Efficient Variable Allocation to Dual Memory Banks of DSPs

Viera Sipkova

CD-Lab Compilation Techniques for Embedded Processors
Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Vienna, Austria
Tel.: (+43-1)-58801-58520
sipka@complang.tuwien.ac.at

Abstract. To improve the overall performance, many of the modern advanced digital signal processors (DSPs) are equipped with on-chip multiple data memory banks which can be accessed in parallel in one instruction. In order to effectively exploit this architectural feature, the compiler must partition program variables between the memory banks appropriately – two parallel memory accesses always must take place on different memory banks. There is some research work that addresses this issue, however, most of this has been proposed as a post-pass (machine dependent) optimization. We attempt to resolve this problem by applying an algorithm which operates on the high-level intermediate representation, independent of the target machine. The partitioning scheme is based on the concepts of the interference graph which is constructed utilizing the control flow, data flow, and alias information. Partitioning of the interference graph is modeled as a Max Cut problem. The variable partitioning algorithm has been designed as an optional optimization phase integrated in the C compiler for a digital signal processor. This paper describes our efforts. The experimental results demonstrate that our partitioning algorithm finds a fairly good assignment of variables to memory banks. For small kernels from the DSPstone benchmark suite the performance is improved from 10% to 20%, for FFT filters by about 10%.

1 Introduction

To improve the effective bandwidth and memory access speed, recently, designers of embedded systems prefer the on-chip memory over the use of the external memory or more complicated hardware mechanisms. They have developed special architectural features to access multiple data memories in parallel, provided that referenced variables have been allocated to different memory banks. Furthermore, the instruction set may encode parallel accesses in a single instruction word, which improves the code density and reduces the code size. Examples of processors which support such memory architecture include the Motorola DSP56000, Analog Devices ADSP2106x, NEC μPD77016, etc. In this research we will be using the experimental digital signal processor xDSPcore [1].

Unfortunately, the current compiler technology is generally unable to deliver high-quality code for DSPs whose architectures are extremely irregular. High-level C data types and language constructs are not easily mapped into dedicated DSP machine instructions. The reason is a lack of suitable optimization techniques. Much of the research for optimizing compilers has been done for general-purpose microprocessors and has focused on traditional machine-independent optimizations. Producing a high-performance code for DSPs requires adequate support for each specialized architectural feature.

The goal of this paper is to present the new optimization technique which attempts to maximize the benefit of dual data-memory bank DSPs. In order to make an efficient use of the bandwidth increase offered by dual memory banks (often denoted by X and Y), the C program variables have to be partitioned appropriately between X and Y.

```
int a[100], b[100];
int dot_product(void)
{   int dot = 0;
    for (i = 0; i < 100; i++)
        dot += a[i] * b[i];
    return dot;
}
```

Fig. 1. Dot Product (C code)

Multi-memory bank architectures have been proved to be effective for many operations commonly found in embedded applications. For instance, in a dot product operation shown in Fig. 1 arrays `a` and `b` must be placed in different memory banks for allowing simultaneous access. The corresponding assembly code looks as outlined in Fig. 2. The notation || denotes that the combined operations should be executed in parallel. The instruction (3) performs both *loads*.

To solve the problem of memory assignment several approaches are possible at different stages of compilation flow. Our partitioning technique has been designed as a separate optimization module of the C compiler for the xDSPcore. It operates on the high-level intermediate representation, so it is not dependent on the target-machine. The result of the partitioning is the intermediate representation annotated with the X/Y bank assignment information for all variables. This can be utilized later in the subsequent code generation phase. The main scheme of our approach is similar to that proposed in [2]. It is modeled by a graph which tries to reflect all the potential parallelisms between the variables and also provides a weight metric for different parallel access demands. The partitioning itself is solved as the combinatorial optimization problem *Max Cut*,

```
(1)   movcl g_b,R1     ||  movc  0,D0
(2)   movcl g_a,R0     ||  bkrep 100,LBL
(3)   ld    (R0)+,D2   ||  ld    (R1)+,D1
(4)   nop
(5)   mul   D2,D1,A1
(6)   nop
(7)   add   D0,D2,D0
(8)   LBL:
(9)   ret
```

Fig. 2. Dot Product (assembly language)

which is known as NP-complete. To find a near optimal partitioning we have implemented several partitioning algorithms, exact and also approximating.

The structure of the paper is organized as follows. In Section 2 a brief summary of the previous work is presented. In Section 3 the partitioning strategy is described. Section 4 provides our experimental results, and finally, Section 5 presents conclusions and future plans.

2 Related Work

The earliest work on this problem was presented by `Powell, Lee`, and `Newman` [3]. Here, the assignment of program variables to the X/Y memory banks occurs on the meta-assembly code, after the scheduling and register allocation phase. Variables are assigned to X and Y in an alternating fashion, according to their access sequence in the program code, without any analysis.

In the work of `Saghir, Chow`, and `Lee` [4, 5] a variable partitioning technique for a hypothetical VLIW DSP architecture is presented. They describe two algorithms: compaction-based data partitioning, and partial data duplication. Both are performed as the post-pass phase operating only on basic blocks. The central data structure is an *interference graph*, whose nodes are partitioned into two sets heuristically, by searching for the minimum-cost partitioning.

In the approach of `Sudarsanam` and `Malik` [6, 7] the memory bank allocation and register allocation take place in a single phase, after a pre-compaction step of the input program producing the symbolic assembly code. The algorithm is based on *graph labeling*, the objective of which is to find an optimal labeling of a constraint graph representing conditions on the register and memory bank allocation. The simulated annealing is used to find a good labeling.

In the work of `Leupers` and `Kotte` [2] the variable partitioning is performed as a separate optimization phase after the initial run of the backend used only to determine the exact set of memory accesses. The variable partitioning is modeled as Integer Linear Programming based on the *interference graph*.

The most recent papers concerning the problem of the memory banks assignment are probably [8–10].

`Cho, Paek`, and `Whalley` [8] presented a work where they study the memory and register allocation for non-orthogonal architectures. Memory bank assignment is done after the code compaction phase. For partitioning they use a heuristic that chooses the *maximum spanning tree* of the *simultaneous reference graph*. Then X memory is assigned in even depth and Y memory in odd depth in this tree.

`Zhuang, Pande`, and `Greenland` [9] proposed a post-register allocation solution which attempts to maximally combine loads and stores to generate parallel load/store instructions after code is generated. They introduce the *motion schedule graph*, which is partitioned applying the two-coloring algorithm.

The work of `Zhuge, Xiao`, and `Sha` [10] describes two algorithms: variable partitioning and scheduling with variable re-partition. The idea here is to reveal the true picture of potentially parallel memory accesses that can really occur in scheduling. The problem is modeled by the *variable independence graph* refined by a *mobility window* used by eliminating these edges that are impossible to be scheduled in the same control step. To partition the graph into multiple disjoint sets a greedy strategy is used.

In all previous work some kind of graphs have been used which are partitioned applying different optimization methods. However, all (except of [2]) have been proposed as a post-pass backend phase operating on the assembly code. This has a benefit that all memory accesses can be captured, however, generally, it can not be performed separately without any impact on the register allocation and scheduling.

In our approach the algorithm operates on high level intermediate representation. To find any potential parallelism between memory accesses information from all the sophisticated program analysis are possible to be utilized. Our framework is global (intra-procedural) and is not just limited to basic blocks. Memory accesses of the entire program are handled and relations between them are analyzed at once, so no contrary demands on assigning a certain variable to either X or Y can arise. Surely, it is not always possible to recognize all memory accesses, however, as will be reported later in this paper, our performance results are quite encouraging.

3 Partitioning Scheme

The C compiler which our variable partitioner has been integrated into, accepts a C-source code that is translated through the frontend into the tree-like high-level intermediate representation (HIR). The root of the HIR is the *unit* which contains a list of *functions, global variables, externals* and *types*. Every function contains a *list of function parameters, local variables*, and *basic blocks* consisting of a *sequence of statements*. The HIR is optimized applying the standard machine-independent transformation. Furthermore, the frontend provides also

```
  IrBlock bb1
      IrAssign
          IrAddress (IrLocal tmp_b)
          IrConvert
*             IrAddress (IrGlobal b)
      IrAssign
          IrAddress (IrLocal tmp_dot)
          IrConstant 0
      IrAssign
          IrAddress (IrLocal tmp_a)
          IrConvert
*             IrAddress (IrGlobal a)
      IrLoopStart
          IrConstant 100
          IrAddress (IrBlock bb2)
  IrBlock bb2
      IrAssign
          IrAddress (IrLocal tmp_dot)
          IrAdd
              IrRead
                  IrAddress (IrLocal tmp_dot)
              IrMult
                  IrRead
                      IrRead
*                         IrAddress (IrLocal tmp_a)
                  IrRead
                      IrRead
*                         IrAddress (IrLocal tmp_b)
      IrAssign
          IrAddress (IrLocal tmp_b)
          IrAdd
              IrRead
                  IrAddress (IrLocal tmp_b)
              IrConstant 1
      IrAssign
          IrAddress (IrLocal tmp_a)
          IrAdd
              IrRead
                  IrAddress (IrLocal tmp_a)
              IrConstant 1
      IrLoopEnd
          IrAddress (IrBlock bb2)
          IrAddress (IrBlock bb3)
  IrBlock bb3
      IrReturnValue
          IrRead
              IrAddress (IrLocal tmp_dot)
          returnReg
```

Fig. 3. Dot Product (HIR code)

some abstract structures of the program, such as *call graph, control flow graph, dominator tree, SSA-form*, which are bases for the advanced analysis framework. The HIR is taken as input for the partitioner which may be invoked at any point after the compiler frontend and before the backend. For illustration, the HIR of the dot product code introduced in Fig. 1 is outlined in Fig. 3.

In our approach we focus on the set of global variables and static local variables. Local variables and parameters of a function are processed later in the code generation phase. They are allocated either in registers, or in the stack which is part of one particular memory bank. These temporaries are handled by the scheduler so that the memory conflicts are avoided. Array variables are treated as monolithic entities that are allocated to a single memory bank. To determine the optimal memory bank assignment for given variables, references over all functions in the program need to be observed at the same time.

The partitioning algorithm is based on the concepts of the *interference graph*, where each memory access is represented by one vertex. An edge between two vertices indicates that they may be accessed in parallel, and that the corresponding variables should be stored in separate memory banks. The goal is to partition the interference graph in such a way that the potential parallelism is maximized. The partitioning process consists of two separate components: the first constructs the interference graph, the second partitions the interference graph.

3.1 Construction of the Interference Graph

Definition 3.1 *The interference graph is defined as an edge-weighted undirected graph $G = (V, E)$, where each vertex $v \in V$ represents a memory access, and an edge $e = (v, u) \in E$ connecting a pair of vertices v and u, indicates that there is no dependence between them. With each edge $e = (v, u) \in E$ a nonnegative weight $W(e)$ is associated which represents the extent of independence between v and u.*

The interference graph is constructed for the whole program. The set of vertices is generated by traversing the HIR of the program (all functions, basic blocks and statements) and looking for objects `IrAddress` which point to global variables (see Fig. 3). Local variables (`tmp_a, tmp_b`, and `tmp_dot`) will be allocated in registers. For each memory access found one interference vertex is created. The `IrAddress` can represent one or more memory accesses, dependent on how many `IrRead` operators are preceding to it. `IrRead` denotes the read of the value at the address which is specified by the following address expression. Multiple consecutively `IrRead` operators substitute the multilevel indirect addressing, and to determine all global variables associated, the *alias analysis* is required. Currently, we utilize only information from the SSA (static single assignment) form, so not all memory accesses can be caught. The percentage of not-resolved variable references is strongly dependent on the structure of the source program. In our example there were recognized two memory accesses to `a` – (12), (25), and two memory accesses to `b` – (5), (28). Accesses (25) and (28) were identified through the double `IrRead` operator.

The interference vertex, besides the memory address itself, encapsulates also all information about its enclosing context (owner statement, owner block, def/use attribute, etc.), which serves as a framework for determining graph edges.

Generating the set of edges E on the set of vertices V is equivalent to the identifying all pairs of memory accesses that can be combined together for parallel execution. To accomplish this problem, at first, we construct the intra-procedural control dependence and data dependence graphs which define the relationship between the basic blocks and also between the statements within each function. There will be an edge $e = (v, u)$ between vertices $v, u \in V$ if and only if the statements (or expressions) enclosing the v and u, respectively, are not control-dependent and also not data-dependent. We suppose that memory accesses occurring in different functions or in different basic blocks can not be scheduled for parallel processing, so, no edge is generated between them.

According to the context in which the memory accesses are included a weight W is assigned to each edge $e = (v, u) \in E$ which is defined :

$$W(e) = EF \times DW(e)$$

where EF represents the *execution frequency* of the enclosing basic block, and DW represents the *distance weight* of the edge.

$$DW(e) = \begin{cases} 2 & \text{if } v \text{ and } u \text{ are contained in expressions of the same statement} \\ 1 & \text{if } v \text{ and } u \text{ are contained in different statements} \end{cases}$$

We chose this simple weight as a heuristic measure, it can be seen as the rate of the probability that the connected vertices will be scheduled into the same instruction.

Once the interference graph has been constructed, each vertex subset $\{v_1, \ldots, v_k\} \subseteq V$ representing accesses to the same variable, is merged into a single vertex v, and all edges containing $v_1, ..., v_k$ are redirected to the new vertex v. The weight of an edge $e = (v, u)$ is modified to

$$W(e) = Max(W(e_i)) \times k$$

where $e_i = (v_i, u)$, for $i = 1, \ldots, k$. So, the size of the graph (number of vertices) is equal to the number of global variables accessed.

3.2 Partitioning of the Interference Graph

The best partitioning of the interference graph $G = (V, E)$ is achieved if the set of vertices V can be divided into two disjoint sets $S \subseteq V$ and $\bar{S} = V - S$, such that the sum of the weights of all edges that connect a vertex $v \in S$ to a vertex $u \in \bar{S}$ is maximal. Variables corresponding to vertices from S are assigned to X memory bank, and variables corresponding to vertices from $\bar{S}$ are assigned to Y memory bank.

Theoretically, in this case the highest number of parallel memory accesses can be obtained. Practically, however, the performance gain is affected by the fact, how the scheduler actually realizes the calculated parallelism.

This partitioning task can be formulated as the combinatorial optimization problem *Max Cut*. The cut $Cut(S, \bar{S})$ is defined as the set of edges that have one endpoint in S and the other endpoint in $\bar{S}$. The Max Cut consists in finding a subset of vertices S such that the weight of $Cut(S, \bar{S})$ given by

$$\sum_{e \in Cut(S,\bar{S})} W(e)$$

is maximized.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of vertices of $G = (V, E)$; we use i for an vertex v_i, and w_{ij} for the weight of an edge $(v_i, v_j) \in E$ (for $e = (v_i, v_j) \notin E$ we set $w_{ij} = 0$). When introducing *cut vectors* $x \in \{-1, 1\}^n$ with $x_i = 1$ for $v_i \in S$, and $x_i = -1$ for $v_i \in \bar{S}$, then the algebraic formulation for Max Cut can be written as follows:

$$\begin{array}{ll} \text{maximize} & \displaystyle\frac{1}{2} \sum_{1 \leq i < j \leq n} w_{ij}(1 - x_i x_j) \\ \text{subject to} & x_i \in \{-1, 1\},\ i = 1, \ldots, n\ . \end{array} \tag{1}$$

The key property of the formulation (1) is that $(1 - x_i x_j)/2$ can take only two values - either 0 or 1, which allows to model the appearance of an edge in a cut within the objective function. For any feasible solution $x = (x_1, \ldots, x_n)$, the set $S = \{v_i \in V : x_i = 1\}$ defines the cut $Cut(S, \bar{S})$ which has the weight equal to the objective value at x.

The first feasible solution of this NP-complete problem was proposed in 1976 by Sahni and Gonzales [11], they presented an approximation algorithm with the performance guarantee 0.5$\times$ optimal value. Since then for a nearly twenty years no significant progress has been made in improving this performance guarantee. Only in 1994 Goemans and Williamson [12, 13] proposed a randomized algorithm based on the semidefinite programming which always delivers a solution of value at least 0.87856$\times$ the optimal value. There exists several extensions of the Goemans and Williamson technique. For example, Frieze and Jerrum [14] designed an algorithm for the *Max k-Cut*, where $k \geq 2$, which can be applicable to an arbitrary number of memory banks.

3.3 Implementation of Partitioning

Provided that the number of vertices is small (less than twenty), the Max Cut is possible to be solved exactly still in reasonable time. Otherwise, approximating techniques are applied. To find a near optimal partitioning we have implemented several approximating algorithms, simple and also more sophisticated. This which yields the best solution is chosen for the partitioning. Algorithms are described in the following.

Exact algorithm

This algorithm computes the Max Cut exactly. It generates recursive all possible cut vectors and calculates its cut. The cut vector having the maximal cut value is chosen as the solution. It can happen that there exists more than one solution – several different cut vectors with the equal maximal cut value. In this case to select the best one must be experimentally examined.

Greedy algorithm

This approximating algorithm represents the iterative approach which utilizes the property of the Max Cut problem that the value of any local optimum is not too far from the value of the total optima. Implementation is based on the scheme described in [15]. The algorithm begins with a naive initial approximation to the solution – all vertices of G are placed into the set S, with the set $\bar{S}$ being empty. Then the method repeatedly iterates over all vertices in order to find a vertex whose relocation to other set could increase the cut. The algorithm is running until it reaches a fix point where each pass produces no further increase of the cut. Algorithm runs in the polynomial time $O(n \times m)$, where n is the number of vertices, and m is the number of edges. It delivers a solution of value at least $0.5\times$ the optimal value.

Semidefinite Programming Relaxation

This approximating algorithm was provided by Goemans and Williamson [12, 13]. It is a simple and elegant technique that randomly rounds the solution to a nonlinear semidefinite programming relaxation. The algorithm always delivers a solution of value at least $0.87856\times$ the optimal value.

Let $\mathcal{R}^n$ denote the space of real n-dimensional column vectors. The *unit scalars* x_i of (1) can be viewed as vectors of *unit norm* belonging to $\mathcal{R}^n$; or more precisely, to the n-dimensional unit sphere $\mathcal{S}_n = \{y \in \mathcal{R}^n : \ \| y \| = y^T y = 1\}$. Associating scalars x_i with the unit vectors $y_i \in \mathcal{S}_n$, for $i = 1, \ldots, n$, the products $x_i x_j \in \{-1, 1\}$ may be relaxed to $y_i^T y_j \in \langle -1, 1 \rangle$.

Then after some mathematical manipulations, (1) can be formulated as a relaxation to a semidefinite program (for more details see [12, 13]):

$$\begin{array}{ll} \text{maximize} & C \bullet Y \\ \text{subject to} & diag(Y) = e \\ & Y \succeq 0 \end{array} \tag{2}$$

Given a feasible solution Y of (2), the set of unit vectors $y_j, j = 1, \ldots, n$, can be obtained by the Cholesky factorization $Y = Z^T Z$, where columns of the matrix Z correspond exactly to the vectors $y_1, \ldots, y_n$.

Using the geometric interpretation, a solution $(y_1, \ldots, y_n)$ consists of n points on the surface of the unit sphere $\mathcal{S}_n$, each representing a vertex of the graph, and the product $y_i^T y_j$ is the cosine of the angle enclosed by these vectors.

Goemans and Williamson proposed the following randomized algorithm for generating cuts : construct a random hyperplane through the origin of $\mathcal{S}_n$ and group all vectors on the same side of this hyperplane together. The hyperplane can be constructed by choosing a random vector r uniformly distributed on the unit sphere $\mathcal{S}_n : \mathcal{H}(r) = \{y \in \mathcal{R}^n : r^T y = 0\}$. Partitioning of the vertex set V into $(S, \bar{S})$ is formed by assigning all vertices $v_i \in V$ to S whose corresponding vectors y_i have positive inner product with r:

$$\begin{array}{l} S = \{v_i \in V \;\; : \;\; y_i^T r \geq 0\} \\ \bar{S} = \{v_i \in V \;\; : \;\; y_i^T r < 0\} \end{array}$$

This semidefinite relaxation has been implemented using the SDPA solver developed by Fujisawa, Kojima, Nakata and Yamashita [16]. For the Cholesky factorization and randomizing the solution the LAPACK-library is utilized.

Semidefinite Rank-2 Relaxation

For experimental reasons we have implemented also this algorithm which was developed by Burer, Monteiro and Zhang [17]. It represents the specialized version of the Goemans-Williamson randomized technique with the same performance guarantee. Algorithm was implemented utilizing the Fortran 90 software package CIRCUT [18] which was rewritten into C++ object.

4 Experimental Results

Our partitioning technique was empirically evaluated on the simulator of the experimental digital signal processor xDSPcore [1]. We did experiments with various small kernels from the DSPstone benchmark suite [19], and some applications. The metrics which the performance is measured in is the number of cycles executed, and the number of memory conflicts appeared. A memory conflict occurs if two accesses to the same memory bank are scheduled in one instruction; in this case an extra (stalling) cycle is generated by a special hardware mechanism.

In order to demonstrate the effectiveness of our partitioning algorithm, for each kernel several variants were compiled, executed, and evaluated. In the first version variables are assigned explicitly only to one memory bank. In the second version variables are not assigned before linking phase; here an optimistic algorithm by scheduling is applied and the linker tries to resolve the variable allocation. For these two cases the partitioner was disabled. In the third version variables are assigned to memory banks by means of the partitioner. These three cases are referred to as *X-Allocating*, *Scheduling*, and *Partitioning*, respectively.

Table 1. DSPstone Kernels

Kernel	X-Allocating				Scheduling				Partitioning			
	Cycl.	X	Y	Confl.	Cycl.	X	Y	Confl.	Cycl.	X	Y	Confl.
dot_product	625	200	0	0	525	200	0	100	525	100	100	0
convolution	625	200	0	0	525	134	66	34	525	100	100	0
matrix_mult_1	5368	2100	0	1000	5368	2100	0	1000	5368	1000	1100	0
matrix_mult_2	5014	2010	0	900	4993	2010	0	900	4993	1100	910	0
mat1x3	85	24	0	0	76	24	0	9	76	9	15	0
lms	219	95	0	0	219	12	83	16	188	48	47	0
fir2dim	963	304	0	144	963	304	0	144	963	144	160	0
biquad_n_sections	71	38	0	12	84	38	0	16	66	21	17	0
	Total Number of Cycles											
dot_product	625				625				525 (84.0%)			
convolution	625				559				525 (84.0%)			
matrix_mult_1	6368				6368				5368 (84.3%)			
matrix_mult_2	5914				5893				4993 (84.4%)			
mat1x3	85				85				76 (89.4%)			
lms	219				235				188 (85.8%)			
fir2dim	1107				1107				963 (87.0%)			
biquad_n_sections	83				100				66 (79.5%)			

Table 1 lists the performance results obtained for some selected DSPstone kernels. Each kernel contains some loops with operations on two or three global arrays. According to the information about the memory assignment the code generator schedules the operations into instructions, so, for our three examined cases the target code may look differently. For each variant the first column shows the total number of cycles executed (memory conflicts are not included), the second and third columns show the number of accesses to X and Y memory bank, and the fourth column shows the number of memory conflicts. We can see that in the first version, where all variables are allocated to X memory bank, the number of memory conflicts is equal to zero only in this case when memory accesses are scheduled in separate instructions, that is, the number of executed cycles is increased. In the optimistic version variables are tried to be allocated to both memory banks, however, the results are not better than in the first case. It is evident that the best performance gain is achieved by the version with partitioning. For these small kernels, all implemented algorithms, exact and also approximating, yield the identical partitioning result which seems to be quite ideal. The pure memory access cycles are decreased by about 50%, and the improvement of total number of cycles ranges from 10% to 20%. In real applications, however, this would not be true.

Table 2. FFT Filters

Kernel	**X-Allocating**				**Scheduling**				**Alternate Alloc.**			
	Cycl.	X	Y	Confl.	Cycl.	X	Y	Confl.	Cycl.	X	Y	Confl.
fft256_1	194681	110252	0	17052	204178	113248	0	20195	195560	60888	49364	12831
fft256_2	162341	91046	0	11053	168927	91291	0	17661	146322	48479	39920	8823

	Partitioning							
	Execution Frequency				No Frequency			
	Cycl.	X	Y	Confl.	Cycl.	X	Y	Confl.
fft256_1	194261	24110	86142	12572 (74%)	194261	24110	86142	12572 (74%)
fft256_2	145977	44909	42518	8785 (79%)	152171	29957	61936	7427 (67%)

	Total Number of Cycles			
	X-Allocating	Scheduling	Alternate Alloc.	Partitioning
fft256_1	211733	224373	208391	206833 (97%)
fft256_2	173394	186588	155145	154762 (89%)

Table 2 presents performance results of code which contains a fixed-point implementation of 256-point complex Fast Fourier Transform (FFT) and the inverse FFT. It is based on Radix-2 decimation in frequency domain algorithm on a block of complex numbers. Two versions of the FFT code have been examined. In `fft256_1` the real and imaginary values of the complex data are stored in one array in interleaved format (real followed by imaginary). The `fft256_2` represents a slightly modified code; in order to avoid the successive memory accesses to the same array, the real and imaginary values of the complex data are stored in two separate arrays. In both versions all global arrays are referenced through the subscripts, not through the pointers, so, all accesses could be found and resolved without any complicated alias analysis.

Additionally to the *X-Allocating, Scheduling*, and *Partitioning* strategies we measured also the approach where the vertices of the interference graph are partitioned in the alternate way starting with X-memory, it is referred to as *Alternate Alloc.* By partitioning we experimented with several heuristics. In Table 2 results from two instances are reported : in the first the edges are weighted by the execution frequency of basic blocks as defined in Section 3.1; while in the second, the edges are weighted without using any frequency estimates (EF is supposed to have the value one).

For the `fft256_1` code the size of the interference graph is equal to 10, and surprisingly, the partitioning algorithm yields only one solution regardless of the execution frequency is used or not. Wenn comparing the *X-Allocating* with the *Partitioning* the number of memory conflicts is decreased by 27%, however, the total number of cycles is approximately the same.

For the `fft256_2` code the size of the interference graph is equal to 13. In this case better results can be achieved because the butterfly FFT-algorithm operates now on two arrays (real and imaginary) instead of on one array. The partitioning algorithm without the execution frequency used yields three solutions which give the equal results. The algorithm using the execution frequency yields twelve solutions giving several different results, the best one is introduced in the table. Also for this version the execution frequency does not improve significantly the quality of the results. Wenn comparing the *X-Allocating* and *Partitioning* strategies, the number of memory conflicts is decreased by 33%, and the total number of cycles by about 10%.

The *Alternate Allocating* approach for both codes shows the comparable results as the *Partitioning* strategy. This is due to the character of the FFT-algorithm.

It is worth to say, that for each observed benchmark approximating algorithms give the identical solution as the exact algorithm. So, which algorithm is preferred has not a great impact on the partitioning result. To obtain a real performance improvement, the most significant is to provide the correct information for partitioning. A good graph model should reflect all the potentially parallel memory accesses that may actually occur in scheduling.

5 Conclusion

In this paper we have presented an algorithm which attempts to maximize the benefit of dual data memory banks. The algorithm is based on partitioning the interference graph whose nodes represent variables and edges represent potential parallel accesses to pairs of variables. The interference graph is constructed utilizing the control flow, data flow, and alias information. For partitioning itself, formulated as *Max Cut* problem, we have implemented several methods. All of them work very well and fast. The important contribution of our approach is that the algorithm operates on the high-level intermediate representation, independent of the target machine. Our framework is global and is not just limited to basic blocks. Both scalar and array variables of the entire program are handled at once, so no contrary demands on assigning a certain variable to either X or Y can arise.

The experimental results demonstrate that our method finds a quite satisfying memory assignment. On small kernels we were able to reduce the number of memory cycles by 50%, and the total number of cycles by 10%–20%. For FFT filters the number of memory conflicts is decreased by 30%, and the total number of cycles by 10%.

In the future we plan to work on the refinement of the interference graph. We would like to make experiments with several new heuristics including runtime profiling information, and evaluate the method on real bigger applications. We also plan to explore the memory partitioning for DSP architectures which are equipped with interleaved memory banks where the interleaving factor can be any number, not only two.

6 Acknowledgments

I would like to acknowledge the *Christian Doppler Forschungsgesellschaft* and *Infineon* for funding this research. I would also like to thank *Andreas Krall* for his valuable comments on this paper and *Ulrich Hirnschrott* for his help by compiling and simulating the kernels.

References

1. C. Panis, G. Laure, W. Lazian, A. Krall, H. Grünbacher, J. Nurmi: *DSPxPlore – Design Space Exploration for a Configurable DSP Core.* In: Proceedings of the GSPx, Dallas, Texas, USA (2003)
2. R. Leupers and D. Kotte: *Variable Partitioning for Dual Memory Bank DSPs.* In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ASSP). Volume 2. (2001) 1121–1124
3. D.B. Powell, E.A. Lee, and W.C. Newman: *Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams.* In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ASSP). Volume 5. (1992) 553–556
4. M.A.R. Saghir, P. Chow, and C.G. Lee: *Automatic Data Partitioning for HLL DSP Compilers.* In: Proceedings of the 6th International Conference on Signal Processing Applications and Technology. (1995) I–866–871
5. M.A.R. Saghir, P. Chow, and C. G. Lee: *Exploiting Dual Data-Memory Banks in Digital Signal Processor.* In: ACM SIGOPS Operating Systems Review, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. Volume 30(5). (1996) 234–243
6. A. Sudarsanam and S. Malik: *Memory Bank and Register Allocation in Software Synthesis for ASIPs.* In: Proceedings of the IEEE/ACM International Conference on Computer Aided Design. (1995) 388–392
7. A. Sudarsanam and S. Malik: *Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs.* Journal of the ACM Transactions on Automation of Electronic Systems (TODAES) **5** (2000) 242–264
8. J. Cho, Y. Paek, and D. Whalley: *Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithm.* In: Proceedings of the International Conference on the LCTES and SCOPES, Berlin, Germany (2002)

9. X. Zhuang, S. Pande, and J.S. Greenland: *A Framework for Parallelizing Load/Stores on Embedded Processors.* In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Virginia (2002)
10. Q. Zhuge, B. Xiao, and E.H.-M. Sha: *Variable Partitioning and Scheduling of Multiple Memory Architectures for DSP.* In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS). (2002)
11. S. Sahni and T. Gonzales: *P-complete Approximation Problems.* Journal of the ACM **23** (1976) 555–565
12. M.X. Goemans and D.P. Williamson: *0.878-Approximation Algorithms for MAX CUT and MAX 2SAT.* In: Proceedings of the 26th Annual ACM Symposium on Theory of Computing. (1994) 422–431
13. M.X. Goemans and D.P. Williamson: *Improved Approximation Algorithms for MAX CUT and Satisfiability Problems Using Semidefinite Programming.* Journal of the ACM **42** (1995) 1115–1145
14. A. Frieze and M. Jerrum: *Improved Approximation Algorithms for Max k-Cut and Max Bisection.* Algorithmica **18** (1997) 61–77
15. Hromkovic, J.: *Algorithmics for Hard Problems.* Springer-Verlag, Berlin (2001)
16. K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita: *SDPA (Semidefinite Programming Algorithm), vers. 4.10*, Research Report on Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan. (1998)
17. S. Burer, R.D.C. Monteiro, and Y. Zhang: *Rank-two Relaxation Heuristics for Max-Cut and Other Binary Quadratic Programs.* SIAM Journal on Optimization **12** (2001) 503–521
18. S. Burer, R.D.C. Monteiro, and Y. Zhang: *CirCut vers. 1.0612, Fortran 90 Package for Finding Approximate Solutions of Certain Binary Quadratic Programs* (2000)
19. V. Zivojnovic, J.M. Velarde, C. Schager, and H. Meyr: *DSPstone – A DSP oriented Benchmarking Methodology.* In: Proceedings of the 6th International Conference on Signal Processing Applications and Technology. (1994)