Nearly Optimal Register Allocation with PBQP *

Lang Hames and Bernhard Scholz

School of Information Technologies
The University of Sydney, NSW 2006, Australia
{lhames, scholz}@it.usyd.edu.au

Abstract. In this work we present a new heuristic for PBQP which significantly improves the quality of its register allocations and extends the range of viable target architectures. We also introduce a new branch-and-bound technique for PBQP that is able to find optimal register allocations.

We evaluate each of these methods, as well as a state of the art graph colouring method, using SPEC2000 and IA-32 as a testbed. Spill costs are used as a metric for comparison. We provide experimental evidence that our new heuristic allows PBQP to remain effective even for relatively regular architectures such as IA-32, generating results equal to those of a start-of-the-art graph colouring technique. Our method is shown to run 3–4 times slower than graph colouring, however it supports a wide range of irregularities.

Using our branch-and-bound solver for PBQP we were able to solve 97.4% of the functions in SPEC2000 optimally. These results are used as a yardstick to show that both PBQP and graph colouring produce results which are very close to optimal.

1 Introduction

Efficient utilisation of machine resources demands highly optimising compilers as we reach the limits of Moore's law [1]. *Register allocation* is a key optimisation which decides how programs will use the *CPU registers* which form the top level of the memory hierarchy. As increases in CPU speed continue to outstrip reductions in memory latency, the efficient use of registers becomes ever more important for ensuring program performance.

In the intermediate representation of a compiler it is assumed that there are an arbitrary number of symbolic registers available. During the register allocation stage the compiler attempts to map these symbolic registers to real registers. Symbolic registers for which no CPU register can be found (because all are already in use) are forced to reside in memory. Such symbolic registers are said to have been *spilled* to memory. Load and store code must be inserted into the program to retrieve spilled values before they are used, and store them after they are defined. This inserted code, called *spill code*, reduces program performance and is referred to as the *spill cost* of the symbolic register. The challenge of register allocation is to find an assignment which minimises the total spill cost, while terminating within a reasonable time frame. The scope of the register

* This work has been supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190.

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 346–361, 2006.

allocation is either on the basic block level (also known as *local register allocation*) or over a whole procedure (also known as *global register allocation*).

The classical formulation of the global register allocation problem is NP complete. It has traditionally been solved heuristically using the *graph colouring* method introduced in [2], and extended in [3]. In this method a global register allocation problem is described by an *interference graph* in which nodes represent symbolic registers, and edges represent interference constraints. Using heuristics, the register allocator attempts to compute a k-colouring of the interference graph, where each colour represents a CPU register. If a k-colouring of the graph can be found it is mapped to a register allocation. If no k-colouring can be found then some non-colourable nodes are spilled. Spill code for these symbolic registers is inserted, the interference graph reconstructed and the colouring process is restarted.

Graph colouring methods have been shown to be highly effective at producing allocations for regular register architectures [3,2]. They have also been extended to support architectures with irregularities such as register pairing [4], and register classes and aliasing [5]. However, graph colouring methods generally lack the descriptive power required to accurately model the costs and constraints of more irregular architectures. Several alternative methods of register allocation have been devised to support irregular architectures, including *Integer Linear Programming* (ILP) [6], *Multi-Commodity Flow Network* (MCFN) methods [7], and *Partitioned Boolean Quadratic Programming* (PBQP) [8].

In this work we focus on the underlying mathematical discrete optimisation problem for register allocation. We are interested in how effective current state-of-the-art graph-colouring approaches [5] are in comparison with approaches designed for highly irregular architectures and small embedded system programs [8]. In order to ensure a fair comparison between the methods we have used spill costs as a metric. Using spill costs provides a clear and solid mathematical comparison, and avoids the noise which is introduced by other optimisations run after the register allocation phase. Since spill costs are estimated by the compiler (either a-priori, or based on dynamic profiles) their accuracy is dependant on the accuracy of the estimator. However, since each of the methods we compare relies on the same spill costs this does not affect the fairness of our comparison.

As a testbed we have chosen the register allocation problems in the SPEC2000 benchmark suite, and IA-32 (which is fairly regular) as a target architecture. This combination of architecture and benchmark suite represents a worst case for our method. This extreme case was chosen in order to investigate how the old PBQP heuristic scaled under such conditions, and how the new heuristic would fare. To find a yardstick for the performance of both approaches we employed our optimal solver which can cope with the large register allocation problems in the SPEC2000 benchmark suite, and the constraints of IA-32.

The contributions of this work are outlined in the following:

1. We show that the heuristic introduced in [8] performs poorly for larger register allocation problems.
2. We describe a new heuristic for PBQP that is able to produce allocations of very high quality in reasonable time.

3. We introduce a new branch-and-bound solver for PBQP. Using this solver we are able to generate optimal register allocations for 97.4% of the functions in the SPEC2000 benchmarks.
4. Using the optimal solutions mentioned above we are able to show, for our testbed, i.e., IA-32 and SPEC2000, that the heuristics for graph-colouring and PBQP leave little room for further improvements.

The paper is organised as follows. Section 2 provides background information on the PBQP method of register allocation. Section 3 describes our new heuristic for PBQP. In Section 4 we explain the branch-and-bound algorithm for PBQP. Section 5 provides experimental evidence both for the performance of PBQP with the new heuristic, and the optimality of the achieved solutions. Section 6 surveys related work. In Section 7 we draw our conclusions.

2 Background

2.1 PBQP

The Partitioned Boolean Quadratic Programming (PBQP) problem [8,9] is a specialised Quadratic Assignment Problem (QAP). Consider a set of discrete variables $X = \{x_1, \ldots, x_n\}$ and their finite domains $\{\mathbb{D}_1, \ldots, \mathbb{D}_n\}$ where $m_i = |\mathbb{D}_i|$. A solution of PBQP is a simple function $h : X \rightarrow \mathcal{D}$ where $\mathcal{D}$ is $\mathbb{D}_1 \cup \ldots \cup \mathbb{D}_n$; for each variable x_i we choose an element d_i in $\mathbb{D}_i$. By imposing a total order for each discrete variable domain, sometimes we refer d_i by its ordinal number ranging from 1 to m_i.

The quality of a solution is based on the contribution of two sets of terms:

1. for assigning variable x_i to the element d_i in $\mathbb{D}_i$. The quality of the assignment is measured by a *local cost function* $c(x_i, d_i)$.
2. for assigning two related variables x_i and x_j to the elements d_i in $\mathbb{D}_i$ and d_j in $\mathbb{D}_j$. The quality of the assignment is measured by a *related cost function* $C(x_i, x_j, d_i, d_j)$.

Thus, the total cost of a solution h is given below:

$$f = \sum_{1 \leq i \leq n} c(x_i, h(x_i)) + \sum_{1 \leq i < j \leq n} C\left(x_i, x_j, h(x_i), h(x_j)\right) \tag{1}$$

The PBQP problem asks for an assignment of a minimum total cost.

We solve PBQP using matrix notation. A discrete variable x_i becomes a boolean vector $\boldsymbol{x_i}$ whose elements are zeros and ones and whose length is determined by the number of elements in its domain $\mathbb{D}_i$. Each 0-1 element of $\boldsymbol{x_i}$ corresponds to an element of $\mathbb{D}_i$. An assignment of x_i to d_i is represented by setting all elements of $\boldsymbol{x_i}$ to zero except the element of d_i, which is set to one. Hence, a possible assignment for a variable x_i is modelled by the constraint $\boldsymbol{x}_i^T \cdot \mathbf{1} = 1$ that restricts vectors $\boldsymbol{x_i}$ such that exactly one element of the vectors is assigned one; all other elements are set to zero.

The related cost function $C(x_i, x_j, d_i, d_j)$ is decomposed for each pair (x_i, x_j). The costs for the pair are represented as a matrix $\overline{C}_{ij}$. An element in the matrix corresponds

to an assignment (d_i, d_j). Similarly, the local cost function $c(x_i, d_i)$ is mapped to cost vectors $\boldsymbol{c_i}$. Quadratic forms and scalar products are employed to rewrite the objective function of Eq. (1) to

$$\begin{aligned} s.t.\ \forall 1 \leq i \leq n : \boldsymbol{x}_i &\in \{0,1\}^{|\mathbb{D}_i|} \\ \forall 1 \leq i \leq n : \boldsymbol{x}_i^T \cdot \mathbf{1} &= 1 \\ \min f = \sum_{1 \leq i \leq n} \boldsymbol{x}_i^T \cdot \boldsymbol{c}_i + \sum_{1 \leq i < j \leq n} \boldsymbol{x}_i^T \overline{C}_{ij} \boldsymbol{x}_j & \end{aligned} \tag{2}$$

In [8,9] a solver was introduced, which solves a sub-class of these problems optimally in $\mathcal{O}(nm^3)$, where n is the number of discrete variables and m is the maximal number of elements in their domains, i.e. $m = \max(m_1, \ldots, m_n)$. For a given problem, the solver eliminates stepwise discrete variables until the problem is trivially solvable, i.e. all quadratic forms $\boldsymbol{x}_i^T \overline{C}_{ij} \boldsymbol{x}_j$ are eliminated. Each elimination step requires a reduction. The solver has reductions R0, RI, RII, which are not always applicable. If no reduction can be applied, the problem becomes irreducible and a heuristic is applied, which is called RN. The heuristic chooses a beneficial discrete variable $\boldsymbol{x_i}$ and a good assignment for it by searching for local minima. The solution found is guaranteed to be optimal when the reduction RN is not used. Once the PBQP graph has been fully reduced the *backpropagation* phase is invoked to compute a final solution by reconstructing the original PBQP problem.

A PBQP problem can be represented as an undirected *PBQP graph* $G\langle N, E\rangle$. The nodes of the PBQP graph are discrete variables $\boldsymbol{x}_i$, for all i $(1 \leq i \leq n)$. In the graph there exists an edge (i, j) for $i < j$ if matrix $\overline{C}_{ij}$ is not the zero matrix.

2.2 PBQP for Register Allocation

Previous work in [9,8] described how the register allocation problem for irregular architectures can be mapped to PBQP. To understand this mapping it is easiest to view the PBQP graph as an extension of the interference graph.

Nodes in the PBQP graph represent symbolic registers as in an interference graph. In addition each node u has an associated cost vector $\boldsymbol{c}_u$ which describes the costs of each allocation option for u. In this work we assume that the first element of this vector will contain the cost of the spill option *sp*, and subsequent elements will contain the costs of each CPU register that is valid for u.

Edges in the PBQP graph represent constraints on the register allocation problem as before. There are usually two types of edges in an interference graph, interference or coalesce edges (which indicate that there is a benefit to assigning two non-interfering symbolic registers to the same CPU register). Edges in PBQP graphs have no explicit type, but are associated with cost matrices $\overline{C}_{uv}$. Each cost matrix $\overline{C}_{uv}$ represents the cost of pairs of allocations for nodes u and v. The contents of each cost matrix determines the effect of its edge on the final solution. Several common matrix forms for register allocation were given in [8].

For our work we employ only *interference matrices*. Interference matrices describe the costs of combinations of assignments for pairs of nodes which interfere. For two interfering nodes u and v the cost of an allocation (a_i, a_j) is infinite if a_i and a_j *alias*.

Pairs of registers r_i and r_j are said to alias if writing to one may affect the value of the other. By definition a register aliases with itself, and the spill option aliases with nothing (not even itself). For allocations where a_i and a_j do not alias the cost is zero. The interference matrix for nodes u and v is thus given by

$$\overline{I}_{uv}(i,j) = \begin{cases} 0, & \text{if } a_i \textit{ aliases } a_j \\ \infty, & \text{otherwise.} \end{cases} \tag{3}$$

As an example, consider the following subset of the IA-32 register architecture. It contains three 16bit registers named AX, BX and CX, each of which is aliased by two 8bit registers as depicted below.

AX		BX		CX	
AH	AL	BH	BL	CH	CL

If two nodes u and v have register option sets {sp, AH, AL, BL, CL} and {sp, AX, BX} respectively, the interference matrix I_{uv} is given by

$$I_{uv} = \begin{array}{c|ccc|l} & \textit{sp} & \textit{AX} & \textit{BX} & \\ & \downarrow & \downarrow & \downarrow & \\ \hline & 0 & 0 & 0 & \leftarrow \text{ sp} \\ & 0 & \infty & 0 & \leftarrow \text{ AH} \\ & 0 & \infty & 0 & \leftarrow \text{ AL} \\ & 0 & 0 & \infty & \leftarrow \text{ BL} \\ & 0 & 0 & 0 & \leftarrow \text{ CL} \end{array} \tag{4}$$

The rows of the matrix represent each allocation option for u (sp, AH, AL, BL and CL respectively). The columns represent each allocation option for v (sp, AX and BX respectively). Each element (i, j) gives the cost of an allocation (a_i, a_j).

The costs in the first row and column are all zero, since the spill option does not alias with anything. The second column contains two infinities since the AX register option for v aliases with both the AH and AL options for u. The third column contains only a single infinity since the BX register option for v only aliases with the BL option for u (we assume that the BH option has been denied to u by a register exclusion). The final row, representing the CL option for u, contains all zeros, because no register option for v aliases with CL.

Neither hardware registers nor register exclusions are explicitly represented in a PBQP graph. Instead, register exclusions remove options from nodes, reducing the length of the cost vectors and matrices. This in turn improves the speed of the PBQP solver.

3 PBQP Heuristic

Our initial experiments using PBQP to allocate registers for SPEC2000 revealed that the heuristic described in [8], Maximal Degree Minimum Solution (MDMS), performed poorly for these benchmarks.

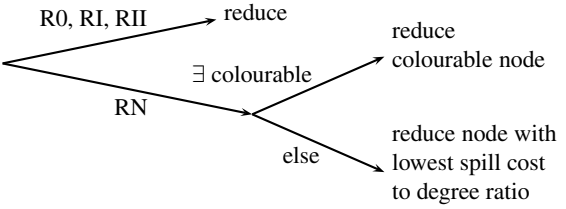

Fig. 1. Reduction Decision Tree

Previous work, presented in [9], showed that better results can be obtained by precomputing an RN reduction order using a traditional graph colouring approach. (Graph colouring approaches select and remove *colourable* nodes before *non-colourable* ones; a node u is considered colourable if no allocation of registers to u's neighbours precludes an allocation of a register to u itself.)

In the following we present a heuristic which is able to dynamically determine a reduction order for RN nodes based on colourability. At the core of our heuristic is an efficient and accurate method of determining colourability for irregular architectures.

The reduction order produced by our method is similar, though not identical, to that produced by graph colouring. Our reduction order is determined during the reduction phase based on the decision diagram depicted in Figure 1. During the reduction phase nodes of degree two or less are removed by the R0, RI and RII reductions. These reductions are performed irrespective of the colourability of nodes (on irregular architectures even low degree nodes may be non-colourable due to large register exclusion sets). Once all remaining nodes are of degree three or higher our RN heuristic is invoked to decide which node to reduce.

Our RN heuristic sorts the remaining nodes in descending order of degree. Based on this order it searches for a colourable node. If a colourable node is found it is removed from the PBQP graph and placed on the reduction stack. Sorting the nodes by degree ensures that the colourable node of highest degree is reduced. This improves the performance of the reduction process by maximising the number of nodes whose degrees are reduced.

If no colourable node is found we apply Brigg's spill heuristic [3] to reduce the node with the lowest ratio of spill cost to degree. No register assignment is made at this stage; instead our heuristic is optimistic in the sense that it defers the actual assignment until the backpropagation phase.

At the end of the reduction phase all nodes reside on the reduction stack. During the backpropagation phase the PBQP solver pops nodes from the reduction stack and reinserts them into the PBQP graph. As nodes are reinserted, the solver selects a decision vector that minimises the cost of the final solution. All nodes that we have classified as colourable (cf. the path via predicate "∃ colourable" in Figure 1) are guaranteed to be allocated a register. Nodes reduced by the R0, RI or RII reductions are solved optimally, and will be allocated a register if one is available. Nodes reduced due to the spill heuristic may or may not be allocated a register. Nodes which cannot be assigned a register will be assigned the *sp* option.

The performance of our heuristic depends on the efficient determination of colourability. For regular architectures colourability can be determined by comparing the degree of a node u to the number of available registers k. If $\mathit{degree}(u) < k$ then the node is colourable. For irregular architectures this condition is insufficient because each neighbour of u may exclude more than one register option from u (due to register aliasing). We describe below a fast and accurate method to determine the colourability of nodes for irregular architectures. Our method is based on the PBQP graph and its associated cost matrices.

We observe that a node u is colourable if either of the following two conditions hold.

(1) *The maximum number of colours which could be denied to u by a colouring of u's neighbours is less than the total number of colours available for u.*
(2) *There is at least one colour which is a valid choice for u, but not for any neighbour of u.*

To determine whether Condition (1) holds we calculate the maximum number of register choices that can be denied to node u by a colouring of u's neighbours. It is not practical to calculate this value exactly, because this would require enumerating all colourings of the neighbours of u. Instead we calculate a safe upper bound on Condition (1) by examining the worst case colourings of each of u's neighbours considered individually. This upper bound we call the impact upon u, denoted by impact_u.

If the adjacency set of a node u is given by $adj(u)$, and the impact of a single neighbour v by $\mathit{impact}_u(v)$, then the impact upon u by its neighbours is given by

$$impact_u = \sum_{v \in adj(u)} impact_u(v). \tag{5}$$

In order to calculate $\mathit{impact}_u(v)$, we need to look at the columns of the cost matrix $\overline{C}_{uv}$. The number of infinite elements in each column j represents the number of registers which could be denied to node u by selecting register r_j for node v (cf. Eq. (3)). For instance it can be seen in Eq. (4) that selecting the AX register for v (column 2) removes two options from u, whereas selecting the BX option (column 3) removes only one option, and the *sp* option (column 1) removes none.

We write $\mathit{inf\_count}(\overline{C}_{uv}, j)$ for the number of infinite cost elements in column j of matrix $\overline{C}_{uv}$, and m_u for the number allocation options for u. Then $\mathit{impact}_u(v)$ is given by

$$impact_u(v) = \max_{1 \le j \le m_u} \left\{ \mathit{inf\_count}(\overline{C}_{uv}, j) \right\}. \tag{6}$$

To determine whether Condition (2) holds for node u we determine the set of registers which cannot be denied to u by any colouring of its neighbours. This set we call $\mathit{safe\_regs}_u$. For Condition (2) to hold the cardinality of $\mathit{safe\_regs}_u$ must be greater than zero. To determine whether register r_i resides in $\mathit{safe\_regs}_u$, we examine row i of each of the neighbouring cost matrices of u. If row i of a matrix $\overline{C}_{uv}$ contains an infinite element, then r_i may be denied to u by some selection for v, thus r_i must be removed from $\mathit{safe\_regs}_u$.

In order to calculate $\mathit{safe\_regs}_u$, we place all register options except the spill element in $\mathit{safe\_regs}_u$. For each neighbour v of u we examine the cost matrix $\overline{C}_{uv}$. Each of the

rows of this matrix represents a valid register choice for u. For each row i that contains an infinite element we remove the corresponding register r_i from $\mathit{safe\_regs}_u$, since a certain colouring of v could exclude r_i from u. At the end of this process the registers remaining in $\mathit{safe\_regs}_u$ are those whose rows contained no infinite elements in any of the neighbouring matrices of u.

In Eq. (4) it can be seen that register AH (row 1) must not be in $\mathit{safe\_regs}_u$, since row 1 contains an infinity. Likewise registers AL and BL must not be in $\mathit{safe\_regs}_u$ because rows 2 and 3 contain infinities. Row 4 however, representing the CL option, does not contain an infinity, so CL is in $\mathit{safe\_regs}_u$.

Because the cost matrix construction process takes into account register classes and aliasing, these phenomena are implicitly considered in the determination of colourability. In addition, a positive effect of register exclusions, not considered in [3] and [5], is accounted for: if all neighbours of a node u are excluded from occupying a register that is a valid option for u, then u is colourable. On regular architectures register exclusions are rare and this effect would not significantly improve accuracy. However, for irregular architectures register exclusions are common and register sets are typically small. Considering this register exclusion effect can therefore yield a small increase in the accuracy of the colourability criterion.

An algorithm to calculate the colourability criterion according to Conditions (1) and (2) is given below. Therein $\mathit{options}(u)$ denotes the valid allocation options for node u, and sp denotes the spill element.

4 Branch-and-Bound for PBQP

Branch-and-bound is a general technique for solving discrete and combinatorial optimisation problems [10]. The general idea of branch-and-bound relies on two concepts. First, *branching* is a decomposition of the problem into sub-problems. Since branching is applied recursively to each of the sub-problems, the generated sub-problems form a tree called a *search tree*. Second, *bounding* is a fast way of finding lower bounds and upper bounds, respectively, for the optimal solution within sub-problems.

The branch-and-bound algorithm prunes sub-problems whose lower bounds are greater than the upper bound for any other sub-problem. If an upper bound for a sub-problem matches its lower bound, then the sub-problem has been solved. For finding the minimum all sub-problems of the search tree are either pruned or solved. Due to limited computational resources, sometimes not all sub-problems of the search-tree are either pruned or solved, and the branch-and-bound algorithm is terminated before finding the minimum of the objective function. In this case, the minimum lower bound and the minimum upper bound, among all non-pruned sub-problems, bound the minimum of the objective function. For branch-and-bound methods there are different ways to bound sub-problems and how to create and inspect the nodes in the search tree.

We extend the PBQP solver with branch-and-bound techniques. The approach introduced in [8] solves a PBQP problem optimally if R0, RI and RII reductions entirely decompose the problem. If no R0, RI or RII reduction can be applied in the reduction phase, the PBQP becomes irreducible and a heuristic selects a discrete variable $\boldsymbol{x}_l$ and chooses a concrete solution for $\boldsymbol{x}_l$ in $\mathbb{D}_l$. We refer to this step as RN reduction. If the

Algorithm 1. Colourability Criterion

Input: PBQP Graph G, node $u \in G$.
Output: Boolean value describing the colourability of u.

```
1: impact_u ← 0
2: safe_regs_u ← options(u)\sp
3: for all v ∈ adj(u) do
4:    impact_u(v) ← 0
5:    for all j ∈ {1, ..., |D_v|} do
6:       inf_count(C_uv, j) ← 0
7:       for all i ∈ {1, ..., |D_u|} do
8:          if C_uv(i, j) = ∞ then
9:             inf_count(C_uv, j) ← inf_count(C_uv, j) + 1
10:            safe-regs_u ← safe-regs_u\r_i
11:         end if
12:      end for
13:      if inf_count(C_uv, j) > impact_u(v) then
14:         impact_u(v) ← inf_count(C_uv, j)
15:      end if
16:   end for
17:   impact_u ← impact_u + impact_u(v)
18: end for
19: if (|safe-regs_u| > 0) ∨ (|impact_u| < |options(u)|) then
20:   colourable ← true
21: else
22:   colourable ← false
23: end if
```

problem domain is known, RN reductions based on heuristics are highly efficient and effective.

To find an optimal solution exhaustive enumeration was employed in [11]. The underlying idea of exhaustive enumeration is to use the ideas of the heuristic approach, i.e., R0, RI and RII reductions are applied until the problem is trivially solvable or an RN reduction needs to be applied. Instead of choosing a single solution for a discrete variable reduced by RN, all possible assignments of the discrete variable are enumerated. The complexity of exhaustive enumeration grows exponentially with the number of discrete variables reduced by RN. Despite the fact that for smaller problems with a small number of RN reductions this approach works sufficiently well, it becomes intractable for huge register allocation problems.

For PBQP a branch-and-bound approach is superior to an exhaustive enumeration approach because many assignments of discrete variables reduced by RN will be pruned. Furthermore, the solving techniques for PBQP allow a natural formulation of a branch-and-bound algorithm: A sub-problem is a PBQP problem which (1) cannot be further reduced by R0, RI and RII, and (2) is not trivially solvable. To each sub-problem we associate the discrete variable $\boldsymbol{x}_l$ which is selected by the RN reduction in the next reduction step and its concrete assignment.

A fragment of a search tree is depicted in Figure 2. The root of the tree represents the overall problem to be solved. If a problem has no RN reduction, the problem has no sub-problems and the tree consists of the root node only. Otherwise the reduced discrete variable x_{i_1} and its possible assignments ranging from 1 to m_{i_1} of the first RN reduction constitute the children of the root node. The discrete variable of the second RN reduction and its assignments constitute the grandchildren of the root node and so forth.

Note that a child of a sub-problem is a new sub-problem for which the discrete variable of its parent sub-problem was reduced, and R0, RI and RII reductions had been applied until the problem became irreducible. For the branch-and-bound algorithm we need to find lower and upper bounds of sub-problems, denoted by $\langle f^l_{i_k}, f^u_{i_k} \rangle$.

Before discussing the specific problem of finding lower and upper bounds of sub-problems, we derive the computation of lower and upper bounds of a general PBQP problem in matrix notation (see Eq. (2)). More formally, we want to find a lower bound f^l and upper bound f^u of f such that

$$f^l < f(x_1, \ldots, x_n) < f^u \tag{7}$$

holds for all possible assignments for discrete variables $x_i \in \mathbb{D}_i$, $(1 \leq i \leq n)$. Bounds can be simply derived by the observation that only one element of a cost vector $\boldsymbol{c}_i$ and matrix $\overline{C}_{ij}$ respectively, contributes to the objective function. Thus, lower and upper bounds of f are given by

$$f^l = \sum_{1 \leq i \leq n} \min \boldsymbol{c}_i + \sum_{1 \leq i < j \leq n} \min \overline{C}_{ij} \tag{8}$$

$$f^u = \sum_{1 \leq i \leq n} \max \boldsymbol{c}_i + \sum_{1 \leq i < j \leq n} \max \overline{C}_{ij}, \tag{9}$$

where $\min \boldsymbol{c}_i$ is the smallest element in $\boldsymbol{c}_i$ and in $\overline{C}_{ij}$, respectively, and $\max \boldsymbol{c}_i$ is the greatest element in $\boldsymbol{c}_i$ and in $\overline{C}_{ij}$, respectively.

The bounds for a sub-problem are computed by reducing the node $\boldsymbol{x}_l$. We choose a concrete element for vector $\boldsymbol{x}_l$ as assignment in $\mathbb{D}_l$. For a given assignment of $\boldsymbol{x}_l$ a sub-problem represented in matrix notation as given in Eq. (2) reduces to

$$\begin{aligned} s.t.\ \forall 1 \leq i \leq n, i \neq l : \boldsymbol{x}_i &\in \{0,1\}^{|\mathbb{D}_i|} \\ \forall 1 \leq i \leq n, i \neq l : \boldsymbol{x}_i^T \mathbf{1} &= 1 \\ \min f = \alpha + \sum_{1 \leq i \leq n, i \neq l} \boldsymbol{x}_i^T (\boldsymbol{c}_i + \boldsymbol{\Delta}_i) &+ \sum_{1 \leq i < j \leq n, i \neq l, j \neq l} \boldsymbol{x}_i^T \overline{C}_{ij} \boldsymbol{x}_j, \end{aligned} \tag{10}$$

where α is a constant, i.e., $\alpha = \boldsymbol{x}_l^T \boldsymbol{c}_l$, and Δ_i is a cost vector, i.e.,

$$\boldsymbol{\Delta}_i = \begin{cases} \overline{C}_{il} \boldsymbol{x}_l, & \text{if } i < l \\ \boldsymbol{x}_l^T \overline{C}_{li}, & \text{otherwise.} \end{cases} \tag{11}$$

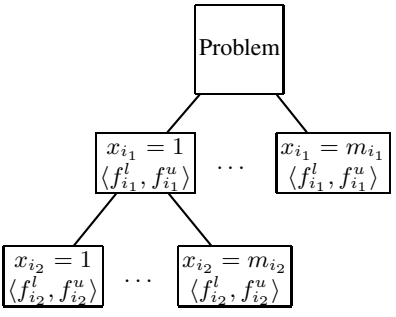

Fig. 2. Search Tree for PBQP

Because the discrete variable $\boldsymbol{x}_l$ is set to a concrete value, quadratic forms involving $\boldsymbol{x}_l$ become scalar products. Applying the lower and upper bounds of Eq. (8) and Eq. (9), we can deduce the following lower and upper bounds for a sub-problem.

$$f^l = \alpha + \sum_{1 \leq i \leq n, i \neq l} \min(\boldsymbol{c}_i + \boldsymbol{\Delta}_i) + \sum_{1 \leq i < j \leq n, i \neq l, j \neq l} \min \overline{C}_{ij} \tag{12}$$

$$f^u = \alpha + \sum_{1 \leq i \leq n, i \neq l} \max(\boldsymbol{c}_i + \boldsymbol{\Delta}_i) + \sum_{1 \leq i < j \leq n, i \neq l, j \neq l} \max \overline{C}_{ij} \tag{13}$$

The PBQP branch-and-bound algorithm is a standard branch-and-bound algorithm: sub-problems are classified in live and dead nodes in the search tree. Live nodes are leafs of sub-problems, which are not solved yet (i.e., lower and upper bound do not coincide). Dead nodes are nodes whose children have been already expanded. For running the algorithm we need an upper bound for the global minimum of the PBQP problem. The upper bound of the global minimum is initialised with infinity.

The live nodes are stored in a priority queue where the priority is determined by the lower bound of the sub-problem. The live node with the smallest lower bound is expanded first. The expansion of a node includes two steps. First, the children of the sub-problem are added to the tree and inserted to the priority queue if their lower bound is smaller than the upper bound of the global minimum. Second, the node is removed from the priority queue and it becomes dead. The branch-and-bound algorithm terminates if there are no nodes left in the priority queue, or if the upper bound of the global minimum is smaller then the smallest element in the priority queue.

We improved the standard algorithm by using the solution of a heuristic algorithm. In a pre-processing phase the search tree is expanded according to the solution of a given heuristic. Before running the canonical expansion the branch-and-bound algorithm has a tight upper bound for the global minimum and the search space becomes significantly smaller if the heuristic used is close to the optimum.

5 Experiments

In our experiments we compared the performance of three different PBQP solvers, i.e. a PBQP solver using the MDMS heuristic introduced in [8], a PBQP solver using our new

Table 1. Number of Functions in SPEC2000

Benchmark	Total	Pairs	Empty	Remaining
164.gzip	89	1	14	74
175.vpr	266	4	47	215
176.gcc	1965	46	367	1552
181.mcf	26	0	2	24
186.crafty	109	39	9	61
197.parser	323	0	27	296
252.eon	1257	0	570	687
253.perlbmk	1015	1	208	806
254.gap	852	6	122	724
255.vortex	923	10	93	820
256.bzip2	74	0	14	60
300.twolf	191	0	17	174
total	7090	107	1490	5493

heuristic (see Sec. 3), a PBQP solver using branch-and-bound (see Sec. 4), and a state-of-the-art graph colouring method described in [5]. The four approaches are compared in terms of number of spills, spill costs, and solve time. We do not consider the effects of register allocation on code size nor on runtime of benchmark programs since register allocation works in concert with other standard compiler optimisations. Measuring the genuine effects of register allocation on code size and runtime would be overlayed with noise. Taking the spill cost as a measurement gives a solid mathematical comparison.

To obtain a comparison of our methods each solver was used to produce allocations for the SPEC2000 benchmarks. The interference graphs, annotated with spill costs and register constraints, were obtained from the GCC 3.3.6 compiler, and passed to the solvers. Empty interference graphs and graphs requiring register pairs were not taken into account, leaving 5493 graphs for our experiments. A quantitative summary of the interference graphs is given in Table 1. Each solver calculates an allocation and produces a raw assignment of registers and spills to symbolic registers, as well as timing information. Our raw allocations were processed to check for correctness and to extract spill costs and other information.

The cost model used for our experiments is highly regular. Our solvers assign only registers of the same size as the allocation candidate (in contrast to GCC's allocator which stores all non-spilled symbolic registers in 32-bit registers). All valid register options are assumed to have zero cost (except the spill option, whose cost is given by GCC's spill cost estimator), and only interference constraints are modelled. Such a regular cost model represents a worst-case scenario for PBQP, which performs better on more constrained architectures.

Summaries of the allocations produced by each of our solvers are given in Table 2. The first three columns describe the total spill cost for each benchmark individually and overall, using each heuristic solver. The next three columns give the number of spills produced by the solvers. The final three columns show the time taken to produce the allocations.

Table 2. Raw Allocation Results for the SPEC2000 Benchmarks

	Spill Cost			Spills			Allocation Time (ms)		
Benchmark	MDMS	New	GrCo	MDMS	New	GrCo	MDMS	New	GrCo
164.gzip	120438	60175	60838	121	114	118	6.9	9.2	3.1
175.vpr	521770	330724	328358	690	710	704	27.5	39.8	12.4
176.gcc	1431081	720548	728731	3078	3341	3335	322.6	532.4	133.2
181.mcf	98796	69440	69445	81	82	83	2.4	3.2	1.1
186.crafty	73491	27978	28267	149	153	153	10.5	15.1	4.9
197.parser	221732	162962	168847	508	525	525	35.9	52.9	15.1
252.eon	446646	366810	367965	815	826	816	34.7	53.0	16.6
253.perlbmk	758888	323161	334957	910	925	921	93.4	126.4	38.8
254.gap	1873241	1090693	1099054	1822	1929	1947	118.8	163.9	49.5
255.vortex	424300	238188	239328	972	983	977	49.9	64.8	23.6
256.bzip2	67531	26944	27349	134	146	146	7.1	10.1	3.2
300.twolf	1085151	560064	564956	1110	1194	1203	91.7	155.7	33.1
total	7123065	3977687	4018095	10390	10928	10928	801.5	1226.5	334.5

It can be seen from the final row of Table 2 that our new heuristic produces a spill cost 44% lower than that of the MDMS heuristic, and 1% lower than graph colouring. This result represents a large improvement over the previous heuristic, and places PBQP on a par with graph colouring in terms of the allocations generated.

The poor performance, in terms of spill cost, of the MDMS heuristic compared to graph colouring has not been observed before. Previous work on PBQP for register allocation using this heuristic, given in [8], was carried out using embedded systems benchmarks. These benchmarks have smaller interference graphs than those of SPEC2000. For such graphs the RN reduction rule is seldom invoked, and the choice of RN heuristic has less impact upon the final result.

The MDMS heuristic generates fewer spills overall than either of the other methods, despite producing a worse allocation overall. This occurs because the MDMS heuristic always reduces the node of highest degree, regardless of whether the node is colourable. Choosing such a node lowers the degree of the maximum number of neighbours, reducing the chance of further spills. No effort is made to decide whether this is a good spill decision however, which leads to a poor final allocation.

Both PBQP heuristics are considerably slower than graph colouring. The MDMS heuristic takes a factor of 2.4 times longer than graph colouring over all benchmarks. An original naive implementation of our heuristic required a factor of 25 times longer than graph colouring. We determined however that most of this time was spent in unnecessary re-evaluations of matrices. By implementing a caching strategy for per-matrix information and using lazy evaluation to update these caches we were able to reduce the time taken to the present factor of 3.7 times longer than graph colouring. Previous work on register allocation [12] showed that the time taken to solve the graph colouring problem is only a small fraction of the overall allocation time. As such we would not expect our method to significantly increase the total compile time.

Table 3 gives the results produced by each solver for those functions which we were able to solve optimally. The first column gives the number of functions solved optimally

Table 3. Optimal Costs, Spills and Comparisons

Benchmark	Functions	Spill Cost			Spills		
		Optimal	New	GrCo	Optimal	New	GrCo
164.gzip	73	47603	47605	48268	105	104	108
175.vpr	207	286466	291553	289036	600	605	599
176.gcc	1491	595785	602444	611952	2212	2281	2272
181.mcf	24	69404	69440	69445	81	82	83
186.crafty	59	23702	23789	24078	120	121	123
197.parser	280	144464	148540	154400	320	329	329
252.eon	686	366613	366723	367874	775	782	775
253.perlbmk	800	429530	441131	451981	795	825	833
254.gap	710	1033923	1042452	1049729	1634	1672	1688
255.vortex	815	225461	228250	229673	862	875	869
256.bzip2	58	23179	24344	27349	92	99	97
300.twolf	149	368726	375961	378710	528	544	547
total	5352	3614856	3662232	3699885	8124	8319	8323

for each benchmark and overall. The next three columns give the spill costs for the optimal solution, PBQP using our new heuristic, and graph colouring. The final three columns give the number of spills generated by each of the methods.

Overall we were able to solve 97.4% of the functions in the SPEC2000 benchmarks optimally over a period of about a day. From the final row it can be seen that the optimal spill cost is 1.3% lower than that produced by our heuristic, and 2.3% lower than that of graph colouring. Our heuristic never generated spill costs more than 3% above the optimal for any benchmark (the highest was perlbmk at 2.7%). Graph colouring never generated an allocation more than 7% above the optimal for any benchmark (the highest was parser at 6.9%). The small margins between the optimal spill costs and the heuristics show that there is little room for improvements for a fairly regular architecture such as IA-32.

6 Related Work

Graph colouring approaches [2,3] are a success story for RISC architectures with large register banks and an orthogonal instruction set. However, attempts to adapt graph colouring to irregular architectures have produced ad-hoc modifications which fail to provide a unified method for dealing with irregularities. Each of these methods is able to deal with a certain subset of irregularities at the expense of breaking from the simple graph colouring analogy.

Smith et al. [5] introduced a new colourability criterion for irregular architectures which is able to determine colourability for architectures featuring register classes and aliasing. However, their approach cannot deal with complex constraints between two symbolic registers such as pairing or dedicated registers. Runeson and Nyström [13] present a retargetable graph-colouring register allocator based on the $\langle p, q \rangle$ test, which is similar to the work in [5]. Other techniques for graph colouring such as the technique

introduced by Koseki et al. [14] modifies the selection phase to increase the likelihood that symbolic registers are given their preferred registers. However, their algorithm can only deal with certain aspects of irregular architectures.

Register allocation based on Integer Linear Programming (ILP) was introduced by Goodwin and Wilken [15]. The approach maps the register allocation problem to an integer linear program, which is solved by CPLEX, a commercial solver for generic ILP problems. The work was extended by Kong and Wilken [6] for irregular architectures. Recently, in [16], an approach was introduced which uses a progressive solver for solving register allocation problems based on multi commodity network flows. With their approach not all possible constraints occurring in irregular architectures can be modelled.

Recently, register allocation approaches exploiting the tree structure of SSA graphs have been investigated [17] stating that the graph colouring problem is solvable in polynomial time without considering coalescing costs at phi-nodes. However these approaches do not consider any irregularities.

Most of the work in this paper builds on work described in [8,11]. The PBQP optimisation problem accommodates for the needs of solving the register allocation problem for a wide range of irregularities. It is a fairly comprehensive approach. However, the exhaustive enumeration approach introduced in [11] is intractable for larger benchmarks and the approach introduced in [8] has a poorly performing heuristic for larger benchmarks and more regular architectures. Both problems have been resolved by this work.

7 Future Work and Conclusion

In this paper we have presented a new PBQP heuristic for register allocation. For larger benchmarks and moderately irregular architectures the new heuristic performs significantly better than the MDMS heuristic introduced in [8]. We also describe a new algorithm for PBQP based on branch-and-bound. The branch-and-bound algorithm was extended to use a heuristic to find a tight upper bound for its global minimum. With this technique we show that 97.4% of the register allocation problems in the SPEC2000 integer benchmark suite can be solved optimally in less than a day.

With the given framework there is still the algorithmic challenge to solve every register allocation problem in SPEC2000 optimally. This challenge might be achieved by exploring some decomposition properties of PBQP, i.e. a PBQP problem disintegrates into independent sub-problems during the reduction phase. By solving the sub-problems independently the search space of the branch-and-bound solver will be significantly reduced.

We plan to integrate this method into a modern optimising compiler in order to evaluate our method's effects on code size and execution speed. Given the closeness of the spill costs we have seen we do not expect significant deviation between our method and graph colouring for IA-32 using these metrics. However we plan to apply both methods to more irregular architectures where we would expect a greater variation.

With the optimal solution as a yardstick we have shown that current graph colouring heuristics [5] for irregular architectures and the new PBQP heuristic introduced in

this work are on average 2% from the optimal solution. In future there will be very little room for further progress in finding better optimisation heuristics for moderately irregular architectures.

References

1. Moore, G.: 40th Anniversary of Moore's Law. Press Conference (2005)
2. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages **6** (1981) 47–57
3. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. **16**(3) (1994) 428–455
4. Briggs, P., Cooper, K.D., Torczon, L.: Coloring register pairs. ACM Lett. Program. Lang. Syst. **1**(1) (1992) 3–13
5. Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (2004) 277–288
6. Kong, T., Wilken, K.D.: Precise register allocation for irregular architectures. In: MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitectures, Los Alamitos, CA, USA, IEEE Computer Society Press (1998) 297–307
7. Koes, D., Goldstein, S.C.: A progressive register allocator for irregular architectures. In: CGO '05: Proceedings of the international symposium on Code generation and optimization, Washington, DC, USA, IEEE Computer Society (2005) 269–280
8. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems, New York, NY, USA, ACM Press (2002) 139–148
9. Eckstein, E.: Code Optimizations for Digital Signal Processors. PhD thesis, Institute of Computer Languages, Compilers and Languages Group, Vienna University of Technology (2003)
10. Murty, K.G.: Operations Research: Deterministic Optimization Models. Prentice Hall (1995)
11. Hirnschrott, U., Krall, A., Scholz, B.: Graph -coloring vs.optimal register allocation for optimizing compilers. Proceedings of the Joint Modular Language Conference (2003) 202–213
12. Briggs, P.: Register allocation via graph coloring. Technical Report TR92-183, Department of Computer Science, Rice University (1998)
13. Runeson, J., Nyström, S.: In Software and Compilers for Embedded Systems (SCOPES). In: Retargetable Graph-Coloring Register Allocation for Irregular Architectures. Volume 2826 of Lecture Notes in Computer Science. Springer Press, Klagenfurt, Austria. (2003) 240–254
14. Koseki, A., Komatsu, H., Nakatani, T.: Preference-directed graph coloring. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (2002) 33–44
15. Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0–1 integer programming. Softw. Pract. Exper. **26**(8) (1996) 929–965
16. Koes, D., Goldstein, S.C.: A global progressive register allocator. In: PLDI '06: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, Ottawa, ON, Canada, ACM Press (2006) (to appear)
17. Pereira, F.M.Q., Palsberg, J.: Register allocation via coloring of chordal graphs. In: APLAS'05: Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems, Springer Spress (2005) 315–329