VARIABLE PARTITIONING FOR DUAL MEMORY BANK DSPS

Rainer Leupers, Daniel Kotte

University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany

ABSTRACT

DSPs with dual memory banks offer high memory bandwidth, which is required for high-performance applications. However, such DSP architectures pose problems for C compilers, which are mostly not capable of partitioning program variables between memory banks. As a consequence, time-consuming assembly programming is required for an efficient coding of time-critical algorithms. This paper presents a new technique for automatic variable partitioning between memory banks in compilers, which leads to a higher utilization of available memory bandwidth in the generated machine code. We present experimental results obtained by integrating the proposed technique into an existing C compiler for the AMS Gepard, an industrial DSP core [1].

1. INTRODUCTION

It is a well-known problem in embedded system design that C compilers for digital signal processors (DSPs) still have problems with code quality. Compiler-generated code usually shows a significant overhead in terms of performance and/or code size as compared to manually written assembly code. This has been quantified in empirical studies [1, 2] for a number of different DSPs. As a consequence, DSP programmers frequently have to resort to time-consuming assembly programming whenever tight real-time or code size constraints have to be met. Essentially, this bottleneck in embedded software development is caused by the distortion between the machine-independent source code language (such as C) and the very specialized architectures of DSPs: High-level C data types and language constructs are not easily mapped into dedicated DSP machine instructions.

One major problem arises for C compilers for DSPs with *dual memory banks*. Such DSPs, e.g. the Motorola 56k and Analog Devices ADSP-210x series, are equipped with two separate memory banks (frequently denoted by X and Y) accessible in parallel. This is to accommodate the fact that many DSP routines, e.g. FIR filters, require the convolution of two data arrays as a kernel operation.

In order to make efficient use of the bandwidth increase offered by dual memory banks, the C program variables have to be appropriately partitioned between X and Y. For instance, for a typical array operation such as

```
for (i=0;i<N;i++) y += A[i] * B[N-i];
```

[1]Publication: ICASSP , Salt Lake City (USA), May 2001, ©IEEE

it must be ensured that arrays A and B are placed in memory banks X and Y, respectively, only in which case the entire loop body can be efficiently mapped to a single multiply-accumulate instruction on a DSP. Also for scalar variables the assignment to either X or Y plays an important role for code quality.

Many existing C compilers, e.g. the GNU gcc versions for Motorola 56k and ADSP-210x, cannot cope well with dual memory banks, but all program variables are assigned to just one bank. It is obvious that this naive strategy implies an enormous performance loss, since potential instruction-level parallelism is largely neglected.

The reason that dual memory banks are poorly exploited by compilers is a lack of suitable optimization techniques. The variable partitioning problem itself is difficult, since it requires a good balancing of the X/Y memory bandwidth among an entire program. As variables may have multiple occurrences in a program, different code segments may show contrary demands on assigning a certain variable to either X or Y. In addition, the X/Y assignment of variables has side effects on other modules of a compiler: Some data path input registers might not be available for both X and Y, which influences register allocation, and the potential parallelism generated by a certain variable partitioning must eventually be exploited in a dedicated scheduling phase.

The goal of this paper is to present a new technique for the variable partitioning problem for dual memory bank DSPs, which results in a significant code quality improvement at comparatively low computation time requirements. The technique has been integrated into a C compiler for a real-life DSP. We provide experimental results both in terms of performance and code size for a number of DSP routines.

The structure of the paper is as follows. In section 2 related work is discussed. Our target architecture, the Gepard DSP core designed by Austria Mikrosysteme International (AMS). is described in section 3. Section 4 gives an overview of the compiler environment, which the variable partitioning technique has been integrated into. The partitioning technique is presented in section 5. Section 6 provides experimental results obtained with our Gepard C compiler. Finally, conclusions are given.

2. RELATED WORK

An early approach for exploiting dual memory banks in compilers is [3], which deals with the Motorola 56k DSP architecture. Here, functional blocks of a DSP program are implemented in a meta-assembly language with symbolic memory references. Complete programs are composed of these

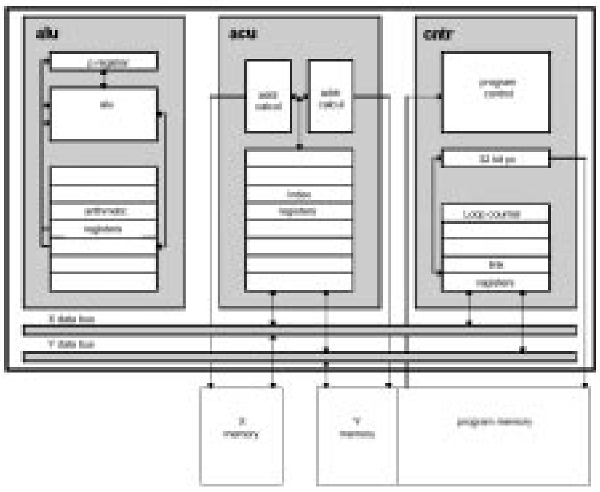

Fig. 1. *Gepard DSP architecture, ©AMS Austria Mikrosysteme International AG*

blocks, which can be instantiated from a library. After a peephole optimization of the composed programs, scheduling and register allocation take place. The assignment of program variables to the X/Y banks takes place using a greedy method: Variables are assigned to X and Y in an alternating fashion, according to their access sequence in the program code. The efficacy of this simple approach strongly relies on the hand-coding of the meta-assembly blocks, and the greedy partitioning may produce inferior results for complex programs.

In [4], a variable partitioning technique for a hypothetical VLIW DSP architecture is presented. The central data structure in that approach is an *interference graph*, whose nodes represent the program variables, while any edge between two nodes represents a potentially parallel memory access to the corresponding nodes. Each node has to be assigned to either X or Y in order to find a valid partitioning. The quality of a certain partitioning is measured by the sum of the graph edges, whose nodes have been not been assigned to different banks, which reflects the fact, that a potentially parallel access cannot be implemented according to that partitioning. The partitioning itself is performed heuristically. Although the experimental results are quite good, a major problem with the approach from [4] is that it is not clear how the technique performs for a real-life DSP with an irregular architecture instead of a synthetic VLIW.

A more practical partitioning technique has been described in [5], which also deals with the Motorola 56k DSP. After a pre-compaction step of the input program, given as symbolic assembly code, memory bank allocation and register allocation take place in a single phase. These problems are mapped to a *constraint graph* labeling problem. The constraint graph nodes represent variables to be mapped to X/Y memory banks or registers. The graph edges are used to reflect both the costs associated with a certain labeling and the code generation constraints imposed by the target DSP. The labeling takes place with a simulated annealing optimization algorithm. For the DSPStone benchmarks [1], typical code size reductions between 5 and 10 % have been achieved as compared to machine code without exploitation of dual memory banks. A problem with the simulated annealing approach is the huge runtime requirement, which might be in the range of minutes or even hours.

3. TARGET DSP ARCHITECTURE

The variable partitioning technique presented in this paper has been integrated into an existing ANSI C compiler for the AMS Gepard DSP [6, 7], an industrial parameterizable DSP core with dual memory banks. The coarse architecture is shown in fig. 1. There are two parallel memory banks (X and Y), each one equipped with an address calculation unit. The data path unit shows input registers for a hardware multiplier, an ALU, as well as an accumulator file. The Gepard instruction set comprises most typical DSP instructions, such as LOAD, STORE, arithmetic and logical operations, as well as address modifications.

For our purpose, the most interesting feature is that up to two memory accesses to X and Y can be issued in parallel to any arithmetic instruction in each cycle. Two parallel memory accesses always must take place on different memory banks. There are also constraints on register allocation: In case of two parallel memory accesses, the targets of LOAD operations are restricted to arithmetic registers. Additionally, address modifications within the same instruction cycle have to take place via index registers, while post-increment and decrement are not available in this mode. Under these restrictions, the goal of our optimization technique is to partition program variables in such as way, that the most effective use of parallel accesses to X and Y is ensured.

4. SYSTEM OVERVIEW

Our work builds on a C compiler for the Gepard DSP that has been provided by a third party. The compiler accepts ANSI C source code, which is first analyzed and translated into a three-address code intermediate representation (IR). The IR is optimized using machine-independent standard techniques such as constant folding and loop invariant code motion. Then, the Gepard backend translates the IR into assembly code.

The original compiler does not include any variable partitioning technique, but simply assigns all variables to the X memory bank. Therefore, instruction-level parallelism is hardly exploited. In principle, there are two points in the compilation flow, at which variable partitioning can take place: either as an *early partitioning* on the IR or as a *late partitioning* on the assembly code. The first variant shows the advantage, that the IR is machine-independent, so that early partitioning can be reused in a number of different backend. However, the IR provides no exact information on the set of memory accesses in the final assembly code, since not all variables might be assigned to memory, but some might be kept in registers. In addition, the backend can insert additional memory accesses due to spill code, which is not visible in the IR. Therefore, the early partitioning information passed to the backend may be either incomplete or overconstrained. In contrast, when using late partitioning, the exact set of variables is already known. However, if all variables are initially assigned to X, then the late partitioning, which reassigns a subset of variables to the Y bank, generally leads to violation of processor-specific constraints on register allocation. In order to meet such constraints, a significant amount of code restructuring would be required.

In order to overcome the phase-coupling problems associated with early and late partitioning, we have extended the compilation flow of the original compiler by the phases shown in the shaded area in fig. 2. An initial run of the backend is used to determine the exact set of memory accesses, while

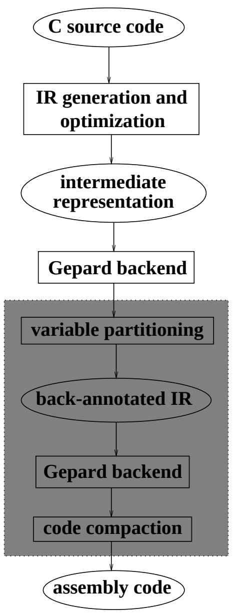

Fig. 2. *Compilation phases in the Gepard C compiler*

taking into account register variables and spill code, and placing all memory values into the X bank. Based on this information, the variable partitioning for maximizing parallel memory accesses is performed, and the IR is back-annotated with the X/Y bank assignment information for all variables. Then, a second run of the backend takes into account the partitioning information, and places variables into either X or Y, while meeting the corresponding register allocation constraints. Finally, a code compaction phase packs pairwise scheduling-independent operations into parallel assembly instructions. Details are given in the next section.

5. VARIABLE PARTITIONING

Similar to [4], our partitioning technique is based on the notion of an *interference graph*. However, in contrast to [4], this graph is not constructed for basic blocks only, but globally for entire functions, so as to reflect the potentially contrary variable assignment requirements of different blocks. In addition, we use an exact partitioning approach instead of a heuristic one.

First, for each assembly code function generated in the first run of the Gepard compiler backend, a data dependency graph is constructed:

Definition A *data dependency graph* (DDG) is a directed, node-labeled graph $G = (V, E, l)$, where each node $v \in V$ represents a memory access in the assembly code, and each edge $(v, w) \in E$ denotes a scheduling precedence between v and w. A node label $l(v)$ denotes the name of the variable accessed by v.

For a given DDG, the interference graph is constructed in such a way, that potential parallelism is reflected by graph edges:

Definition For some DDG $G = (V, E)$, the *interference graph* is an undirected graph $I = (V', E')$ with $V' = V$. There is an edge $(v, w) \in E'$, if and only if v and w are not reachable from each other via a path in G.

In order to reduce the interference graph size, we also apply a folding step: Each node set $v_1, \ldots, v_n \subseteq V'$ representing accesses to the same variable is merged into a single node v, and all edges containing $v_1, \ldots, v_n$ are redirected to v. Finally, all edges in $e = (v, w) \in E'$ are assigned a weight $A(e)$ denoting the sum of the total number of accesses to the variables represented by v and w in the function. The weight $A(e)$ is used to reflect the gain achieved by assigning v and w to different memory banks.

Obviously, the best partitioning is achieved if the interference graph is divided into two disjoint node sets X and Y, such that the sum of the edge weights between X and Y is maximal, since in this case the highest number of parallel memory accesses can be obtained. For a given folded and edge-weighted interference graph $I = (V, E, A)$, we use an *Integer Linear Programming* (ILP) approach to solve this NP-hard optimization problem. The ILP comprises the following solution variables:

$$\forall v_i \in V : \quad Z_i = \begin{cases} 1, & \text{if} \quad v_i \quad \text{is assigned to bank X} \\ 0, & \text{if} \quad v_i \quad \text{is assigned to bank Y} \end{cases}$$

$$\forall v_i, v_j \in V : \quad U_{ij} = \begin{cases} 1, & \text{if} \quad v_i \neq v_j \\ 0, & \text{if} \quad v_i = v_j \end{cases}$$

$$\forall e = (v_i, v_j) \in E, A(e) = g_{ij} : \quad W_{ij} = U_{ij} \cdot g_{ij}$$

The Z_i variables account for the X/Y assignment of the graph nodes, while the auxiliary variables U_{ij} indicate whether v_i and v_j have been assigned to different banks. These variables are used to compute the *implemented weight* W_{ij} of v_i and v_j: In case that $U_{ij} = 1$, up to g_{ij} parallel accesses to v_i and v_j can be achieved. Thus, the sum of W_{ij} over all v_i and v_j needs to be maximized as the objective function.

The setting of the $U_{ij} = 0$ in case that v_i and v_j are assigned to the same bank is enforced by the following constraint pairs:

$$U_{ij} \leq Z_i + Z_j$$
$$U_{ij} \geq 2 - Z_i - Z_j$$

The ILP corresponding to a given interference graph can be solved with existing tools. The setting of the Z_i solution variables accounts for the X/Y assignment of variables used in the second run of the compiler backend (see fig. 2). It should be noted that, although the ILP itself is solved optimally, the partitioning in general is only an approximation of the optimum, since only the *potential parallelism* is maximized. Whether or not a potentially parallel access can actually be implemented, is only determined during code compaction.

The code compaction phase packs potentially parallel instructions, so as to minimize the schedule length. In our approach, we use an efficient *list scheduling* algorithm [8]. First, a data dependency graph is built for the assembly code resulting from the second pass of the backend. Then instructions are scheduled step-by-step in accordance with the scheduling constraints imposed by the target processor (see section 3).

6. EXPERIMENTAL RESULTS

The partitioning and compaction techniques described above have been empirically evaluated by compiling the DSPStone

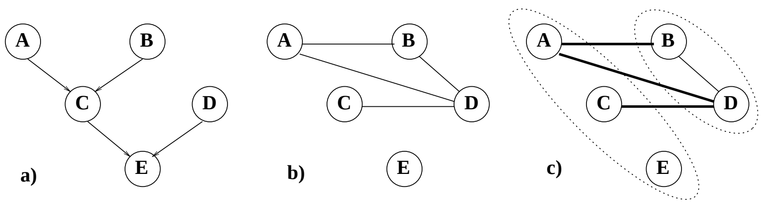

Fig. 3. *Graph models for variable partitioning: a) Example data dependency graph, b) corresponding interference graph, c) optimum partitioning into two sets $\{A, C, E\}$ and $\{B, D\}$. All edge weights are one in this simple example.*

C benchmarks [1] into assembly code for the Gepard DSP. As a platform, we have used a 333 MHz Pentium II Linux PC.

The exploitation of parallel accesses to the X/Y memory banks leads to code quality improvements both in terms of code size and performance. Code size is affected, since two originally separate memory accesses may be encoded into a single instruction for the Gepard DSP. The performance increase naturally is due to the better exploitation of memory bandwidth.

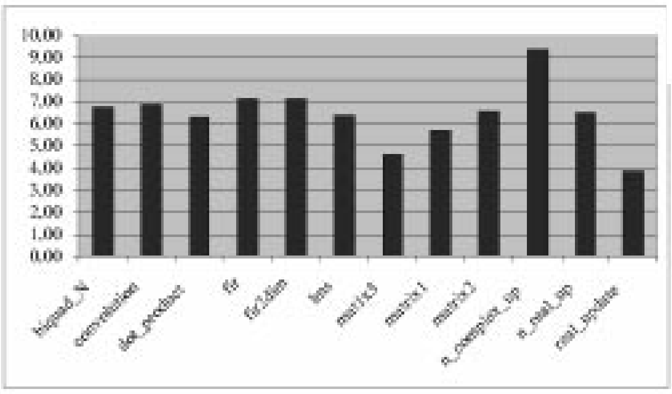

Fig. 4. *Code size reduction (%)*

Fig. 4 shows the percentage of code size reduction for the compiled benchmarks, as compared to the original, unoptimized assembly code. The gain ranges from 3.85 % (real_update) to 9.38 % (n_complex_updates). This is comparable to the results for the Motorola 56k reported in [5]. However, the compilation time is dramatically lower: The CPU times for partitioning (including ILP solving) and code compaction have been within 2 seconds for all benchmarks. This is due to the fact that we employ a relatively simple ILP formulation for the core partitioning problem, while the remaining code generation tasks are solved with fast standard techniques in the existing compiler framework. The ILP approach at least works well for small time-critical DSP kernel routines. For very large programs, the ILP partitioning technique might need to be replaced by more efficient heuristics, though.

Even higher code quality improvements are obtained w.r.t. performance. Fig. 5 gives the percentage of speedup as compared to the unoptimized assembly code without exploitation of parallel memory accesses. Here, the gains range between 4.11 % (real_update) and 17.12 % (matrix1).

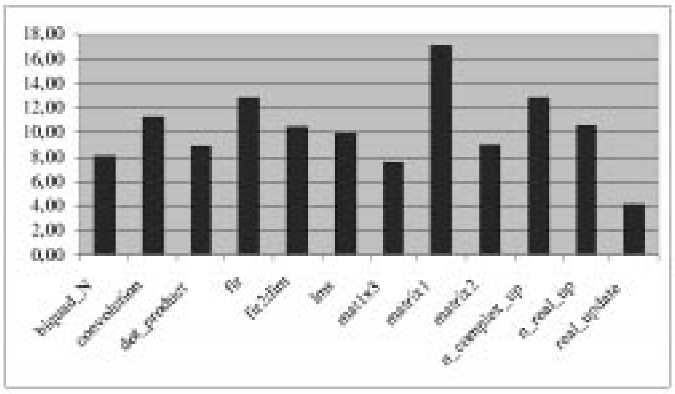

Fig. 5. *Execution time speedup (%)*

7. CONCLUSIONS

We have presented a new technique for variable partitioning in C compilers for DSPs with dual memory banks. The goal of this technique is to enable compilers to exploit the available memory bandwidth in such DSPs without the need for C language extensions or assembly programming. Experimental results for the Gepard DSP show that the code quality is improved for realistic DSP routines. The key contributions of this paper are twofold: For the DSPStone benchmarks, similar improvements as reported in previous work are obtained at much lower computation time requirements. In addition, we have shown how an existing compiler infrastructure can be enhanced by a variable partitioning technique without the need for an extensive compiler redesign.

8. REFERENCES

[1] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[2] M. Coors, O. Wahlen, H. Keding, O. Lüthje, H. Meyr: *TI C62x Performance Code Optimization*, DSP Germany, 2000

[3] D.B. Powell, E.A. Lee, W.C. Newman: *Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams*, Proc. ICASSP, 1992

[4] M. Saghir, P. Chow, C. Lee: *Exploiting Dual Data-Memory Banks in Digital Signal Processors*, 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996

[5] A. Sudarsanam, S. Malik: *Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs*, ACM TODAES, vol. 5, no. 2, 2000

[6] A. Gierlinger, R. Forsyth and E. Ofner: *Gepard – A Parameterisable DSP Core for ASICs*, Proc. ICSPAT, 1997

[7] Austria Mikro Systeme International: asic.amsint.com/databooks/digital/gepard.html, 2000

[8] S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallett: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, Vol. 30, No. 7, 1981