Published in *Proceedings of the 6th International Conference on Signal Processing Applications and Technology*, pp. I-866–I-871, October 1995.

Automatic Data Partitioning for HLL DSP Compilers

Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee

{saghir, pc, corinna}@eecg.toronto.edu

Department of Electrical and Computer Engineering, University of Toronto
10 King's College Road, Toronto, Ontario M5S 1A4
CANADA

ABSTRACT

With the proliferation of larger and more sophisticated DSP applications, there is a trend towards a heavier reliance on HLLs to program DSPs. With this trend comes a need to automate the exploitation of DSP-specific architectural features and to relegate it to the compiler. Automatic data partitioning is one technique aimed at exploiting the parallelism in accessing the multiple memory banks that typically characterize many DSPs. It relieves the programmer from having to partition data manually and ensures that data is partitioned in a manner that improves performance across an entire application.

In this paper, we describe two automatic data partitioning algorithms that we developed as part of our research into HLL DSP compilers. We also show the impact of the algorithms on the performance of our suite of DSP benchmarks. Our results show that, on average, the two algorithms increase the parallelism and the execution-time speedup of our DSP kernels by 42% and 43%, compared to an ideal increase of 45%. The algorithms also increase the parallelism and execution-time speedup of our DSP applications by 6% and 8%, compared to an ideal increase of 18%. These results indicate that it is possible for a HLL DSP compiler to automatically partition data and improve the execution performance of DSP kernels and applications.

1. INTRODUCTION

Programmable digital signal processors (DSPs) often use multiple data memory banks to facilitate the exploitation of parallelism in accessing operands from memory. To utilize this capability, however, the data must be properly partitioned and distributed among different memory banks.

To illustrate the importance of data partitioning on execution performance, consider the Nth-order FIR filter in Figure 1 implemented in the C-language and the pseudo-assembly language of a hypothetical DSP similar to the Motorola DSP56001 [1]. This hypothetical DSP has two data-memory banks that can each be accessed once every clock cycle. But to fetch the elements of arrays **a** and **b** in parallel (lines [2] and [4]), the arrays must be stored in separate memory

```
sum = 0;
for (i = 0; i < N; i++)
    sum += a[i] * b[i];
```

(a)

```
[1] R1=0;i=0;j=0;
[2] R2=a[i++];R3=b[j++];
[3] repeat #N-1
[4] R1=R1+(R2*R3);R2=a[i++]; R3=b[j++];
[5] R1=R1+(R2*R3);
```

(b)

Figure 1: (a) C-language implementation of an N-th order FIR filter (b) Pseudo assembly language implementation of the same filter

banks. Conversely, if the arrays are stored in the same memory bank, the elements of the arrays would have to be fetched sequentially, and the execution of the code would be slowed down. The ability to access multiple memory banks in parallel with other operations is fundamental to most DSPs. However, failure to properly distribute the data among the memory banks can limit the exploitation of such parallelis and can therefore degrade execution performance.

Data partitioning is currently performed manually by the programmer who must typically resort to the use of assembler directives or compiler pragmas to control the partitioning of data in a program [2]. As programs become larger and more sophisticated, it becomes very difficult to partition data manually and to ensure that the partitioning results in optimal execution performance. Automatic data partitionin is therefore becoming a necessity, especially with the current trend towards a heavier reliance on high-level languages (HLLs) to program DSPs. Automatic data partitioning relieves the programmer from this task and helps improve performance across an entire program.

As part of our research into HLL DSP compilers, we developed two algorithms for automatic data partitioning. In this paper, we describe and compare the performance of these two algorithms. In section 2, we briefly describe our

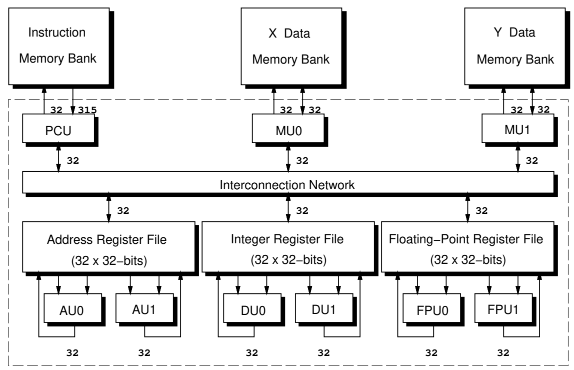

Figure 2: VLIW Model Architecture

research framework and show how automatic data partitioning fits into it. In section 3, we describe the algorithms in some detail and explain how each one evolved. In section 4, we present and discuss the results of our study on the impact of the algorithms on the performance of our suite of DSP benchmarks. Finally, in section 5, we summarize our results and present our conclusions.

2. EXPERIMENTAL FRAMEWORK

Automatic data partitioning is part of our research into DSP architectures and HLL DSP compilers. The main objectives of our research are to improve current DSP compiler technology and to find ways of making DSPs easier targets for HLL compilers. To help us perform this research, we developed an experimental framework that consists of a model DSP architecture, an instruction-set simulator, an optimizing C compiler, and a suite of DSP benchmarks. A more detailed description of this framework can be found in [3].

Figure 2 shows our model DSP architecture which is based on a *Very Long Instruction Word* (VLIW) architecture [4]. A VLIW architecture typically consists of multiple functional units whose simultaneous operation are controlled by corresponding fields of a very long instruction. This makes the VLIW architecture a very flexible target for compilers that can schedule parallel operations into long instructions. Our choice was therefore based on this flexibility offered to the compiler in exploiting parallelism.We also simulated the operation of the architecture on our instruction-set simulator, and used it to gather execution-performance statistics. In later stages of our research, we will focus on making the architecture more feasible to implement in hardware.

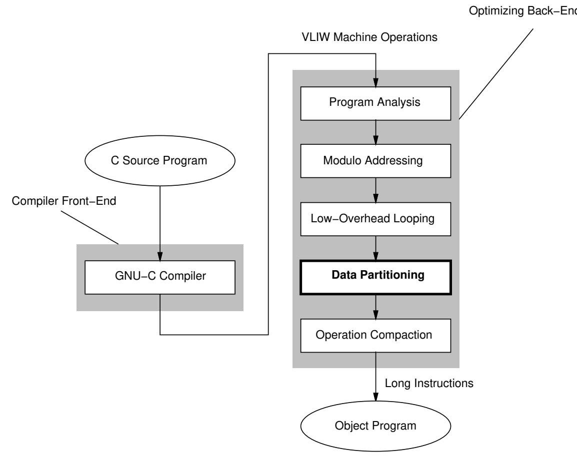

Figure 3: Optimizing C Compiler

The architectural features that are of interest to this paper are the two data-memory banks (X and Y), and the two memory-access units (MU0 and MU1) used to access them. Each one of these units can execute load or store operations to access the corresponding data memory bank. To maximize the exploitation of load/store parallelism, data must be appropriately distributed between these two data-memory banks.

Figure 3 shows a block diagram of our optimizing C compiler, which consists of a GNU-C [5] compiler front end and an optimizing back end. The front end translates C-language source programs into a sequence of VLIW machine operations. These operations are fed into the optimizing back end where they are analyzed and then optimized by a number of passes to exploit specific architectural features of our target architecture. Of particular interest to this paper is the *data partitioning pass*, which partitions the global variables of a program to expose the most parallelism in load and store operations. This parallelism is communicated to the following *operation-compaction pass*, which packs parallel operations into long machine instructions and generates these long instructions as the output of the compiler.

Finally, to evaluate our model architecture and our compiler, we developed a suite of DSP benchmarks that we coded in C. Our previous work [6] has shown that when evaluating the performance of a DSP, it is necessary not only to rely on performance measurements of kernels, but to also consider the performance of complete programs that represent typical applications. We therefore chose twelve kernels and six applications as our benchmarks.

The kernels are based on six algorithms that are commonly used in DSP applications. For each algorithm, we implemented two kernels of different problem sizes. The kernels include: radix-2, in-place, decimation-in-time FFT; FIR filter; IIR filter; normalized lattice filter; LMS adaptive FIR filter; and matrix multiplication.

The applications are from the areas of speech and image processing, and they include: ADPCM speech encoder based on CCITT G.721; LPC speech encoder; spectral analysis using periodogram averaging; edge detection using Sobel operators and 2D convolution; image compression using the discrete cosine transform; and image enhancement using histogram equalization.

3. ALGORITHMS FOR AUTOMATIC DATA PARTITIONING

We developed two algorithms for automatic data partitioning and implemented each as an optimizing pass in our C compiler. We developed the second algorithm to overcome what we perceived to be limitations in our initial algorithm and to ensure a better impact on performance.

Both algorithms are based on partitioning the nodes of an undirected interference graph that is constructed prior to partitioning. The nodes of the graph represent the global variables of a program, and an edge between a pair of nodes in the graph indicates that the corresponding variables may be accessed in parallel. A weight is also assigned to each edge to represent the degradation in performance if the corresponding variables are not accessed in parallel. The algorithms use these weights to compute a minimum-cost data partitioning that will result in the least degradation in performance. After constructing the interference graph, the nodes are partitioned into two sets, and the corresponding variables are stored in different memory banks. All load and store operations in the program are then assigned to an appropriate memory-access unit based on the memory bank assignment of the data they access. These assignments are later used by the operation-compaction pass when issuing long machine instructions.

To simplify the design of the algorithms, we assume that all variables are global. Global variables are easy for the compiler to identify and partition because they are explicitly declared in the program and can be indirectly accessed using different address registers. Local variables, on the other hand, are declared on the stack and are indirectly accessed using a single stack pointer. Because the stack operates under the assumption of a single memory space, it is difficult to partition its contents among multiple memory banks and use a single stack pointer to access them. That is why we currently partition stack references that are due to function calls and returns in a mechanical fashion: successive stack locations are assigned to different memory banks, and the stack pointer is used to access the same location in both memory banks.

3.1. Maximal Independent Set (MIS) Data Partitioning

The first algorithm that we developed is called *maximal-independent set* data partitioning. Since most DSP algorithms consist of loops that perform repetitive arithmetic operations, this algorithm aims to expose the parallelism in load operations that fetch operands for such arithmetic operations. The algorithm adds an edge to the interference graph between a pair of nodes that correspond to variables that are fetched as operands for an arithmetic operation. It also assigns a weight to each edge to give more importance to exploiting the parallelism in accessing operands from inner loops.

The algorithm partitions the nodes of the graph into two sets: a MIS on the nodes of the graph and its corresponding complement set. By definition, a MIS is any subset of nodes such that no two nodes in the subset are edge-connected, and such that each node not in the subset is edge-connected to at least one node in the subset [7]. The complement set consists of those nodes that are not in the MIS. Since the nodes of the MIS are not edge-connected, their corresponding variables need not be fetched in parallel, and are therefore stored in the same memory bank. The variables corresponding to the nodes of the complement set are stored in the other memory bank. However, because the nodes of the complement set are generally edge-connected, storing the corresponding variables in the same memory bank may degrade performance. To overcome this problem, the algorithm tries to find a minimum-cost partitioning. Since an MIS is not unique, the algorithm computes all possible MIS's, and selects the partitioning that incurs the least cost. Here, the cost is equal to the sum of the weights of the edges connecting the nodes of the complement set. The computational complexity of partitioning the graph is $O(2^{|E|})$, where $|E|$ represents the number of edges in the graph.

A major limitation of this algorithm is that it only attempts to expose the parallelism in load operations that fetch operands for arithmetic operations. This may cause the algorithm to miss out on exploiting other instances of parallelism among load and store operations and thus yielding better overall performance. The main limitation of this algorithm, however, is that it does not ensure an optimal partitioning that exposes the most parallelism or yields the most speedup in execution time. This is due to the partitioning being based on selecting a minimum-cost solution among a class of partitionings having the specific characteristic of being a maximal independent set, instead of finding a solution that is minimum-cost across all possible partitionings. Although this latter problem is provably NP-complete [8], heuristics exist for generating near-optimal solutions [9].

3.2. Compaction-Based (CB) Partitioning

To overcome the limitations of our initial algorithm, we developed a *compaction-based* data partitioning algorithm. Whereas the initial algorithm attempted to expose parallelism among load operations only, this algorithm attempts to expose parallelism in both load and store operations. The algorithm relies on the operation-compaction pass to suggest all pairs of load or store operations that can be executed in parallel and adds an edge to the interference graph between the nodes corresponding to the variables being accessed by every such pair of operations. Similar to the initial algorithm, this algorithm assigns a weight to each edge equal to the depth of the basic block containing the pair of load or store operations in the control-flow graph of the program. This again ensures that exploiting load/store parallelism inside inner loops is given high priority.

Unlike the initial algorithm, where a minimum-cost partitioning is selected from a class of partitionings, this algorithm actively searches for a minimum-cost partitioning among all possible solutions. Even though this problem is NP-complete, we can ensure a near optimal partitioning by using a greedy approach to partition the nodes of the graph. The algorithm initially stores all nodes in one of two sets, with the second set being empty. It also sets the initial cost for the partitioning to the sum of the weights of the edges connecting the nodes in the first set. The algorithm then selects a node from the first set such that its removal results in the greatest decrease in cost, and stores the node in the second set. It then selects another node from the first set such that its removal from the first set and its addition to the second set results in the greatest decrease in cost. Note that adding a node to the second set may increase the cost if the node is edge-connected to any of the nodes in the second set. That is why a node is only moved from the first set to the second if its removal results in an overall decrease in cost. This process is continued until reducing cost is no longer possible, upon which the data is partitioned. The variables corresponding to the nodes in the two sets are then stored in separate memory banks.

4. RESULTS AND ANALYSIS

To measure the impact of the algorithms on our suite of DSP benchmarks, we chose overall parallelism as our metric of performance. The data partitioning pass exposes parallelism in load and store operations to the operation-compaction pass, which exploits this parallelism when issuing long machine instructions. Since different data partitioning algorithms may differ in their exposure of load/store parallelism, the operation compaction pass may generate different instruction schedules exploiting different levels of parallelism for similar programs. By exposing as much load/store parallelism as possible, the data partitioning algorithms can therefore help improve the overall exploitation of parallelism. Since we are assuming that all instructions execute in a single clock cycle, overall parallelism is also indicative of the amount of speedup achieved in execution time. Thus, by measuring the impact of the data partitioning algorithms on parallelism, we are also measuring their impact on execution-time speedup. All references to parallelism should therefore be understood to mean speedup in execution time.

To assess the performance of our algorithms, we first determined bounds for such performance. To do this we compiled our benchmarks with the data partitioning pass deactivated and used the resulting levels of parallelism as lower bounds in our performance evaluation. Recall that when the data partitioning pass is deactivated, the compiler does not exploit load/store parallelism. To obtain an upper bound on performance, we also simulated the execution of the benchmarks for an ideal case where we assumed that the data is stored in a single, dual-ported memory bank. A dual-ported memory bank

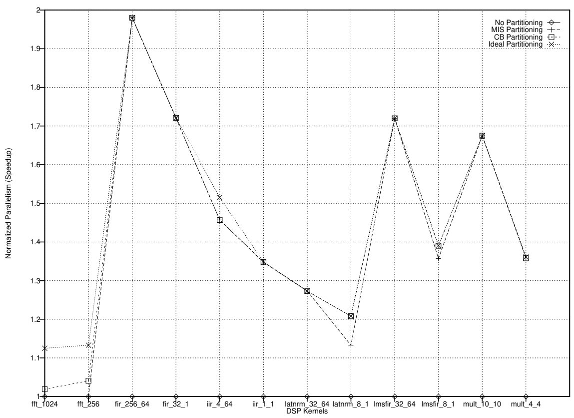

Figure 4: Normalized Parallelism (Speedup) for DSP Kernels

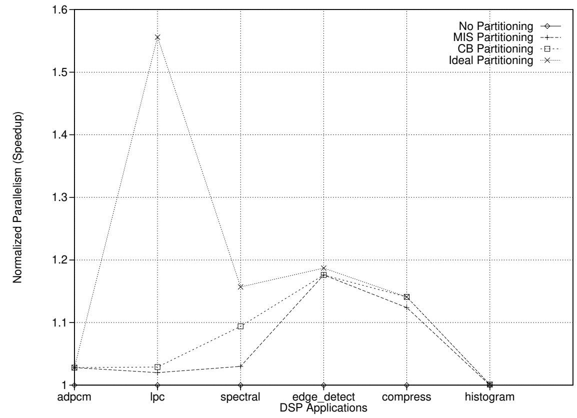

Figure 5: Normalized Parallelism (Speedup) for DSP Applications

enables two simultaneous accesses to be made to two different memory locations in the same memory bank. This represents an ideal case for the exploitation of parallelism because parallel load and store operations may be executed whenever possible without being constrained by the placement of data in memory. Recall that with data partitioning, the ability to exploit load/store parallelism depends on where the data is stored in memory. Thus, for example, a pair of load operations cannot be executed in parallel if the operands they fetch are stored in the same memory bank. This is not a problem with a dual-ported memory bank, where both operands can be stored in the same memory bank and accessed simultaneously by the load operations. However, multi-ported memories are generally more costly to implement on an integrated circuit than multiple memory banks, and that is why very few DSPs make use of them. Deactivating the data partition pass and using ideal partitioning also affect the operation-compaction pass and the overall exploitation of parallelism in the generated instruction schedule.

The impact of both data partitioning algorithms on overall parallelism for the DSP kernels and applications is shown in the graphs of Figures 4 and 5, respectively. These graphs show the levels of overall parallelism normalized to the levels of parallelism when no data partitioning is used. Each figure shows four different data sets that correspond to no data partitioning (No Partitioning), maximal-independent-set partitioning (MIS Partitioning), compaction-based partitioning (CB Partitioning), and ideal partitioning (Ideal Partitioning). Each point of a data set represents the ratio of the parallelism attained when the corresponding partitioning is used to the parallelism attained when no partitioning is used.

Our results show that, on average, the MIS and CB partitioning algorithms increase the overall parallelism in the kernels by 42% and 43%, compared to an ideal increase of 45%. The algorithms also increase the overall parallelism in the applications by 6% and 8%, compared to an ideal increase of 18%. The differences between the results of the kernels and the applications are mainly due to the differences in the levels of parallelism present in kernels and applications [6]. Kernels typically consist of short loops that perform repetitive arithmetic calculations. These loops commonly exhibit high levels of parallelism with high concentrations of load and store operations. Exploiting load/store parallelism inside these loops therefore has a great impact on overall parallelism. On the other hand, applications also consist of other sections of code, such as control code or the intervening code between loops. These sections of code generally exhibit low levels of parallelism, but also contain many load and store operations. Exploiting the load/store parallelism in these sections of code therefore has little impact on overall parallelism.

The algorithms also appear to be more successful at exposing load/store parallelism in the kernels than in the applications. The results show that most of the parallelism found by ideal partitioning is also found by the algorithms for the kernels. On the other hand, the parallelism found by the algorithms in the applications is, on average, lower than that found by ideal partitioning. This is mainly due to the greater variance in parallelism found by the algorithms and ideal partitioning in such applications as **lpc** and **spectral**.

Our results also show that the CB partitioning algorithm performs slightly better than the MIS partitioning algorithm in improving the overall parallelism of our benchmarks. This is most evident in the **`latnrm_8_1`** kernel and the **`spectral`** application, where the CB algorithm yields 7.5% and 6.4% more parallelism than the MIS algorithm. The better performance of the CB algorithm is mainly due to its active search for a minimum-cost partitioning. Recall that the MIS algorithm only selects a minimum-cost partitioning among a class of partitionings that are maximal independent sets but that may not necessarily be minimum-cost. The CB algorithm therefore succeeds in exposing higher levels of load/store parallelism, especially inside loops.

Finally, our results show that there is still room for improving the data partitioning algorithms. This is most evident in the **`fft_1024`** kernel and the **`lpc`** application, where up to 10.6% and 52.7% more parallelism can ideally be attained over that which is currently attained by the CB algorithm. On comparing the code generated under the ideal case and using the CB algorithm, we found that the higher levels of parallelism attained under ideal partitioning are mainly due to better exploitation of load/store parallelism in loops that are repeated a variable number of times. Because the CB algorithm currently relies on an estimate of cost that does not reflect the frequency of execution, the importance of partitioning the load and store operations inside such loops is not emphasized. This suggests that better cost estimates may result in better data partitioning and higher levels of parallelism.

5. CONCLUSIONS

In this paper, we described two automatic data partitioning algorithms that we developed as part of our research into HLL DSP compilers. We also presented the results of our study of the impact of the algorithms on the performance of our suite of DSP benchmarks. Our results show that, on average, the two algorithms increase the parallelism and the execution speedup of our DSP kernels by 42% and 43%, compared to an ideal increase of 45%. The algorithms also increase the parallelism and the execution speedup of our DSP applications by 6% and 8%, compared to an ideal increase of 18%. These results show that it is possible for a HLL DSP compiler to automatically partition data and improve the execution performance of DSP kernels and applications.

The results also show that there is still room for improving performance, especially in the DSP applications. Although the MIS and CB algorithms currently produce similar performance results, we believe that using better cost estimates will improve their performance. We also believe that the CB algorithm's active search for a globally optimal solution will enable it to achieve better performance than the MIS algorithm, especially on such benchmarks as the **`fft`** kernels, and the **`lpc`** and **`spectral`** applications. Finally, we will also be extending the algorithms to handle local variables.

6. ACKNOWLEDGEMENTS

This research has been funded by a grant from the Information Technology Research Centre, a Centre of Excellence supported by Technology Ontario.

7. REFERENCES

[1] Motorola, *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.

[2] Benjamin Krepp, "DSP-Oriented Extensions to ANSI C," *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 695-702, DSP Associates, 1994.

[3] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, "Towards Better DSP Architectures and Compilers," *Proceedings of the International Conference on Signal Processing Applications and Technology*, pp. 658-664, DSP Associates, 1994.

[4] Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Symposium on Computer Architecture*, pp. 140-150, IEEE, 1983.

[5] Richard M. Stallman, *Using and Porting GNU C*, Free Software Foundation, Inc., 1990.

[6] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee, "Application-Driven Design of DSP Architectures and Compilers," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. pp. II-437-440, IEEE, 1994.

[7] Allan Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[8] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[9] Thomas H. Cormen, Charles E, Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.