Optimal Bitwise Register Allocation using Integer Linear Programming

Rajkishore Barik[1], Christian Grothoff[2], Rahul Gupta[1], Vinayaka Pandit[1], and Raghavendra Udupa[1]

[1] IBM India Research Lab, Delhi
[2] University of Denver, Colorado

Abstract. This paper addresses the problem of optimal global register allocation. The register allocation problem is expressed as an integer linear programming problem and solved optimally. The model is more flexible than previous graph-coloring based methods and thus allows for register allocations with significantly fewer moves and spills. The formulation can also model complex architectural features, such as bit-wise access to registers. With bit-wise access to registers, multiple subword temporaries can be stored in a single register and accessed efficiently, resulting in a register allocation problem that cannot be addressed effectively with simple graph coloring. The paper describes techniques that can help reduce the problem size of the ILP formulation, making the algorithm feasible in practice. Preliminary empirical results from an implementation prototype are reported.

1 Introduction

This paper presents a family of new register allocation algorithms that are suitable for off-line computation of high-quality register allocations. The algorithm is targeted for embedded systems which need to run medium-sized applications with limited resources. In particular, we assume that the number of registers available is relatively small and that hardware optimizations such as out-of-order execution, caching and prefetching are not available to an extent that would nullify the cost of spilling. In this context, the compile-time cost for a critical method is almost irrelevant – even if the compilation takes extremely long, the resulting gain in either reduced hardware cost or possibly more important reduced energy consumption due to improved code performance is all that matters.

The traditional approach towards decent-quality register allocation is to color the interference graph [6]. If the number of colors exceeds the number of registers, temporaries are selected for spilling until the graph becomes colorable. The approach presented in this paper makes a radical departure from the graph coloring model, completely eliminating the boolean decision of spilling or not spilling a temporary. The basic idea is to allow temporaries to switch registers at any time and to use constraints to force temporaries that are used at a given instruction into appropriate registers only at the time of use. Moving a variable between registers or between registers and the stack is associated with a cost in the goal function of the integer linear program (ILP). The search space for the ILP solver is reduced using observations about the points in time at which it may make sense for a temporary to be spilled in an optimal allocation.

In order to show the expressive power of the approach, this work describes a way to perform optimal register allocation for architectures that allow bit-wise access to registers as proposed in [11]. Various applications, in particular from the embedded domain, make extensive use of sub-word sized values. Examples include network stack implementations and signal- and image-processing as well as hardware drivers. For these applications, bit-wise register allocation can reduce register pressure. Under the assumption that the cost of adding instructions that support sub-word access is smaller than the cost of adding additional registers (or increasing the processor speed to offset the spill-cost), bit-wise register allocation can help reduce total system cost.

The proposed approach uses Integer Linear Programming (ILP) [13]. Using minor variations to the integer linear program the model is able to encompass features from a large body of previous work on register allocation, including bit-wise allocation, coalescing, spilling, use of registers for both spilling and ordinary temporaries, and a limited form of rematerialization [3].

The remainder of the paper is organized as follows. After an overview of related work in Section 2 the ILP formulation for the bit-wise register allocation is introduced in Section 3. Section 4 presents the implementation, performance data for various benchmarks and concludes the paper with a discussion of the benefits and limitations of this approach.

2 Related Work

Previous work has focused on register allocation at local, global, intraprocedural, and interprocedural levels. Intraprocedural register allocation is known to be a NP-hard problem and several heuristics have been proposed to solve the problem [4,5,6].

ILP has been used previously for register allocation and instruction scheduling. The optimal register allocator presented in [9] uses 0-1 linear programming to compute an optimal register allocation. Like this work, the ILP formulation there allows for a variable to moved between registers and memory. The primary difference is that in [9] models the register allocation problem as a sequence of binary decision problems: to allocate or not to allocate a register for a given variable at a given time. The objective function then accumulates the associated costs for each of the possible decisions. The decision model cannot cope with bitwise allocations, mostly because for a bitwise allocator the decision which temporaries should share a register is no longer binary. In [7] the ILP from [9] is transformed in order to improve the solution time while retaining optimality. This is achieved by reduction techniques that exploit the control-flow graph structure and live-ranges of the variables. The presented techniques implicitly assume that every register can only hold one variable and are thus not directly applicable to the model presented in this paper. The transformations resulted in a speed-up of a factor of 150 for solving the resulting ILP over the naïve model.

In [2], the authors solve the register allocation problem for CISC machines by dividing it into two subproblems - optimal spilling and register coalescing. The optimal spilling problem is solved optimally using ILP, while a variant of Park and Moon's heuristic [14] is used for solving the register coalescing problem (sub-optimally). The authors exploit the CISC instruction set to optimize the cost of loads and stores. They take into account that number of available registers may vary from program point to program point, but do not consider variable bit-width requirements of temporaries. In [12] an approach to speculative subword register allocation based on optimistic register allocation is proposed. The idea behind this paper is that there are more opportunities for subword allocation at runtime than what can be statically determined to be safe. The authors propose to use profiling data to speculatively pack variables that are most of the time sufficiently small into a register and to have the processor dynamically detect and flag the case where the size constraints are not met. This requires support by the processor which needs to keep an additional bit per register. If the flag is raised for a given register, the processor executes the appropriate stack accesses instead.

3 ILP Formulations

The key idea for the algorithm is to avoid assigning a temporary (both program defined variables and compiler generated temporaries) a fixed location and instead to formulate certain constraints on the location in the form of an integer linear problem (ILP) [13]. The goal function of the ILP captures the cost of the resulting register allocation. The ILP is then solved using off-the-shelf ILP solver technology.

The input to the ILP-solver is a set of constraints that describe the register allocation problem. The basic constraints are that each temporary must be assigned to exactly one register, and that each register must have enough space to hold all temporaries assigned to it. Dead temporaries do not use any space. Throughout the paper, the stack is modeled as a special register σ that has no space constraints. All other registers can only store b bits of information.

3.1 Basic formulation

The basic formulation only considers allocating one register per temporary for the lifetime of that temporary. Moving a temporary between registers or between registers and the stack is not allowed in this model. The reader can choose to ignore the bit-wise nature of the register allocation and view the problem in a simplified manner where dead temporaries have a bit-width of zero and all live temporaries are word-sized. This would also be the natural simplification to the algorithm for architectures without bit-wise access to registers.

The input to the problem is sets of temporaries $i \in V$ and registers $r \in R$, spill costs S_i for each temporary $i \in V$ and the size $w_{i,n}$ for temporary $i \in V$ at all nodes $n \in N$ in the control flow graph. The special register $\sigma \in R$ is used to represent the stack, which is not constrained in size and is used for spilling. All other registers $r \in R - \{\sigma\}$ are limited to b bits (b is typically 32). The result of the computation is an allocation $x_{i,r} \in \{0,1\}$, with $i \in V$ and $r \in R$, that assigns every temporary a unique register (or the stack σ). The problem can then be stated in a way suitable for integer-linear programming (ILP):

$$\min \sum_{i \in V} S_i \cdot x_{i,\sigma} \qquad (1) \qquad\qquad \bigwedge_{i \in V} \sum_{r \in R} x_{i,r} = 1 \qquad (3)$$

$$\text{such that} \quad \bigwedge_{\substack{r \in R - \{\sigma\} \\ n \in N}} \sum_{i \in V} x_{i,r} \cdot w_{i,n} \leq b \qquad (2) \qquad\qquad \bigwedge_{\substack{i \in V \\ r \in R}} x_{i,r} \in \{0,1\} \qquad (4)$$

Equation (1) models the cost of the solution by summing up the cost of loading temporaries $i \in V$ that have been spilled ($x_{i,\sigma} = 1$) from the stack. Equation (2) states that at all times and for all registers except the stack, the total size of all temporaries stored in any register must not exceed the register size b. Equations (3) and (4) state that every temporary must be assigned exactly one location.

3.2 Control-flow graph formulation

Let (N, E) be the control-flow graph with nodes $n \in N$ and directed edges $(a,b) \in E \subseteq N \times N$. Note that the control-flow graph does not use basic blocks; instead, each individual statement corresponds to two nodes in the graph, one for the register allocation before the instruction and another one for the register allocation after the instruction.

The goal is to be more flexible in terms of *when* temporaries are spilled. The goal is to allow allocations where a temporary is spilled part of the time or where it is moved from one register to another (for example, in order to reclaim fragmented space in the registers in bitwise allocation). This problem can be formulated using linear constraints by introducing additional temporaries $c_{i,r,n} \in \{0,1\}$ that capture at which nodes $n \in N$ register $r \in R$ has been allocated to a new temporary $i \in V$. Let $x_{i,r,n} \in \{0,1\}$ be the decision function that places temporary $i \in V$ into register $r \in R$ at node $n \in N$.

Let $S_{i,n}$ be the spill cost of temporary $i \in V$ at node $n \in N$. The value of $S_{i,n}$ is zero if i is neither defined in the instruction before n nor used in the instruction after n (whichever case applies). $S_{i,n}$ gives the cost of loading i from the stack if i is used at the instruction after node n. If i is defined at the instruction before t, then $S_{i,n}$ is the cost of spilling i to the stack. The cost estimate S also includes a factor that estimates the execution count for spill operations at node n.

Let $\mu_{r,n} \in \mathbb{R}$ be the cost of moving a value to or from register $r \in R$ at node $n \in N$. Using $\mu_{r,n}$, the cost of a move at node n between register a and b is assumed to be given by $\mu_{a,n} + \mu_{b,n}$. For example, given only one family of registers, the value for $\mu_{r,n}$ for $r \in R - \{\sigma\}$ is half the cost of a register-to-register move. In this case, if ζ_n is the cost of moving a value between the stack and a register at node n, then $\mu_{\sigma,n} = \zeta_n - \mu_{r,n}$. Unlike S, the value of μ is independent of the access pattern of the program. For $\mu_{r,n}$ the node $n \in N$ only plays a role in that it can again be used to factor in the fact that a move within a loop is more expensive than a move outside of loops.

Let $w_{i,n}$ be the number of bits that temporary $i \in V$ requires at node $n \in N$. The resulting integer linear program is shown in Figure 1.

The new goal function (5) adds the cost for moving a temporary i from or to register r at node n in (indicated by $c_{i,r,n} = 1$) at cost $\mu_{r,n}$. The new formulation allows each variable to be spilled part-time while residing in registers at other times. Thus the spill cost is now differenciated into a per-node, per-variable access cost $S_{i,n}$ which is incurred if $x_{i,\sigma,n} = 1$, that is i is spilled onto the stack σ at node n.

$$\min \sum_{\substack{i \in V\\ n \in N}} S_{i,n} \cdot x_{i,\sigma,n} + \sum_{\substack{i \in V\\ r \in R, n \in N}} \mu_{r,n} \cdot c_{i,r,n} \tag{5}$$

$$\text{such that} \quad \bigwedge_{\substack{r \in R - \{\sigma\}\\ n \in N}} \sum_{i \in V} x_{i,r,n} \cdot w_{i,n} \leq b \tag{6}$$

$$\bigwedge_{\substack{i \in V\\ n \in N}} \sum_{r \in R} x_{i,r,n} \geq 1 \tag{7}$$

$$\bigwedge_{\substack{i \in V, r \in R\\ (p,n) \in E}} (x_{i,r,n} - x_{i,r,p}) + c_{i,r,n} \geq 0 \tag{8}$$

$$\bigwedge_{\substack{i \in V, r \in R\\ (p,n) \in E}} (x_{i,r,p} - x_{i,r,n}) + c_{i,r,n} \geq 0 \tag{9}$$

$$\bigwedge_{\substack{i \in V, r \in R\\ n \in N}} x_{i,r,n} \in \{0,1\} \tag{10}$$

$$\bigwedge_{\substack{i \in V, r \in R\\ n \in N}} c_{i,r,n} \in \{0,1\} \tag{11}$$

Fig. 1. ILP formulation for bitwise register allocation with a graph-based control flow model.

Equation (6) directly corresponds to equation (2); the only change is that now also x depends on $n \in N$. The new constraints (8) and (9) ensure that $c_{i,r,n}$ must be 1 each time that $x_{i,r,n} \neq x_{i,r,p}$ for some predecessor p of $n \in N$. Equation (10) states that for any node n a variable i must either be assigned to register r or not; a partial assignment is not allowed. While implied, equation (11) makes it obvious that $c_{i,r,n}$ is a boolean variable and that in particular values greater than 1 are not possible.

3.3 Zero-cost moves

Dead temporaries can be moved between registers at no cost. The model so far considers all moves to be of equal cost, regardless of the liveness of the temporary. A good way to allow for zero-cost moves is to split all temporaries with multiple disjoint live-times into multiple temporaries, one for each live-time. While this increases the size of the set V, the simplification presented in the next section avoids significant growth of the problem due to this extension.

3.4 Optimizations to the ILP

The ILP stated so far can be optimized in order to achieve faster solution times. The basic idea is to reduce the search space for the ILP solver by adding constraints that fix the value of problem variables without changing the value of the goal function for the optimal solution.

Let $pred(n) := \{p | (p,n) \in E\}$ be the set of immediate predecessors of node $n \in N$. Let $L_n \subseteq V$ be the set of temporaries that are not accessed at nodes n and $pred(n)$ and that are either dead or have maximum size at node $n \in N$. Considering that $S_{i,n}$ specifies the spill-cost for temporary i at node $n \in N$, the exact definition of L_n is

$$L_n := \left\{ i \in V \,\middle|\, w_{i,n} \in \{0, b\} \wedge \bigwedge_{d \in pred(n) \cup \{n\}} S_{i,d} = 0 \right\}.$$

Let $M \subseteq N$ be the set of nodes where the move cost is equivalent compared to all previous and next nodes (and thus performing the move earlier or later does not change the cost of the move). The intuitive meaning of M is the set of nodes where the control-flow is linear. Formally

$$M := \left\{ n \in N \,\middle|\, \bigwedge_{p \in pred(n)} \bigwedge_{\substack{s \in succ(n) \\ r \in R}} \mu_{r,p} = \mu_{r,n} = \mu_{r,s} \right\}. \tag{12}$$

Lemma 1 *The optimality of the solution computed by the ILP solver is preserved if the constraint*

$$\bigwedge_{\substack{r \in R \\ n \in M}} \bigwedge_{\substack{i \in L_n \\ p \in pred(n)}} x_{i,r,n} = x_{i,r,p} \tag{13}$$

is added. The constraint further implies that

$$\bigwedge_{\substack{r \in R \\ t \in M}} \bigwedge_{i \in L_n} c_{i,r,n} = 0. \tag{14}$$

Proof: Suppose for some $r \in R$, $n \in M$, $i \in L_t$ an optimal solution to the ILP exists with $x_{i,r,n} \neq x_{i,r,pred(p)}$. If $x_{i,r,n} = 1$, then i was moved at node n out of register $r \in R$. If $w_{i,n} = 0$, performing the move earlier at time p makes no difference at all (since the temporary is dead and only assigned a register pro-forma). Suppose $w_{i,n} = b$. In that case, i must be moving from the stack into a register or vice versa, since moving i from one register $r \in R - \{\sigma\}$ to another register $r' \in R - \{r, \sigma\}$ must be a useless move in an architecture with orthogonal registers and can thus not occur in an optimal solution.[3] Assume that i is moved from σ to register $r \in R - \{\sigma\}$. Then this move can be deferred until time $succ(n)$ (change to the previous optimal solution requires setting $x_{i,r,n} = 0$ and $x_{i,\sigma,pred(n)} = 1$). This is always possible, since deferring the move only reduces register pressure (σ has no space constraint). Similarly, if the move is from register $r \in R - \{\sigma\}$ to σ, the move can be performed earlier at $pred(n)$ without changing the cost and again strictly reducing register pressure. The situation for $x_{i,r,n} = 0$ is analogous. $\square$

Symmetry Another improvement in performance can sometimes be achieved by eliminating symmetry from the ILP formulation. Symmetry occurs in the ILP since all registers (in the same register family) are equal. Thus for n registers, there exist $n!$ completely equivalent solutions. If symmetry is not addressed, the ILP solver may end up

[3] Note that this would not be true for $w_{i,n} \in (0, b)$, since such a move might defragment the available space in the registers in that case.

enumerating all of these. Symmetry can be reduced by adding constraints that eliminate redundant solutions.

Note that the register order can only be chosen arbitrarily for one specific node since in an optimal solution the selected permutation may limit the register order at other places. Since only live variables will impose constraints on neighboring nodes, the impact of selecting a register order can be increased by adding a constraint at a node where the number of live variables in registers is high.

Lemma 2 *Let $n_S \in N$ be a node where the number of live variables is maximized. Let $W \subseteq V$ be the set of variables at node n_S for which $w_{i,n_S} = b$.*[4] *Let $Q \subseteq R$ be a subset of equivalent registers (that is, $S_{i,n} = S_{j,n}$ and $\mu_{i,n} = \mu_{r,n}$ for all $i, j \in Q$ and $n \in N$). For example, given a suitable architecture $Q := R - \{\sigma\}$. Let $<_Q$ be a total ordering of the registers and $<_W$ be a total ordering of the variables in W.*

Then, adding the constraint

$$\bigwedge_{\substack{r_1,r_2 \in Q, i_1,i_2 \in W \\ r_1 <_Q r_2, i_1 <_W i_2}} x_{i_1,r_1,n_S} + x_{i_2,r_2,n_S} < 2 \tag{15}$$

does not change the value of the optimal solution for the ILP.

Proof: Let $x \in \{0,1\}^{V,R,N}$ be an optimal solution for the ILP without constraint (15). It needs to be shown that there exists a solution $x' \in \{0,1\}^{V,R,N}$ that satisfies (15) with the same cost. Equation (15) can be reformulated into a logical expression:

$$\begin{aligned} & \bigwedge_{\substack{r_1,r_2 \in Q, i_1,i_2 \in W \\ r_1 <_Q r_2, i_1 <_W i_2}} x_{i_1,r_1,n_S} + x_{i_2,r_2,n_S} < 2 \\ \Leftrightarrow & \bigwedge_{\substack{r_1,r_2 \in Q, i_1,i_2 \in W \\ r_1 <_Q r_2, i_1 <_W i_2}} (x_{i_1,r_1,n_S} = 1) \Rightarrow (x_{i_2,r_2,n_S} = 0) \\ \Leftrightarrow & \bigwedge_{\substack{r_1 \in Q \\ i_1 \in W}} \left((x_{i_1,r_1,n_S} = 1) \Rightarrow \bigwedge_{\substack{r_2 \in Q, i_2 \in W \\ r_1 <_Q r_2, i_1 <_W i_2}} x_{i_2,r_2,n_S} = 0 \right) \end{aligned}$$

For variable $i \in W$ let $r_i \in R$ be this register ($x_{i,r_i,n_S} = 1$) for the optimal solution x. Let $\tau^0 : R \to R$ be the identity permutation. For any permutation τ^k define $x^k_{v,r,n} := x_{v,\tau^k(r),n}$. While x^k violates (15) define x^{k+1} iteratively as follows:
Let $r^k \in Q$ be the smallest (with respect to $<_Q$) register for r_1 such that the above equation is not satisfied for $r_1 := -r^k$ and suitable choices for the other terms. Let $i^k \in W$ be the unique variable for which $x_{i^k,r^k,n_S} = 1$. Let $r_2 \in Q$ be the largest register

[4] W could be defined to include all variables that at node n_S cannot be co-located with others in the same register. However, this definition makes the proof simpler and more to the point.

$r^k <_Q r_2$ for which there exists an $j \in W$ with $x^k_{j,r_2,n_S} = 1$ and $i_k <_W j$. Define $\tau^{k+1} := \tau^k \circ \tau_{r^k,r_2}$.
Note that the permutation will eliminate the violation of the constraint since $r_2 >_Q r$ among all registers $r \in Q$ for which there exists an $i \in W$ with $x^k_{i,r,n_S} = 1$ and $i_k <_W i$. Note that precondition $w_{i,n_S} = b$ is important here since it implies that at most one variable from the set W can be assigned to any given register ($\sum_{i \in W} x_{i,r,n_S} \leq 1$ for all $r \in R$). This ensures that there does not exist a $j \neq i$ for which $x^k_{j,r_{i_2},n_S} = 1$ also holds (which would result in (15) making the problem infeasible).
The inductive definition of x^k terminates with a solution $x' = x^{k_{max}}$ (with $k_{max} < |W|$) since in step $k+1$ the equation $j_{k+1} > W\, j_k$ holds and thus enforces progress. The result x' is a feasible solution (using $c'_{v,r,n} = c_{v,\tau(r),n}$ gives the values for c) of equivalent cost (since $\tau(r) = r$ for $r \in R - Q$ implies that cost equivalence follows trivially from register equivalence for $r \in Q$) that satisfies (15). □

3.5 Avoiding to block registers for spilling

Whenever a spilled temporary is used by an instruction, some architectures require that it must be loaded into a register before execution of that instruction. Similarly, if an instruction produces a temporary that is not allocated a register at that point, the result may need to be temporarily stored in a register before it can be spilled onto the stack. A common technique addressing such requirements is to reserve a few registers for accessing spilled temporaries. These registers are then excluded from the ordinary register allocation process. The formulation presented so far assumes that a sufficient number of such registers exist outside of the set of available registers R that is made available to the ILP solver.

Let $a_{i,n} \in \{0,1\}$ be a condition temporary that indicates that at node n temporary i is accessed (used or defined). Since $S_{i,n}$ gives the cost for accessing a spilled variable, this means $a_{i,n} := \mathtt{sgn}(S_{i,n})$. Let $b_{r,n} \in \{0,1\}$ indicate that register r is used for at least one assigned temporary at node n. The term "assigned temporary" is used to differentiate these temporaries from "spilled temporaries" in the classical sense. The following constraints are then sufficient to reserve registers for spilling (extending the ILP model from Section 3.2):

$$\bigwedge_{n \in N} \sum_{i \in V} a_{i,n} \cdot x_{i,\sigma,n} + \sum_{r \in R - \{\sigma\}} b_{r,n} \leq |R - \{\sigma\}| \tag{16}$$

$$\bigwedge_{\substack{i \in V, r \in R \\ n \in N}} b_{r,n} \cdot w_{i,n} \geq x_{i,r,n} \cdot w_{i,n} \tag{17}$$

$$\bigwedge_{\substack{r \in R \\ n \in N}} b_{r,n} \in \{0,1\} \tag{18}$$

In (17) the value of $b_{r,n}$ is forced to be one if a live temporary i exists ($w_{i,n} \neq 0$) that is assigned to register r at node n ($x_{i,r,n} = 1$). As a result, $\sum_{i \in V} a_{i,n} \cdot x_{i,\sigma,n}$ is the number of registers that must be reserved for spilled temporaries at node n and $\sum_{r \in R-\{\sigma\}} b_{r,n}$ is the number of registers assigned to registers at that instruction. Equation (16) describes that the number of allocated registers for both the spilled and register-allocated temporaries must not exceed the total number of registers.

Note that this formulation does *not* take bit-wise access to registers into account. While the ILP solution does allow assignment of temporaries to all registers at times, the formulation does not allow for the possibility of allocating just some bits of registers for the spilled temporaries. The problem with a formulation supporting this kind of allocation is that one would still have to ensure that the spilled temporaries are not scattered over multiple registers. This precludes a simple formulation over the sum of all bits assigned to temporaries.

3.6 Coalescing

Coalescing is an important step in register allocation that assigns the same register to temporaries that are connected by a move in order to reduce the number of move instructions. The problem with coalescing for previous register allocators is that forcing two temporaries to be in the same register can result in additional spilling. In [8] an algorithm is presented that attempts to minimize the number of moves without introducing any additional spills.

Since the ILP formulations presented in this paper (with the exception of the basic formulation in section 3.1) allow spilling temporaries on a per-instruction basis, they do not have the problem of additional spills due to too aggressive coalescing. Where in [5] the merging of move-connected temporaries would force these temporaries to be either spilled or kept in a register throughout their lifetime, the ILP formulation allows for partial spilling. Hence it is possible to merge all temporaries that are connected by moves upfront. The ILP solver will insert the necessary minimal number of moves and spills as required.

Also note that coalescing reduces the total number of variables and thus reduces the problem size of the ILP. From that perspective, coalescing should also be considered an optimization that improves the run-time of the register allocation algorithm.

3.7 Rematerialization

Rematerialization [3] is an important technique based on the realization that is sometimes cheaper to recompute a temporary than to load it from memory or to hold it in a register (and possibly spilling other temporaries). The simplest case for this is where the temporary is a constant value. Extending the presented formulation to the case where values need to be recomputed is not easily possible since this may change the lifetimes of the other temporaries that are used in the expression. On the other hand, rematerialization of constants can be handled easily by the ILP formulation.

Note that it would not be optimal to just replace all uses of a constant with a fresh temporary. Inserting an instruction to load a constant value can be more expensive than keeping the constant in a register if a register is available for free. The cost of loading an immediate value can be modeled precisely in the goal function. This can be achieved by modifying the goal function to allow for temporary-specific spill costs. Let $\mu_{i,r,n}$

be the cost of spilling temporary $i \in V$ at node $n \in N$ to or from register $r \in R$. For temporaries of constant value, the spill-cost to the stack would be zero. The spill-cost for loading a constant-value temporary from the stack would be the cost of a load-immediate instruction. The resulting goal function that incorporates the differentiated spill-cost due to constant rematerialization is then:

$$\min \sum_{\substack{i \in V \\ n \in N}} S_{i,n} \cdot x_{i,\sigma,n} + \sum_{\substack{i \in V \\ r \in R, n \in N}} \mu_{i,r,n} \cdot c_{i,r,n} \tag{19}$$

4 Results

The current implementation uses gcc [1] to generate the ILP problems. gcc was modified to support register allocation for hypothetical architectures with 4 to 8 orthogonal registers. The integer linear problems generated by gcc are then solved using ILOG's AMPL/CPLEX linear optimizer. Code generation based on the ILP results has not yet been implemented. Note that a target platform with bit-wise access does not yet exist to the best of our knowledge. However, platforms allowing subword-register allocation exist and while it is trivial to adjust the ILP model for these architectures the necessary modifications to gcc would be extensive. The various cost-parameters in the ILP formulation were set using gcc's built-in cost estimation functions.

The resulting performance metric is the cost estimate given by the goal function in the ILP formulation (5). While this does not allow us to deduce the overall speed-up that could be obtained from the proposed algorithm, this cost estimate should be a good general metric for comparing register allocators. In order to compare the new register allocation algorithm, the output of various other register allocation algorithms available for gcc was judged using the same metric.

For the evaluation, three previously published register allocators were compared with the various ILP-based allocators presented in this paper. The previously published register allocators are traditional graph coloring [5], linear-scan register allocation [15] and Tallam and Gupta's bitwidth aware global register allocation [17]. In order to allow a fair comparison with the graph coloring and linear scan allocators we give numbers for the ILP models for both bitwise and ordinary (wordwise) register allocation. Also, the bitwidth estimation algorithm used is the same for all allocators supporting bitwise allocation.

4.1 Bitwidth-estimation

The bitwidth information for various temporaries at different program points are determined using the approach suggested by [17]. The bitwidth of a temporary i at program point n is represented in a pair (ld, tr), where ld represents the leading zero bits of i at n (leading dead bits) and tr represents the trailing zero bits of i at n (trailing dead bits). In order to determine (ld, tr) pairs for all temporaries at all program points, first a forward data flow analysis is performed to chain the definition of temporaries to their uses. Then a backward data flow analysis is performed to refine the (ld, tr) pairs by chaining the use of temporaries to their respective definitions. The (ld, tr) pair for all temporaries are computed simultaneously as they are interdependent on each other.

4.2 Benchmarks

The performance of the approach is evaluated using benchmarks from the Bitwise [16] and Mediabench [10] suites. These benchmarks are appropriate since they correspond to real-world applications where sub-word access to temporaries is common. Furthermore, using some of the same benchmarks as [17] enables comparison with prior work on bitwise register allocation.

4.3 Impact of the optimizations

Applying Lemma 1 to the ILP formulation reduces the searchspace for the ILP solver. Depending on the benchmark the resulting constraints can eliminate up to 90% of the free variables. Note that the reduction of problem variables does not only reduce the space requirements but also significantly reduces the search space for the ILP solver. For example, `mpegcorr` with 8 registers takes 422s without the constraints allowed by Lemma 1, but only 46s with those constraints.

Surprisingly, using the constraints from Lemma 2 increases the cost for this benchmark, if applied together with Lemma 2 the solution time is between roughly 50 and 300s depending on the choice of n_S. While the additional constraints from Lemma 2 also reduce the search space, this reduction can somehow not offset the cost of evaluating the additional constraint for the specific ILP solver and any of the benchmarks that have been tested for this during our study.

Another possible optimization is to use any of the other register allocation algorithms to compute a feasible register allocation and to feed this as an input into the ILP solver. Such a feasible starting solution can help the solver in its branch and bound search to prune the search space. In our benchmarks, we use the best solution computed by either graph coloring, linear scan or Tallam's bitwise allocator as the starting solution. The resulting performance improvements vary depending on the benchmark. Typically smaller benchmarks see no improvement or even performance degradation, whereas larger benchmarks see (often minor) performance improvements. For example, the `mpegcorr` benchmark runs in 46s with a starting solution, but takes 418s without it. We speculate that the smaller search space for small benchmarks gives fewer opportunities for pruning and the given initial solution is further away in the search space from the optimal solution than the default infeasible starting point that would be used without a starting solution. Future work may result in some deeper understanding of the circumstances under which the various combinations of these optimizations (namely Lemma 1, Lemma 2 and giving a starting solution) are most effective.

The performance results reported for the benchmarks henceforth use a feasible starting solution in combination with the constraints from Lemma 1.

4.4 Performance

Figure 2 and Figure 5 shows the cost as estimated by the goal function (5) for the various register allocation algorithms for all benchmarks.

The time it took to solve the different ILP problems were obtained by running ILOG's cplex v9.1 on an Intel Xeon 3 Ghz with 4 GB memory running Linux 2.6.10.

Benchmark	Reg.	adpcm	convolve	median	mpegcorr	NewLife	MotionTest	Histogram
Graph Coloring [5]	4	1225415	0	91280	92400	2236190	4690	7515
Linear [15]	4	1450425	0	131217	127913	1752698	7060	106605
Tallam [17]	4	800330	0	91280	92400	2136180	4690	5160
ILP GCF	4	490124	0	44710	73850	599642	1919	3773
ILP GCFB	4	330071	0	44710	73850	599642	1916	2837
Graph Coloring [5]	6	750315	0	34575	34835	531305	260	1990
Linear [15]	6	1025311	0	82283	67444	743840	4560	4310
Tallam [17]	6	325230	0	34575	34835	531305	260	1195
ILP GCF	6	270084	0	17795	28550	251428	105	794
ILP GCFB	6	120045	0	17795	28550	251428	105	6
Graph Coloring [5]	8	275215	0	17870	8055	27915	0	0
Linear [15]	8	575214	0	72248	38415	218790	0	0
Tallam [17]	8	130	0	17870	8055	27915	0	0
ILP GCF	8	120054	0	6452	1062	11404	0	0
ILP GCFB	8	42	0	6452	1062	11404	0	0

Fig. 2. ILP GCF is the ILP model with the graph-based control flow model and without bitwise allocation. ILP GCFB is the graph-based control flow model with bitwise allocation. Memory load/store cost metric is 5.

Benchmark	adpcm	convolve	median	mpegcorr	NewLife	MotionTest	Histogram
4 registers	53257s	0s	73s	10s	57s	2s	6s
6 registers	$\geq 10^5$s	0s	44s	35s	163s	3s	11s
8 registers	454s	0s	80s	46s	312s	1s	6s

Fig. 3. Solver time for the bitwise graph-based ILP formulation, ILP GCFB. Entries prefixed with $>$ indicate that the ILP was timed out before completing.

Benchmark	adpcm	convolve	median	mpegcorr	NewLife	MotionTest	Histogram
4 registers	$\geq 10^5$s	0s	23s	10s	42s	2s	5s
6 registers	331s	0s	39s	27s	168s	3s	11s
8 registers	4162s	0s	80s	43s	286s	1s	6s

Fig. 4. Solver time for the graph-based ILP formulation, ILP GCF.

While the time may seem excessively long for some benchmarks, note that this is using a stock ILP solver that has not been specialized to the problem at hand. Furthermore, the run-times should still be feasible when compiling small programs or performance critical functions for embedded systems where high performance and low per-unit cost are paramount.

Note that various benchmarks show that adding more registers does not always have to increase the cost of the ILP – typically at some point computing an optimal solution becomes much easier because there are few or no points of register pressure left.

Benchmark	Reg.	bilint	edge-detect	levdurb	g721	adpcm-coder
Graph Coloring [5]	4	1225490	3331565	912080	90565	1419575
Linear [15]	4	687730	2134138	1424956	148124	1736976
Tallam [17]	4	975390	3331565	912080	87705	1028675
ILP GCF	4	260104	1333229	582787	29943	581829
ILP GCFB	4	$\leq$80072	$\leq$1333181	582787	29943	412035
Graph Coloring [5]	6	325130	2604260	427840	32040	955580
Linear [15]	6	487665	1535156	851177	98388	1409078
Tallam [17]	6	287615	2604260	427840	32040	539670
ILP GCF	6	142557	$\leq$962851	260489	13608	401715
ILP GCFB	6	$\leq$92537	$\leq$952981	260489	13608	222531
Graph Coloring [5]	8	150060	2076610	193580	0	639035
Linear [15]	8	350125	1129861	489472	11945	1032645
Tallam [17]	8	150060	2076610	193580	0	234130
ILP GCF	8	72529	$\leq$1346346	41844	0	278352
ILP GCFB	8	$\leq$35014	$\leq$1346346	41844	0	100776

Fig. 5. Solution cost for various larger benchmarks using the graph-based control flow model. Entries prefixed with $\leq$ indicate that the ILP solver was aborted prior to proving optimality. Memory load/store cost metric is fixed to 5.

Benchmark	bilint	edge-detct	levdurb	g721	adpcm-coder
4 registers	$> 10^6$s	$> 10^5$s	24s	57s	5545s
6 registers	$> 10^5$s	$> 10^5$s	102s	321s	845s
8 registers	$> 10^5$s	$> 10^5$s	138s	41s	648s

Fig. 6. Solver time for the graph-based bitwise ILP formulation, ILP GCFB. Entries prefixed with $>$ indicate that the ILP was timed out before completing.

Benchmark	bilint	edge-detct	levdurb	g721	adpcm-coder
4 registers	5s	1342s	26s	443s	3855s
6 registers	16s	$> 10^5$s	60s	311s	935s
8 registers	18s	$> 10^5$s	77s	15s	1588s

Fig. 7. Solver time for the graph-based ILP formulation, ILP GCF

In general, the runtime of the solver is rather unpredictable. For example, the adpcm benchmark with word-wise register allocation runs takes more than 10^5s for 4 registers, runs rather quickly in 331s for 6 registers. Astonishingly, if the number of registers is increased to 8 the solver takes again significantly longer with 4162s.

Figure 8 gives some data comparing the size of the benchmarks and the respective runtime of the ILP. The size of the benchmark is determined by the number of local variables ($|V|$), the number of nodes ($|N|$) and edges ($|E|$) in the control flow graph.

Benchmark	$\|V\|$	$\|E\|$	$\|N\|$	time (GCFB)
adpcm	29	228	218	53257s
median	34	190	184	73s
NewLife	61	312	302	57s
levdurb	37	206	199	24s
mpegcorr	31	185	178	10s

Fig. 8. This Table shows the relationship between benchmark size and the time it takes to solve the ILP for some selected benchmarks. The number of registers is fixed to 4.

4.5 Discussion

At the surface the large and unpredictable ILP solution times seem to be the big problem with the presented approach. However, in practice, the optimal solution is computed rather quickly, especially given a good initial feasible starting solution. The solver spends most of its time proving that this solution is optimal. Naturally such a proof does not yield any speedups later, so it is perfectly reasonable to turn the presented algorithm into a heuristic by simply aborting the ILP solver if the computation takes too long without improving the solution. This will allow the user to select an appropriate trade-off between register allocation quality and compile-time.

In addition to heuristics that abort the ILP solver earlier, solution times can be improved dramatically using straight-forward reductions of the ILP problem size. One possibility is to map multiple nodes from the original control-flow graph to one node in the ILP formulation. Also, on many processors bit-wise register allocation may not be useful or effective anyway. As the 32-bit timings have shown, using a more coarse allocation granularity can dramatically improve solution times. Changing the ILP formulation to other granularities such as nibbles or bytes is trivial. All of these changes can improve ILP solution times at the expense of reduced quality of the produced solutions. Future work will have to evaluate which graph reduction strategies will give the most effective trade-offs.

5 Conclusion

This paper introduced a new ILP-based algorithm for bit-wise register allocation. The presented formulation expands the expressiveness of the model of existing ILP-based register allocation algorithms and hence allows for better solutions. The algorithm integrates previous techniques including coalescing, spilling, constant rematerialization and register families and allows for temporaries to be temporarily spilled. The formulation supports using the same register for access to spilled temporaries or direct temporary assignment at different times. Experimental results show that the resulting ILP problems can be solved by modern of-the-shelf ILP software, resulting in register allocations that substantially improve on allocations computed by state-of-the-art techniques.

6 Acknowledgments

We thank Calin Cascaval, Satish Chandra, Nandivada V. Krishna and Jens Palsberg for their comments on earlier drafts of this paper.

References

1. http://gcc.gnu.org/, 2004.
2. Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 243–253, 2001.
3. Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 311–321, New York, NY, 1992. ACM Press.
4. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
5. Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, Jun.
6. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
7. Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
8. Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
9. David W. Goodwin and Kent D. Wilken. Optimal and Near-Optimal Global Register Allocation Using 0-1 Integer Programming, 1996.
10. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, 1997.
11. Bengu Li and Rajiv Gupta. Bit section instruction set extension of arm for embedded applications. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 69–78. ACM Press, 2002.
12. Bengu Li, Youtao Zhang, and Rajiv Gupta. Speculative subword register allocation in embedded processors. In *Proceedings of the LCPC 2004 Workshop*, 2004.
13. John L. Nazareth. *Computer Solution of Linear Programs*. Oxford University Press, 1987.
14. Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 196–204, Paris, October 1998. IFIP,ACM,IEEE, North-Holland.
15. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
16. Mark Stephenson, Johnathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation.
17. Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. *ACM SIGPLAN Notices*, 38(1):85–96, January 2003.

A ILP Progress over Time

The presented algorithm can easily be converted into a heuristic by aborting the ILP solver before optimality is established. In this context, it is interesting to know how fast the ILP solver finds good solutions. The following graphs show the improvement of the best known integer solution over time for some representative benchmarks (starting with the starting solution and ending with the optimal solution).

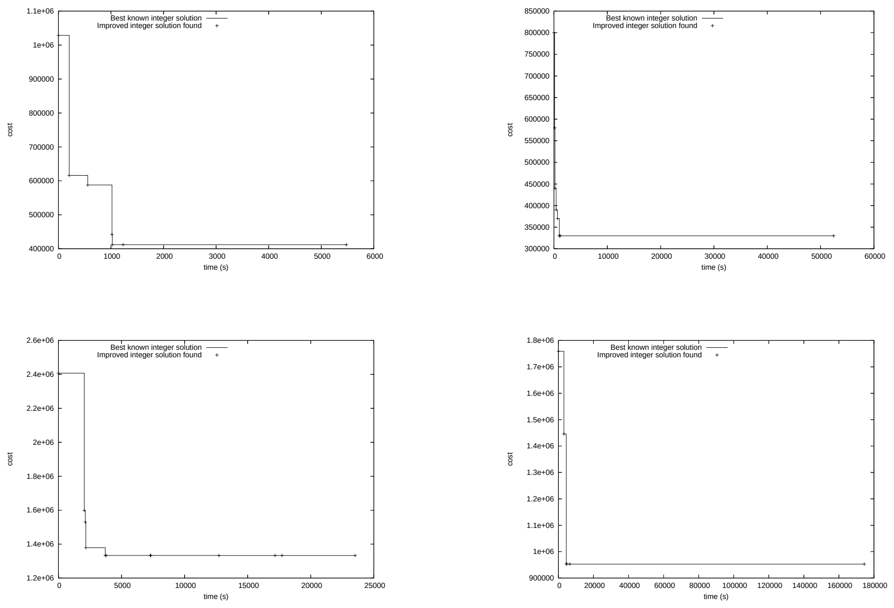

Fig. 9. For many benchmarks, near-optimal solutions are found quickly. Graphs for bitwise solutions for adpcm-coder (4 reg.), adpcm (4 reg.) and edge-detect (4 and 6 reg.).

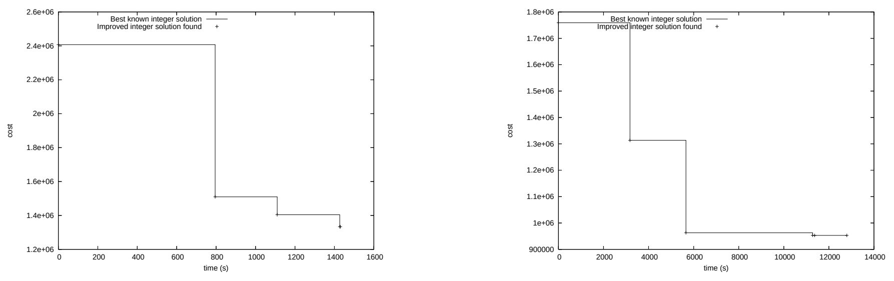

Fig. 10. For some benchmarks, it takes a while for better solutions to be found. Graphs for wordwise solutions for edge-detect (4 and 6 reg.).