, we cannot assign physical registers to $vd0$, $vd1$, and $vd2$ (if we assign $d0$ to $vd0$ then both $vd1$ and $vd2$ must be $d4$ – but $vd1$ and $vd2$ cannot share the same register—first because we would have two writes to a register in the same cycle, and even assuming we would ungroup the instructions—if neither of them is dead code, then $vd1$ and $vd2$ have a conflict in the conflict graph). Normally, we would have to spill one of the registers—but a better solution is to split the live range of one register as follows, in order to obtain two colorable restriction graphs instead of an uncolorable one:

$$
\begin{array}{l}
tfr\,vd0, temp \\
[\\
max\,vd0, vd1 \\
max\,temp, vd2 \\
]
\end{array}
$$

[5] This example is SC100 assembly code. On StarCore, there is a restriction which forces a distance of +4 between the indexes of the two registers that serve as operands to a *max* instruction; also both operands must be from the same bank—i.e., it is disallowed to have one operand in the interval $d0$–$d7$ and the other one in $d8$–$d15$.

$$\left.\begin{array}{c}\text{tfr } R_x, R_y \\ \text{use } R_y\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\text{tfr } R_x, R_y \\ \text{use } R_x\end{array}\right.$$

Fig. 8. Transfer-use peephole.

$$\left.\begin{array}{c}\text{tfr } R_x, R_y \\ \left[\text{no\_use/redefinition of } R_y\right]\end{array}\right\} \Rightarrow \text{remove tfr } R_x, R_y$$

$$\text{tfr } R_x, R_x \Rightarrow \text{remove tfr } R_x, R_x$$

Fig. 9. Two simple peephole optimizations for transfer instruction removal.

Note that efficiently splitting the conflict graphs in a way that enables the registers to be colored, and by adding a minimal amount of transfer instructions, is a problem by itself and not at all a trivial one. However, we do not discuss this problem here.

If the register allocator has a mechanism for preferences, it is useful to add preferences in such a way that all temporary registers that result after spilling a virtual register will prefer the same color—thus enabling the spill elimination algorithm to remove the useless load/store instructions without leaving register-to-register transfers behind. Note that even if the register allocation algorithm has no mechanism for preferences, we can still avoid useless transfers (but a little bit less efficient) using the following peephole optimizations:

- the peephole optimization in Fig. 8 increases parallelism and provides more opportunities for the following peephole optimizations
- the two peepholes in Fig. 9 are designed to remove the useless transfer instructions
- finally, we may add the (somehow more sophisticated) peephole in Fig. 10 that removes transfer instructions, at the same time increasing parallelism.

Generally, the stack consumption is another important performance parameter that is monitored especially in DSP applications. The presented algorithm can be easily enriched with analysis and heuristic functions in

$$\left.\begin{array}{c}[\text{definition of } R_x] \\ \text{tfr } R_x, R_y \\ [\text{definition of } R_x]\end{array}\right\} \Rightarrow \left\{\begin{array}{l}[\text{definition of } R_y] \\ \text{remove tfr } R_x, R_y \\ [\text{definition of } R_x]\end{array}\right.$$

Fig. 10. A more complex peephole optimization for transfer instruction removal.

order to reuse the stack locations assigned to spill registers in the second phase. The analysis can be done using the spill registers conflict graph: one spill register is replaced by a new stack location or by the stack location assigned to another spill register that is not in conflict with it. The main purpose of this optimization is to decrease stack consumption, but probably on most architectures (including StarCore) it can also be used to reduce code size because the encoding of stack slots with small offsets is usually more efficient than the encoding of stack slots with large offsets. Thus, if we choose to allocate small stack slot offsets to a register (stack slot) that if frequently referenced, can gain a significant amount of code size; we can gain even more code size if instead of individual registers we consider groups of registers that can share the same stack slot. However, care must be taken when sharing a stack slot if we do cycle-count optimizations, because sharing a stack slot introduces additional dependencies that may limit the amount of parallelism available to the compiler.

4. RESULTS

In this section, we present a set of results that prove the efficiency of our spill reduction algorithm. These tests consist mainly of optimized and un-optimized DSP code (which is our primary interest). However, good results have been reported for control code, too.

We tested this optimization in the context of the Metrowerks C Compiler for StarCore, on a set of large applications as well as on several DSP benchmarks. The applications are:

- G729—MDCR[6] optimized C implementation for the ITU-T G.729 8 kbps speech vocoder
- EFR—Out-of-the-box implementation of the GSM Enhanced Full Rate 12.2 kbps speech vocoder
- AMR—Out-of-the-box implementation of the 4750 ... 12200 bits/s speech codec for Adaptive Multi-Rate speech traffic channels
- G723—Out-of the box implementation of the ITU-T G.723 dual-rate speech codec
- G728—optimized implementation of G728 16 kbit/s speech coding, specified in ITU-T recommendation G.728
- JPEG—image compression-decompression
- MP3—mp3 audio format decoder

We compare performance results with and without spill reduction. One should note that the compiler's default spill reduction strategy is

[6] MDCR stands for Motorola DSP Center Romania.

Table I. Cycle-Count Improvement

Application name	Compiler options	Without spill reduction	With spill reduction	Improvement(%)
G729 coder	-O3	96,959	89,219	8.67
	-O3 -Og	92,879	85,747	8.31
G729 decoder	-O3	16,204	15,643	3.58
	-O3 -Og	14,589	14,139	3.18
G723 codec	-O3	755,715	744,407	1.50
	-O3 -Og	734,775	732,896	0.26
EFR codec	-O3	268,497	25,2668	5.90
	-O3 -Og	203,786	19,6014	3.81
AMR coder	-O3	334,468	316,249	5.45
	-O3 -Og	257,597	252,051	2.15
G728 decoder	-O3	363,707	357,481	1.71
	-O3 -Og	330,470	323,661	2.06
G728 encoder	-O3	3,274,962	3197,099	2.38
	-O3 -Og	2,924,223	2,875,217	1.68
JPEG decoder	-O3	2,892,110	2,782,148	3.80
	-O3 -Og	2,885,510	2,777,489	3.74
JPEG encoder	-O3	2,476,065	2,278,939	7.96
	-O3 -Og	2,487,888	2,302,342	7.46
MP3 decoder	-O3	64,564,031	55,936,948	13.36
	-O3 -Og	64,747,121	55,979,876	13.54
Cor_h	-O3	3234	3234	0
Mb01	-O3	16,491	16,486	0
Mb02	-O3	2639	2639	0
Mb03	-O3	22,421	21,743	3.1
Norm_corr	-O3	5704	5704	0
Search_10i40	-O3	10,165	9643	5.41
Vq_subvec	-O3	781	578	26

activated by default, so 'without spill reduction' is actually not a spill-everywhere strategy. The register allocation performs by default live range splitting at tile boundary, and it also performs useless load/store removal (similar in spirit with Chaitin's heuristics described in Ref. 2).

Table I shows cycle-count improvement on some DSP applications and unit-tests.

Notes:

- The results are in cycles on the Motorola StarCore®140 DSP core.
- Compilation options notations: -O3 for cycle-count optimization, -Og for global optimization.
- The results for cycle-count performance do not include memory contention cycles.

Table II. Code Size Improvement

Application name	Compiler options	Without spill reduction	With spill reduction	Improvement (%)
G729 coder	-O3s	53,888	50,880	5.7
	-O3s -Og	53,376	50,432	5.5
G729 decoder	-O3s	45,376	40,144	11.5
	-O3s -Og	40,624	39,888	1.8
G723 codec	-O3s	29,440	28,160	4.35
	-O3s -Og	27,332	25,504	6.35
EFR codec	-O3s	24,256	22,800	6.0
	-O3s -Og	22,592	21,312	5.67
AMR coder	-O3s	62,112	58,784	5.36
	-O3s -Og	59,856	56,416	5.75
G728 decoder	-O3s	28,032	27,488	1.94
	-O3s -Og	27,968	27,312	2.35
G728 encoder	-O3s	27,488	26,912	2.10
	-O3s -Og	27,760	26,784	3.52
JPEG decoder	-O3s	66,688	65,008	2.52
	-O3s -Og	66,384	64,784	2.41
JPEG encoder	-O3s	57,440	56,560	1.53
	-O3s -Og	57,136	56,224	1.60
MP3 decoder	-O3s	82,896	72,896	12.06
	-O3s -Og	82,640	72,576	12.18
Cor_h	-O3s	5008	5008	0
Mb01	-O3s	5728	5648	1.4
Mb02	-O3s	4742	4752	0
Mb03	-O3s	5888	5840	0.8
Norm_corr	-O3s	5312	5248	1.2
Search_10i40	-O3s	8000	7200	10
Vq_subvec	-O3s	4864	4848	0.3

- For some applications such as JPEG and MP3, a significant amount of the presented cycle-count number is due to intensive file I/O necessary for testing the applications.
- The performance increase is more significant on the encoders compared with the decoders. This shows that the encoders are more complex than the decoders, and have significantly higher register pressure.
- Where we have 0% improvement, it is due to good register allocation (no spill)—no improvements could be done to it.

Table II shows code size improvement on some DSP applications and unit-tests.

Notes:

- The results are in bytes and include only the code section measurements.
- Compilation options notations: -O3s for code-size optimization, -Og for global optimization.
- The algorithm efficiency can be noticed especially on out-of-the box code. Also, on G729 decoder one may notice that the spill reduction seems less effective in global optimization, but in fact in this case there is a much lower register pressure and hence less spill code to be optimized-away.

5. CONCLUSIONS AND FUTURE WORK

While graph coloring is widely recognized as 'the method' for solving register allocation problems, the problem of inserting spill code has not yet seen a solution with such a wide acceptance. Several heuristics have been developed in order to tackle this problem, and there are generally two directions in attempting spill code reduction:

- live range splitting register allocators
- spill reduction heuristics.

Aggressive register allocators use both these approaches to try to minimize the spill code—and we have also taken this approach. We used a hierarchical register allocation algorithm that splits the live ranges and inserts spill code in cheaper points (considering the program's control structure) combined with a Chaitin-style spill reduction heuristic. However, this algorithm produced unsatisfactory results when dealing with very large tiles that had high register pressure. We had the choice of adding an intra-tile live range splitting algorithm or designing a better spill reduction algorithm. In our first attempt, we tried to implement a liverange splitting algorithm similar to the one described by Chow (see Ref. 10) in the existing register allocation algorithm, which proved to be a tough task since the two algorithms use radically different approaches. We ended up implementing a live range splitting algorithm that was executed before the actual coloring of the tile—actually, it was only a code transformation designed to reduce register pressure. Two different live range splitting heuristics were implemented:

- the first heuristic starts by choosing 'split points' as being points where we have a relevant increase or decrease in register pressure, then at each split point it splits those variables that have high/low spill costs and a high usage pattern both before and after the split point

- the second approach starts by selecting the live ranges to split (those with a low/high spill cost) and then decides how many times and where to split the live range.

The results yielded by these algorithms were disappointing. Although we had good (sometimes excellent) results on some test cases, on other test cases we had significant performance decrease (going as high as 15–20%).

Note that our live range splitting heuristics do not fit well in a Briggs-like register allocator (since they start from the ideas presented by Chow and Hennessy,[10] which have a different coloring framework). Bergner's interference region splitting algorithm presented in Ref. 4 is a more natural complementary live range splitting solution for Briggs-like register allocators as well as for our hierarchical register allocation algorithm. Still, we did not choose to implement it as the results presented in his PhD thesis showed performance decrease similar to that observed for our live range splitting approach.

The results of our efforts toward implementing a complementary live range splitting algorithm (as well as the results presented by Bergner) led us to the conclusion that (except for the particular cases of architecture-specific restriction graph splitting algorithms described in Section 3.5) it is generally sufficient to do live range splitting at tile boundary. In order to further improve the results of hierarchical register allocation, we need to have a good spill reduction heuristic—and the results obtained using the solution presented here make us believe that this algorithm is a fairly good solution for spill reduction. The great advantage of our approach is that (as opposed to Bergner's IR spilling/splitting) this approach always yields better (or equal) performance when compared with that provided by the supporting register allocator alone (in this paper, a spill reduction heuristic similar in spirit with the one presented by Chaitin).

As future purposes, we plan to fully implement the spill reduction algorithm in our hierarchical register allocator and to evaluate the benefits of the full implementation. In addition, we want to experiment with new heuristics for rematerialization and coloring. Besides, we intend to enrich the stack slot reuse capability with good heuristics for code size reduction. It should also prove interesting to know what is the performance of the spill reduction algorithm when used with a non-hierarchical register allocator, and what other improvements it needs for becoming truly efficient when combined with other register allocation algorithms.

REFERENCES

1. SC140 DSP Core Reference Manual, http://e-www.motorola.com/files/dsp/doc/ref_manual/MNSC140CORE.pdf.

2. G. J. Chaitin, Register Allocation and Spilling via Graph Coloring. SIGPLAN Notices, *Proc. ACM SIGPLAN 1982 Symposium on Compiler Construction*, **17**(6), pp. 98–105 (June 1982).
3. D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter, Spill Code Minimization Techniques for Optimizing Compilers. SIGPLAN Notices, *Proc. ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, **24**(7), pp. 258–263 (July 1989).
4. P. E. Bergner, Spill Code Minimization Techniques for Graph Coloring Register Allocators. PhD thesis, University of Minnesota (1997).
5. D. Callahan, B. Koblenz, Register Allocation via Hierarchical Graph Coloring. *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada (June 26–28, 1991).
6. C. Click and K. D. Cooper, Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, **17**(2):181–196 (1995).
7. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kauufmann Publishers (1997).
8. R. A. Huff, Lifetime-Sensitive Modulo Scheduling, *SIGPLAN Conference on Programming Language Design and Implementation* (1993).
9. J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero, Swing Modulo Scheduling: A Lifetime Sensitive Approach, *International Conference on Parallel Architectures and Compilation Techniques* (October 1996).
10. F. W. Chow and J. L. Hennesy, The Priority-Based Coloring Approach to Register Allocation, *ACM Trans. Program. Languages Syst.*, **12**(4):501–536 (October 1990).