Fast Source-Level Data Assignment to Dual Memory Banks

Alastair Murray
a.c.murray@sms.ed.ac.uk

Björn Franke
bfranke@inf.ed.ac.uk

University of Edinburgh
School of Informatics
Institute for Computing Systems Architecture

Abstract

Due to their streaming nature memory bandwidth is critical for most digital signal processing applications. To accommodate for these bandwidth requirements digital signal processors are typically equipped with dual memory banks that enable simultaneous access to two operands if the data is partitioned appropriately. Fully automated and compiler integrated approaches to data partitioning and memory bank assignment, however, have found little acceptance by DSP software developers. This is partly due to their inflexibility and inability to cope with certain manual data pre-assignments, e.g. due to I/O constraints. In this paper we present a different and more flexible approach, namely source-level dual memory assignment where code generation targets DSP-C, a standardised C language extension widely supported by industrial C compilers for DSPs. Additionally, we present a novel partitioning algorithm based on soft colouring *that is more efficient and scalable than the currently known best integer linear programming algorithm, whilst achieving competitive code quality. We have evaluated our scheme on an Analog Devices TigerSHARC DSP and achieved speedups of up to 1.57 on 13 UTDSP benchmarks.*

1. Introduction

Digital signal processors are domain specific microprocessors optimised for embedded digital signal processing applications. The demand for high performance, low power and low cost has led to the development of specialised architectures with many non-standard features exposed to the programmer. With the recent trend towards more complex signal processing algorithms and applications high-level programming languages, in particular C, have become a viable alternative to the predominant assembly coding of earlier days. This, however, comes at the price of efficiency when compared to hand-coded approaches [6].

Optimising compiler technology has played a key role in enabling high-level programming for DSPs. Many of the newly developed approaches to code generation for specialised DSP instructions [4], DSP specific code optimisation [12] and instruction scheduling [17] have transitioned out of the research labs and into product development and production.

The situation, however, is different with compiling techniques targeting one of the most distinctive DSP features: dual memory banks. Designed to enable the simultaneous fetch of two operands of, e.g., a multiply-accumulate operation they require careful partitioning and mapping of the data to unfold their full potential. While the dual memory bank concept has found active interest in the academic community this work does not seem to have found its way into production compilers. Instead, DSP specific language extensions of the ISO C language such as DSP-C [2] and Embedded C [9] that shift the responsibility for data partitioning and mapping to the programmer are widely embraced by industry. We believe this is partly due to the fact that fully automated and compiler integrated approaches to memory bank assignment ignore that programmers require control over the mapping of certain variables, for example, used for I/O buffering and tied to a specific bank. Additionally, programmers would frequently like to specify a partial mapping to achieve a certain effect on particular regions of code, and leave the rest to the compiler. To our knowledge, none of the previously published memory bank assignment schemes allows for this level of interaction.

Here we follow a different approach, namely explicit memory bank assignment as a source-level transformation operating on ISO C as input language and generating output in DSP-C. Next to its inherent portability the advantage of this high-level approach is the ease with which manual pre-assignment of variables, i.e. coercing them into a specific user-directed bank, can be accomplished. On the other hand, a high-level approach like the one presented in this paper needs to address the difficulty of having to cope with "unpredictable" later code optimisation and generation

```
void lmsfir(float input[], float output[],
            float expected[], float coefficient[],
            float gain)
{
  /* Variable declarations omitted */

  sum = 0.0;
  for (i = 0; i < NTAPS; ++i) {
    sum += input[i] * coefficient[i];
  }
  output[0] = sum;
  error = (expected[0] - sum) * gain;
  for (i = 0; i < NTAPS-1; ++i) {
    coefficient[i] += input[i] * error;
  }
  coefficient[NTAPS-1] = coefficient[NTAPS-2] +
                         input[NTAPS-1] * error;
}
```

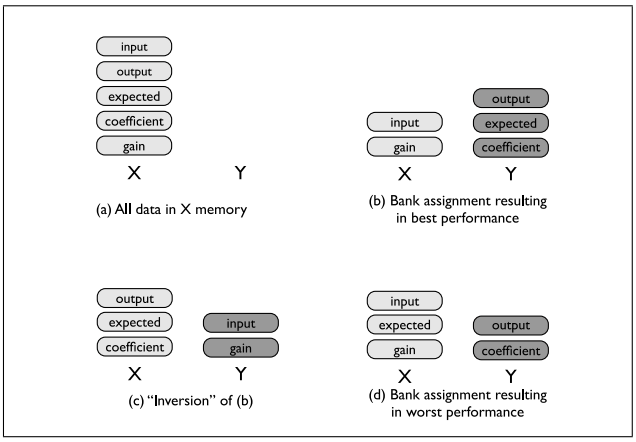


Figure 1. `lmsfir` function with four memory bank assignments resulting in different execution times.

stages that may interact with the earlier bank assignment.

In this paper we model possible interactions and extend the variable interference graph introduced in [15] with non-dataflow edges, effectively introducing "uncertainty" into our model. This also helps balance the memory bank utilisation. Where previous approaches, e.g. [13, 7], aim for optimality of the generated partitioning we show that an optimal solution according to the standard interference graph model does not necessarily result in the fastest program in practice. We have exhaustively enumerated all legal memory bank assignments for a set of benchmark applications and demonstrate that (a) there is room for improvements over "optimal" algorithms, and (b) different solutions equally "optimal" in the standard model may result in greatly different actual performance. Finally, we replace an expensive integer linear programming based algorithm (as in [13]) with a more efficient and scalable stochastic soft colouring algorithm. Not only is compiler efficiency improved, but we demonstrate that the achievable code quality is highly competitive.

1.1. Motivation

Efficient assignments of variables to memory banks can have a significant performance impact, but are difficult to determine. For instance, consider the example in figure 1. It shows the `lmsfir` function from the *UTDSP lmsfir_8_1* benchmark. This function has five parameters that can be allocated to two different banks. Local variables are stack allocated and outside the scope of explicit memory bank assignment. On the right side of figure 1 four of the possible legal assignments are shown. In the first case, as illustrated in figure 1(a), all data is placed in the *X* memory bank. This is the default case for many compilers when no explicit memory bank assignment is specified. Clearly, no advantage of dual memory banks can be taken and this assignment results in an execution time of 100 cycles for our Analog Devices TigerSHARC TS-101 platform. The best possible assignment is shown in figure 1(b), where `input` and `gain` are placed in *X* memory and `output`, `expected`, and `coefficient` in *Y* memory. Simultaneous accesses to the `input` and `coefficient` arrays have been enabled and, consequently, this assignment reduces the execution time to 96 cycles. Interestingly, an "equivalent" assignment scheme as shown in figure 1(c) that simply swaps the assignment between the two memory banks does not perform as well. In fact, the "inverted" scheme derived from the best assignment results in an execution time of 104 cycles, a 3.8% slowdown over the baseline. The worst possible assignment scheme is shown in figure 1(d). Still, `input` and `coefficient` are placed in different banks enabling parallel loads, but this scheme takes 110 cycles to execute, a 9.1% slowdown over the baseline.

This example demonstrates how difficult it is to find the best source-level memory bank assignment. Source-level approaches cannot analyse code generation effects that only occur later in the compile chain, but must operate a model generic enough to cover most of these. In this paper, we propose a refined variable interference graph construction together with a fast and scalable soft colouring algorithm capable of handling complex DSP applications and allowing for partial pre-assignments where required.

The rest of this paper is structured as follows. In section 2 we discuss the large body of related work. Relevant background material is explained in section 3. The source-level memory bank assignment scheme is introduced in section 4, with two different colouring techniques described in sections 5 and 6 before we present our results in section 7. Finally, we summarise and conclude in section 8.

2. Related Work

An early attempt to solve the dual memory bank assignment problem was undertaken by Saghir *et al*. [15]. They produced a low-level solution that performs a greedy minimum-cost partitioning of the variables using the loop-nest depth of each interference as a priority heuristic. The problem was formulated as an interference graph where two nodes interfere if they represent a potentially parallel access in a basic block. This is a very intuitive representation of the problem and has been used in many other solutions since.

Gréwal *et al*. used a highly-directed genetic algorithm to provide a solution to dual memory bank assignment [8]. They used a constraint satisfaction problem as a model, with hard constraints such as not being able to exceed memory capacity, and soft constraints such as not wanting interfering variables in the same memory. The genetic algorithm is then used to find the optimal result in terms of this model. Due to technical limitations, however, this method was only evaluated on randomly generated synthetic benchmarks.

Another approach by Ko and Bhattacharyya uses synchronous data flow specifications and the simple conflict graphs that accompany such programs. Three techniques were proposed, a traditional colouring algorithm, an integer linear programming based algorithm and a low complexity greedy heuristic using variable size as a priority metric. Though for all benchmarks the techniques were evaluated against there exists a two-colouring, so the techniques are not demonstrated to work on hard problems.

Leupers and Kotte described an integer linear programming solution [13] prior to Ko and Bhattacharyya [10]. While the other techniques described worked at level of the compiler IR, this technique runs after the compiler back-end has generated object code. This lets it consider spill code and ignore accesses which don't reach memory. The assignment problem is modelled as an interference graph where two variables interfere if they are in the same basic block and there is no dependence between them. The interference is weighted according to the number of potentially parallel memory accesses.

More recently Gréwal *et al*. described a more accurate integer linear programming model for DSP memory assignment [7]. The model described here is considerably more complicated than the one previously presented by Leupers and Kotte [13] but provides larger improvements. Both of these techniques were evaluated using DSPstone but with different base architectures, however the two architectures used are sufficiently similar for a comparison to be meaningful. The simpler integer linear program achieved speedups between 4% and 17%, the more complex model achieved speedups between 7% and 42%.

Finally Sipkovà describes a technique [16] that operates at a higher-level than the previously described methods. It performs memory assignment on the high-level IR, thus allowing the colouring method to be used with each of the back-ends within the compiler. The problem is modelled as an independence graph and the weights between variables take account of both execution frequency and how close the two accesses are in the code. Several different solutions, based on a max-cut formulation, were evaluated on a subset of the DSPstone benchmark suite and two fixed-point FFTs.

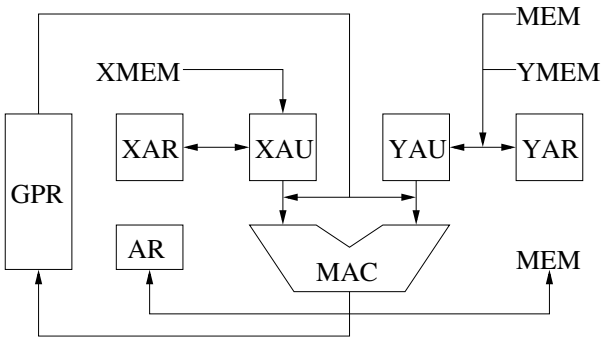


Figure 2. Example DSP processor architecture with dual-input memory data path and MAC unit [3].

3. Background

3.1. Dual Memory Banks

Typical digital signal processing operations such as convolution filtering, dot product computations and various matrix transformations make intensive use of *multiply-accumulate (MAC)* operations, i.e. computing the product of two numbers and adding the product to an accumulator.

Digital signal processors are application-specialised microprocessors designed to most efficiently support digital signal processing operations. Among the most prominent architectural features of DSPs are support for MAC operations in the instruction set and dual memory banks that enable simultaneous fetching of two operands. Provided the data is appropriately partitioned across the two memory banks this effectively doubles the memory bandwidth and ensures efficient utilisation of the DSP datapath.

Figure 2 shows a generic DSP architecture with dual memory banks *X* and *Y*. These two banks are accessed via the *X* and *Y* addressing units, which may support DSP specific post-increment addressing modes.

3.2. DSP-C and Embedded C

DSP-C [2] and its later extension *Embedded C* [9, 3] are

sets of language extensions to the ISO C programming language that allow application programmers to describe the key features of DSPs that enable efficient source code compilation. As such, DSP-C includes C-level support for fixed point data types, circular arrays and pointers, and, in particular, divided or multiple memory spaces.

DSP-C uses address qualifiers to identify specific memory spaces in variable declarations. For example, a variable declaration like

```
int X a[32];
```

defines an integer array of size 32, which is located in the *X* memory. In a similar way, the address qualifier concept applies to pointers, but now up to two address qualifiers can be provided to specify where the pointer and the data it points to is stored. For example, the following pointer declaration

```
int X * Y p;
```

describes a pointer `p` that is stored in *Y* memory and points to integer data that is located in *X* memory. For unqualified variables a default rule will be applied (e.g. to place this data in *X* memory). A common constraint on assigning variables to memory banks is that only global variables may be placed on a specific memory bank. This is the case on our target architecture, as the compiler is unable to split the stack across both memory banks so all stack allocated variables must be placed in the first memory bank.

Note that DSP-C does not specify any predefined keywords to be used as address qualifiers, as the actual memory segmentation is left to the implementation. Commonly used address qualifiers are, for example, `X` and `Y`, or `pm` and `dm`.

4. Methodology

Our memory bank assignment schemes comprises the following the stages:

1. **Group Forming.** In this stage groups of variables that must be allocated to the same memory bank due to pointer aliasing are formed.
2. **Interference Graph Construction.** An edge-labelled graph representing potential simultaneous accesses between variables is constructed during this stage.
3. **Colouring of the Interference Graph.** Finally, the nodes of the interference graph are coloured with two colours (representing the two memory banks) such as to maximise the benefit from simultaneous memory accesses.

Of these three stages only stage one is critical for correctness, whereas approximations are acceptable for stages two and three, i.e. an inaccurate interference graph or a non-optimal colouring still result in correct code that, however, may or may not perform optimally.

4.1. Group Forming

Group forming is the first stage in our memory bank assignment scheme. It is based on pointer analysis and summarises those variables in a single group that arise through the *points-to* sets of one or more pointers. All variables in a group must be allocated to the same bank to ensure type correctness of the memory qualifiers resulting from our memory bank assignment.

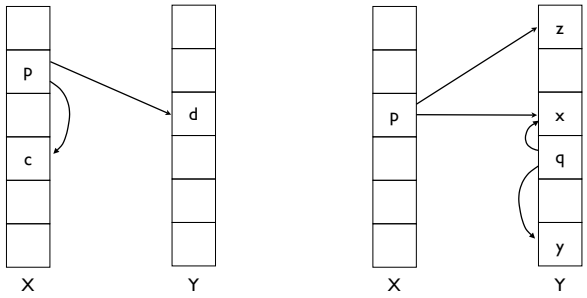
Figure 3 illustrates this concept. In figure 3(a) the pointer *p* may point to *c* or *d*. However, *c* and *d* are stored in memory banks *X* and *Y*, respectively. This eventually causes a conflict for *p* because both the memory bank where *p* is stored and the bank where *p* points to must be statically specified. Thus, *p* must only point to variables located in a single bank. A legal assignment would place *c* and *d* in the same bank as a result of previous grouping. This grouping is shown in figure 3(b) for two pointers *p* and *q*. In this example *p* may point at variables *x* and *z* at various points in the execution of a program and, similarly, *q* is assumed to point at *x* and *y*. Grouping now ensures that *x* and *z* are always stored in the same bank (due to *p*), and also *x* and *y* (due to *q*). By transitivity, *x*, *y* and *z* have to be placed in the same memory bank. Note that *p* and *q* themselves can be stored in different memory banks, only their targets must be grouped and located in a single memory bank.

Algorithm 1 Group Forming(Variables V)

Require: Points-to for all pointers
Ensure: Type-safe variable grouping

1: **for all** $v_k \in V$ **do**
2: Form singleton group g_k containing v_k
3: **end for**
4: $L \leftarrow \{p | p \text{ is a pointer} \wedge p \in V\}$
5: **while** $L \neq \emptyset$ **do**
6: Select $p \in L$
7: Merge groups(points-to(p))
8: **end while**

In algorithm 1 a working list algorithm for group forming is presented. It is assumed that points-to sets for all pointer variables are available, e.g. through prior pointer analysis of the program. The algorithm operates on the set of variables V to group. Initially, the algorithm places each variable v_k in a singleton group g_k, these are then merged into larger groups. For each pointer p in the set of variables V the points-to set is calculated and the groups of the corresponding variables merged. This process is repeated until



(a) Incompatible pointer assignments. (b) Pointer induced variable grouping.

Figure 3. Incompatible pointer assignments and pointer induced grouping.

all pointer have been visited. The algorithm can be efficiently implemented and the main costs usually arise from the required pointer analysis phase.

"Aliasing" of different actual parameters from multiple call sites to a single set of formal function parameters are handled analogously.

4.2. Interference Model

To be able to effectively assign groups of variables to memory banks it is necessary to build an interference graph that represents the memory accesses in the program. This is done statically by taking the dataflow dependence graph for each expression and marking each pair of memory or variable accesses with no dependence between them as potentially interfering (see figure 4). This represents cases where loads or stores could be scheduled in parallel. Each of these potential interferences is given a weight that is equal to the estimated number of times that the expression will be executed. This estimate is determined by calculating each loop's iteration count (or using a value of 100 if the exact count can not be statically determined), and assuming all branches are taken with 50% probability. The estimated call count for each function is also calculated this way by calculating how many times each call site is executed. This variable interference graph is then reduced to a group interference graph using the previous assignments. This approximate information is sufficient for determining which groups of variables are the most important.

We optionally extend this model to be able to handle variables that must be assigned to a fixed memory bank, e.g. *automatic* variables which must be placed on the stack or variables which the programmer has already assigned to a specific memory bank. We extend the model with a single additional node per fixed groups of variables, if this group is the automatic variables we name the node the *automatic node*. It is not possible to assign these nodes to a memory bank, they are always placed on a fixed one, but it is possible to interfere with them. Thus it may be possible to more accurately determine assignments for other groups.

5. ILP Colouring

5.1. Single Solution

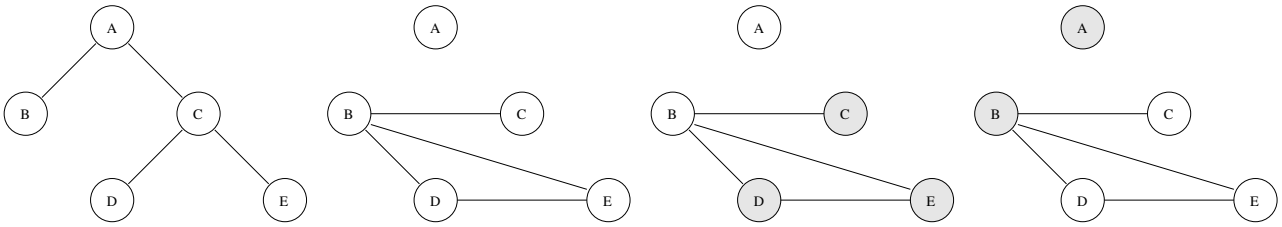
A reference Integer Linear Programming (ILP) colouring approach that is approximately equivalent to the model by Leupers and Kotte [13] is implemented. An ILP model is constructed from the interference graph, $I = (G, E)$ where G is the set of groups of variables (vertices in the graph) and E is the weighted interferences between them.

$$\forall g_i \in G: \quad X_i, Y_i = \begin{cases} 1, & \text{if } g_i \text{ is placed in bank } X/Y \\ 0, & \text{otherwise} \end{cases}$$
$$X_i + Y_i = 1$$

This ensures that each group g_i is placed in exactly one memory bank as if it was placed on neither or both then $X_i + Y_i$ would not equal 1.

$$\forall g_i, g_j \in G: \quad U_{ij} = \begin{cases} 1, & \text{if } X_i = Y_j \text{ or } Y_i = X_j \\ 0, & \text{otherwise} \end{cases}$$
$$U_{ij} \leq 2 - X_i - X_j$$
$$U_{ij} \leq 2 - Y_i - Y_j$$
$$U_{ij} \geq X_i - X_j$$
$$U_{ij} \geq Y_i - Y_j$$

The above constraints ensure that U_{ij} is set to 1 if and only if groups g_i and g_j are placed in different memory banks. The first two constraints ensure that $U_{ij} \leq 1$ if g_i and g_j are in different banks or 0 otherwise. The next two constraints set $U_{ij} \geq 1$ if g_i and g_j are on different banks or 0 otherwise. When combined these ensure that U_{ij} is always set correctly.



(a) Expression dependence tree (b) Expression interference graph (c) Coloured interference graph (d) Alternate colouring

Figure 4. Mapping dependances to potential interferences.

The linear program solver then optimises the following objective function while obeying the above constraints. As the values of the U_{ij} variables are set by the values of the X_i and Y_i variables it is only these latter variables that the solver may set to attempt to optimise the objective function. Thus it is effectively assigning variables to memory banks and attempting to maximise the available parallelism.

$$\max\left(\sum_{0}^{i}\sum_{0}^{j} U_{ij}\cdot W_{ij}\right) \text{ where } W_{ij}\in E$$

Here W_{ij} is the interference weight associated with each edge $(g_i, g_j) \in E$ calculated as described in section 4.2, if there is no edge between g_i and g_j in E then a weight of 0 is used. By maximising the above equation the linear programming solver is finding a set of assignments for the variables in G that favours placing variables with a high interference on different memory banks. This means that it should be possible to perform the most critical memory operations in parallel.

This set of assignments is not necessarily truly optimal though, it is only an optimal solution in terms of the interference graph. This ILP model is based on the model described in the Leupers and Kotte paper [13], their model used an interference graph built after the back-end of the compiler had run so it is a reasonably accurate model of the potential parallelism in the program. However, in our technique we build the interference model based on the program's source-code, the entire target compiler still has to be run on the program after variables have been assigned to memory banks. Thus this technique is re-evaluated in section 7 to ensure that it is still a valid colouring model at the high-level.

5.2. Multiple Solutions

Building the interference graph at a high-level also means that the problem is less constrained, this means that there may be many optimal solutions to the ILP model above. For example, if a node is completely disconnected in the interference graph then the score to be maximised by the linear solver will be the same whichever memory bank that group of variables is assigned to.

Integer linear program solvers generally work by first reducing as much of the program to a non-integer linear problem that can be solved quickly and then using a branch-and-bound technique to solve what remains. If there are multiple optimal solutions then they may only be found during the branch-and-bound stage, where it is possible to keep on searching even after an optimal solution has been found. However, in the process of reducing the integer problem to a non-integer one many of the alternate optimal solutions may be lost and there will be fewer solutions for the branch-and-bound technique to find.

For the above memory bank assignment ILP model the branch-and-bound technique always only found a single optimal solution, as the problem reduced to a non-integer problem very well. This reduction is a crucial part of the ILP optimisation process and skipping it would make all but the most trivial problems intractable. So we use an alternative method to find multiple optimal solutions.

Our alternative approach is to find sets of nodes that may be inverted, where each node is a variable group. All the variable groups in a set are connected to at least one other node in the set (equation 1. below) and every node that is connected to a node in the set belongs to the set (equation 2. below).

$$\forall g_i \in S \subseteq G : \exists g_j \in S \text{ s.t. } (g_i, g_j) \in E \quad (1)$$

$$\forall g_i \notin S, \forall g_j \in S : \nexists (g_i, g_j) \in E \quad (2)$$

These sets have now been defined such that if we take an optimal solution to the ILP program then the memory assignment of every node within a set may be flipped simultaneously to give a new solution that will still be optimal in terms of the ILP model. The exception to this is that any set that contains a node which is fixed to a specific memory bank (e.g. the *automatic node* belongs to the set) then that group is not invertible. If we take $|S|$ to be the number of sets in G that may be inverted then there are at least $2^{|S|}$ different possible optimal memory assignments for G.

As an example of this consider the interference graph in figure 4(b), this would be split into two sets: $\{A\}$ and $\{B, C, D, E\}$. If we assume all the edges have equal weight

then figure 4(c) contains one possible optimal set of memory assignments. Some potential parallelism between accesses to D and E has been blocked due to them being assigned to the same memory bank, but as this graph is not two-colourable this is inevitable. There are 3 additional optimal assignments that can be found by inverting groups, assuming that none of the nodes are fixed to a specific memory. The first additional assignment can be found by flipping the assignment of $\{A\}$, the second by flipping the assignments of $\{B, C, D, E\}$ and the third by flipping both. Figure 4(d) shows the assignment after flipping both sets.

This simple example also allows us to see that the set of optimal solutions found by inverting sets is not necessarily the full set of optimal solutions. In figure 4(b) there are 6 optimal ways of colouring $\{B, D, E\}$, C will always be the opposite of B meaning there are 6 optimal ways of colouring $\{B, C, D, E\}$. Combine this with the 2 ways of colouring $\{A\}$ and you have 12 different optimal solutions instead of the $2^2 = 4$ found by inverting sets. However, as our aim is not to find every possible optimal solution to the ILP program but only a representative set for evaluation this approximation is sufficient. In fact, we do not even take all $2^{|S|}$ colourings as there would be too many possibilities for many benchmarks, instead we just take $|S| + 1$. Specifically the colouring found by inverting all sets and then the colourings found by inverting each set individually.

6. Soft Colouring

As the previously described ILP assignment solution finds an optimal solution to an NP-hard problem it has exponential run-time. For small and simple problems the ILP solver is generally able to reduce most of the integer problem to a linear problem, this part can then by solved in polynomial time. However, for larger and more complex problems (or interference graphs) the reduction is less effective. This means that small changes to the interference graph can change its reducibility, resulting in a large increase in the time it takes to solve the model. Both the exponential runtime and the unpredictability of the solving time make the ILP assignment solution undesirable in many cases. A solution which finds good colourings quickly and with more predictable solving time would seem advantageous.

6.1. Single Solution

Graph colouring is well established within compilers, primarily for register allocation. However, graph colouring in terms of memory bank assignment is slightly different from graph colouring for register allocation. In register allocation the colouring is done under the 'hard' constraint that two interfering registers must not be placed in the same register. For memory bank assignment we operate under the 'soft' constraint that we would prefer two interfering variables to not be placed in the same memory bank.

Therefore conventional graph colouring approaches are unlikely to be adequate. Instead we make use of an algorithm designed for a distributed environment where colouring problems frequently operate under soft constraints. We serialise a distributed stochastic soft-colourer [5].

Algorithm 2 Soft Colouring(Variable Groups G)

Require: An interference graph
Ensure: Locally-optimal memory assignment

1: **for all** $g_i \in G$ **do**
2: $C_i \leftarrow \text{rand}(\{0, 1\})$
3: **while** G is still not a local optimum **do**
4: Determine C_i^{opt}
5: Inform central controller whether $C_i = C_i^{opt}$
6: With probability P: $C_i \leftarrow C_i^{opt}$
7: **end while**
8: **end for**

Here, C_i is the current colouring of g_i and C_i^{opt} is the current locally optimal colouring of g_i. The results of step 5 for all nodes allows the central controller to make a decision for step 3. If every node is already at an optimal colour then the algorithm terminates, as the colouring will no longer change. If any node still isn't an optimal colour then it may change, which would then cause other nodes to change colour in the next iteration, so the loop continues to execute. It is also worth noting that in step 6 C_i may already be equal to C_i^{opt}.

Step 4 can be calculated using the equation below, where X_i, Y_i and W_{ij} are defined as for ILP in section 5.1 and g_i refers to the current node as in the above algorithm.

$$\begin{aligned} \forall e = (g_i, g_j) \in E : \\ costX &= \sum X_j \cdot W_{ij} \\ costY &= \sum Y_j \cdot W_{ij} \\ cost &= \min(costX, costY) \end{aligned}$$

Essentially this calculates the weighted value of how many of the neighbours of g_i are on memory banks *X* and *Y* and then picks the colour with the lowest value, i.e. the one with the fewest conflicts. Although this method of minimising conflicts is different from the ILP optimisation metric, where we try to maximise the potential parallelism, they are actually equivalent.

6.2. Changes To Interference Graph

In addition to using a different algorithm from ILP we also make some changes to the interference graph $I =$

(G, E) for soft colouring. The set of vertices G stays the same, but some additions are made to the set of weighted interferences E. Specifically we make the graph fully weakly connected by connecting every unconnected pairs of nodes in G with an edge with a very low weight. This weight is set low enough that it will always be lower than any weight relating to an actual detected interference.

This low weight means that these extra nodes never change a colouring decision between two interfering nodes. What it does do is provide a balancing metric for all unconnected nodes (such as $\{A\}$ in figure 4(b)) so that they will be roughly equally distributed between the *X* and *Y* memories. If the *automatic node* is being considered then there will also be a slight bias to placing the nodes in memory *Y*.

6.3. Multiple Solutions

As there are stochastic elements in the soft colouring algorithm it is possible to get a range of colourings by repeated execution of the technique. Every set of assignments returned will be some local optimum.

7. Experimental Evaluation

7.1. Platform and Benchmarks

We implemented our source-level C to DSP-C compiler using the SUIF compiler framework [18]. The C program is converted into the SUIF intermediate format which is then annotated with aliasing information using the SPAN tool [14]. We use this information to form groups of variables as described in section 4.1 and output DSP-C with group identifiers in place of memory qualifiers. The C preprocessor may be used to assign a group of variables to a specific memory bank according to the generated group to memory bank mapping.

Both the soft colourer and the ILP colourer are implemented in Java. The ILP colourer makes use of the *lp_solve* [1] library, which is implemented as a native binary, with the default pre-solve and optimisation settings.

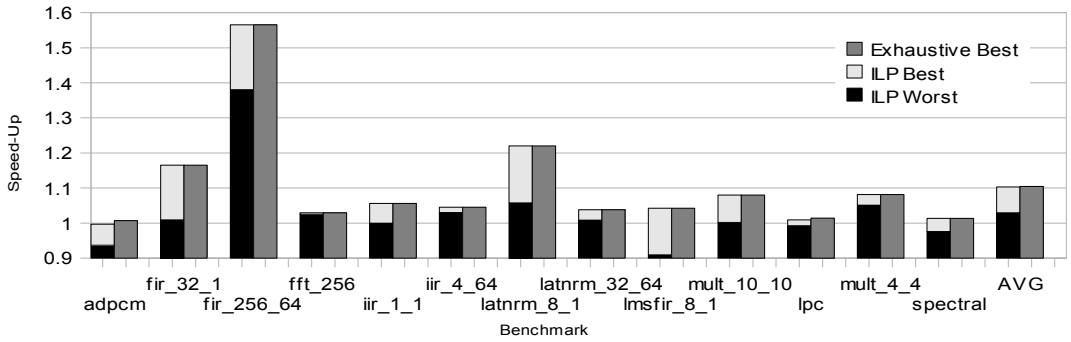
The colourings were done on a Linux system with two dual-core 3.0GHz Intel Xeon processors and 4GB of memory. The experiments where run on an Analog Devices TigerSHARC TS-101 DSP operating with a clock of 300MHz, the DSP-C programs were compiled using the Analog Devices VisualDSP++ compiler. We evaluated our technique using the UTDSP benchmark suite [11].

7.2. Results

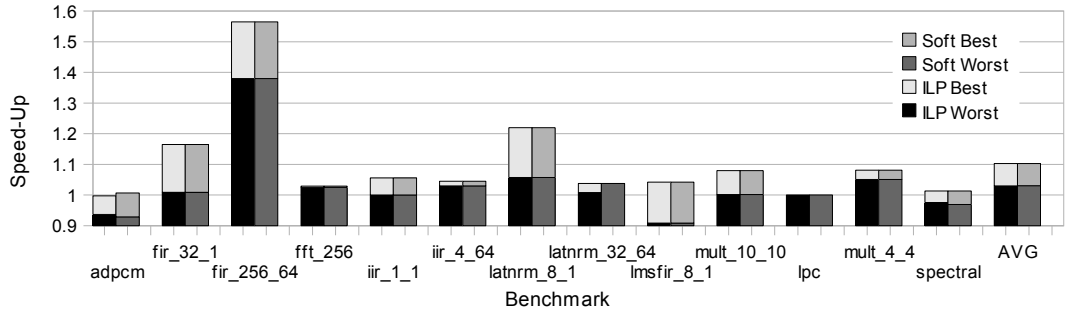
Initially we evaluated the effectiveness of the colourings provided by ILP against exhaustive results. These exhaustive results were obtained by running every possible colouring for each benchmark (except for *lpc*, where only 2833 different colours were tried, representing 8.6% of the total colourings). Figure 5(a) shows the speed-up achieved by the various different equivalent ILP colourings described in section 5.2. The 'best' and 'worst' bars in the figure correspond to the highest and lowest speed-ups, relative to the performance of ISO C, in the set of equivalent ILP solutions found per benchmark. It can be seen that most of the benchmarks see a significant range of results, and *lmsfir_8_1*, *lpc* and *spectral* see performance improvements for some ILP colourings but degradations for others. Another observation is that for all benchmarks except *adpcm* the ILP colourer was able to find the optimal solution. On average the range of speed-ups due to ILP was between 1.030 and 1.103, compared to 1.104 for the best of the exhaustive results.

After this we compared the soft colouring technique against the ILP colourer. The results of this are shown in figure 5(b), where the ranges of both the ILP and the soft colouring results are shown. Here we see that despite not being guaranteed to solve the colouring model optimally soft colouring does just as well as the ILP colourer, finding almost exactly the same range of results. However, the range is still quite wide so we attempt to constrain the assignments by adding the additional *automatic node* described in section 4.2. The effects of this are shown in figure 5(c). The *automatic node* does shorten the range of solutions for both ILP and soft colouring, but not in the same way. For ILP colouring mostly good results are eliminated (going from 1.030-1.103 to 1.032-1.096 on average), for soft colouring mostly bad results are eliminated (going from 1.031-1.103 to 1.039-1.101 on average). Also, the *automatic node* allows soft colouring to always find the truly optimal solution for *fir_256_64* and to find better solutions than the ILP colourer for *lpc* and *spectral*.

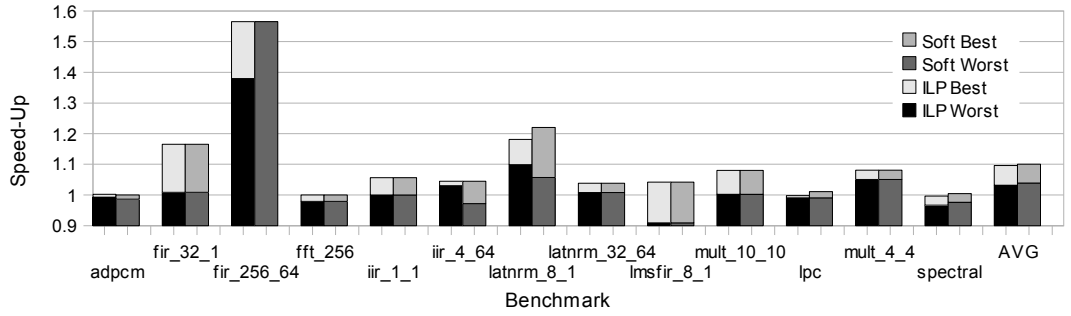
The final thing to consider is how long it takes to execute the two colouring algorithms. Figure 5(d) shows the time taken by both the ILP and soft colourers for the largest of the UTDSP benchmarks. The remaining benchmarks all had total colouring times of under a second. The time reported is the time taken to perform alias analysis and then to execute the colouring algorithm, the alias analysis takes a notable amount of time for these benchmarks, up to 7 seconds for *adpcm*. The *Glob* timings are the time it takes to perform colouring if as many automatic variables as possible are made global (any declared in non-recursive functions). This is an artificial change that always results in slower code, however it allows results to be obtained for larger interference graphs, without the difficulties of getting larger programs to run on the target architecture. It roughly doubles to quadruples the number of nodes in the interference graph. The larger interference graph due to globalisation exposes the dangers of the ILP solvers optimisation strategies and exponential run-time. Although for the



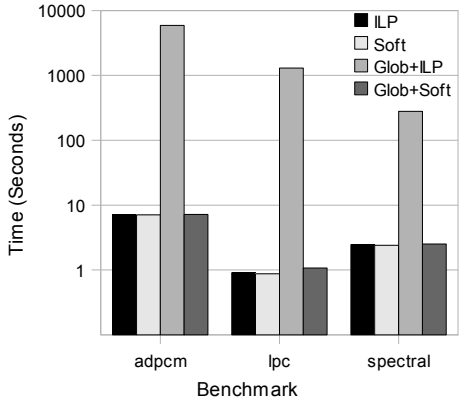
(a) A comparison of the range of ILP solutions against the true optimum.



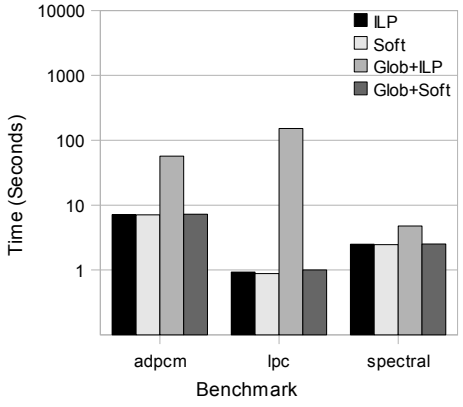
(b) A comparison of the range of ILP solutions against the range of soft colouring solutions.



(c) Figure 5(b) with an additional *automatic node*.



(d) Time taken to perform memory assignment.



(e) Figure 5(d) with an additional *automatic node*.

Figure 5. Various results for ILP and soft colouring.

non-globalised code ILP and soft colouring take roughly the same time, for the globalised code the soft colourer is hundreds of times faster. Most notably for the globalised *adpcm* the soft colourer takes 7.2 seconds, but the ILP colourer takes over one and half hours (5873 seconds). Figure 5(e) shows the affect that adding the additional *automatic node* to the interference graph has on the time it takes to do the colouring. There is little change for soft colouring, but it has a dramatic effect on the ILP colourer for globalised code, *adpcm* is now coloured in under a minute (57 seconds). This possibly means that this single extra node allowed a larger part of the integer problem to be reduced to a linear problem, demonstrating the fragility of ILP colouring.

8. Summary, Future Work and Conclusions

We have presented a method for performing dual memory bank assignment at the source-level, using a C to DSP-C compiler. We have demonstrated an assignment technique that performs as well as ILP colouring but returns a shorter range of results and has a lower and more predictable execution time. We evaluated our technique on the *UTDSP* benchmark suite where we achieved up to a 1.103 speed-up on average.

The colouring techniques that have been applied to the problem are mature, and the comparisons against exhaustive results show this. However, the range of equivalent optimal results that these techniques find and the effect that adding the *automatic node* to the interference graph can have – both on the range of solutions found and the time it takes to find them – suggests that further developing the interference model is likely to provide further results. We intend to investigate applying machine learning in this area.

Our technique may be easily introduced to an existing DSP tool-chain due to operating at the source-level. The shorter range of returned results and the more predictable colouring times compared to ILP makes this technique less risky to include than previous methods. Also, the results are demonstrated to be very close to the true optimum meaning it should be able to do as well as hand-colouring, but automatically. Combined, these make it seem likely that our technique would be effective in an industrial scenario.

References

[1] lp_solve package. `http://lpsolve.sourceforge.net/5.5/`, 2007.

[2] ACE. DSP-C, an extension to ISO/IEC IS 9899:1990. Technical report, ACE Associated Compiler Experts bv, 1998.

[3] M. Beemster, H. van Someren, W. Wakker, and W. Banks. The Embedded C extension to C. `http://www.ddj.com/cpp/184401988`, 2005.

[4] S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9), 2000.

[5] S. Fitzpatrick and L. Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In *Proceedings of the International Symposium on Stochastic Algorithms*, pages 49–64, December 2001.

[6] A. Frederiksen, R. Christiansen, J. Bier, and P. Koch. An evaluation of compiler-processor interaction for DSP applications. In *Proceedings of the 34th IEEE Asilomar Conference on Signals, Systems, and Computers*, 2000.

[7] G. Gréwal, S. Coros, A. Morton, and D. Banerji. A multi-objective integer linear program for memory assignment in the DSP domain. In *Proceedings of the IEEE Workshop on Memory Performance Issues*, pages 21–28, February 2006.

[8] G. Gréwal, T. Wilson, and A. Morton. An EGA approach to the compile-time assignment of data to multiple memories in digital-signal processors. *SIGARCH Computer Architecture News*, 31(1):49–59, March 2003.

[9] JTC1/SC22/WG14. Programming languages - C - extensions to support embedded processors. Technical report, ISO/IEC, 2004.

[10] M.-Y. Ko and S. S. Bhattacharyya. Data partioning for DSP software synthesis. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 344–358, September 2003.

[11] C. G. Lee. UTDSP benchmark suite, 1998.

[12] R. Leupers. Novel code optimization techniques for DSPs. In *Proceedings of the 2nd European DSP Education and Research Conference*, 1998.

[13] R. Leupers and D. Kotte. Variable partioning for dual memory bank DSPs. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages 1121–1124, May 2001.

[14] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemenation*, pages 77–90, May 1999.

[15] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 234–243, September 1996.

[16] V. Sipkovà. Efficient variable allocation to dual memory banks of DSPs. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, pages 359–372, September 2003.

[17] A. Timmer, M. Strik, J. van Meerberger, and J. Jess. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. In *Proceedings of the Design Automation Conference (DAC)*, 1995.

[18] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, December 1994.