

A COMPILER FOR MULTIPROCESSOR DSP IMPLEMENTATION

Phu Hoang and Jan Rabaey

Department of Electrical Engineering
University of California, Berkeley, CA 94720

ABSTRACT

McDAS, a software environment designed to support the real-time implementation of Digital Signal Processing (DSP) applications onto multiple processors is described. Users program their algorithms as they would on a single processor, and McDAS will automatically schedule and compile the program onto the target multiprocessor. The scheduler maximizes the computational throughput by simultaneously considering pipelining, retiming, and parallelism while accounting for processor and memory constraints, as well as interprocessor communication delays. If the architecture is scalable or configurable, the scheduler can be invoked with different numbers of processors and multiprocessor topologies to explore various implementations. The code generator is similarly retargetable to different memory architectures and core processors. Data buffers and synchronizations are automatically inserted to ensure correct execution. The code generated can also execute the algorithm with either quasi-infinite precision or bit-true precision, allowing the algorithm designer to assess the effects of quantization and truncation. The results on a set of benchmarks demonstrate McDAS's ability to achieve near optimal speedups across programs with different types of concurrency, with very good scalability with respect to processor count.

1. INTRODUCTION

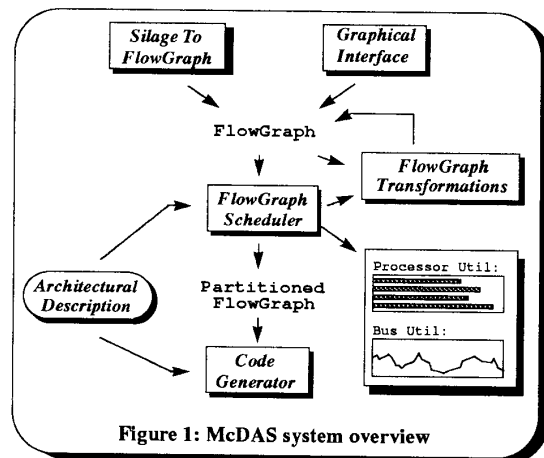
With the ever greater increase in complexity and computational requirements of digital signal processing (DSP) applications, designers are turning more and more to multiprocessors as the cost-effective alternative to custom implementations. Recently, a number of research and commercial prototype multiprocessor systems have been built to support these high performance applications. The major obstacle to the prevalent use of these systems for real-time implementations, however, has been the absence of adequate CAD tools to automatically partition and schedule the program onto the multiple processors, coordinate their communications and synchronizations, and synthesize the required code. Most multiprocessor DSP design environments currently available are block-diagram based, where tasks are represented as blocks connected by edges representing data flow. Examples of these systems include the Gabriel system from U.C. Berkeley[1], the Block Diagram Compiler (BDC) from Lincoln Labs[2], the ZC compiler from Carnegie Mellon[3], the cyclo-static scheduler at Georgia Tech[4], and the MultiProx system from Comdisco Systems, Inc.[5]. The *Gabriel* system, a design environment for DSP based on synchronous data flow principles, uses a modified list scheduling algorithm that considers communication delays and processor constraints to map tasks onto multiple DSP processors. However, it does not take advantage of the infinite time loop of DSP applications to exploit pipelining concurrency. The same is true for the BDC compiler, which uses simulated annealing to meet memory and real-time constraints. The ZC compiler also only exploits parallelism, but pays special attention to blocks with

massive data-parallelism to balance loads. The cyclo-static scheduler was designed to support cyclo-static realizations of DSP flow graphs. Cyclo-static scheduling exploits all concurrency to yield optimum schedules, but requires extensive communication support, and does not consider communication delays and processor and memory constraints. Furthermore, the search for optimal solutions is exponential. The MultiProx system from Comdisco currently does not employ a scheduler, requiring the user to partition the DSP program by hand.

All systems above, with the exception of the ZC compiler, do not allow the nodes or blocks to be decomposed into smaller sub-nodes during scheduling. This is a serious drawback as often it is the large grain nodes that contain the most concurrency to be exploited. The McDAS compiler, besides being able to fully exploit all types of concurrency and meeting all resource constraints, is further unique in that it can perform flowgraph transformations to decompose large nodes to expose more concurrency when needed.

2. McDAS OVERVIEW

The McDAS system is composed of 5 modules which operate on a centralized control/data flowgraph (CDFG) database as shown in Figure 1. The DSP algorithm is described using Silage[6], a



signal-flow language developed especially for DSP specification. As such, it contains language support for DSP operations such as sample delays, multirate sampling, and fixed point data types. The *SilageToFlowGraph* module converts a Silage program to a CDFG. Aside from the standard arithmetic operations, the CDFG allows for a number of macro control flow operations such as loops, and function calls. The introduction of those operations

results in a hierarchical graph, where the body of a loop or function definition is represented by a sub-graph, which is contracted into a single node at the next hierarchy. The *FlowGraph Scheduler* reads the CDFG and the architecture description, performs the scheduling, and writes back a different CDFG decorated with all the scheduling and assignment information. This can be repeated for different numbers of processors and different architecture topologies. Display tools can be invoked to show the scheduling results such as processor utilization, bus congestion, speedup, etc. A history mechanism stores all previous designs and allows easy backtracking. Once the scheduling is done, the *Code Generator* takes the decorated CDFG and generates code for each processor. The architecture description is also needed here to determine the memory layout and synchronization strategy. The Silage translation and flowgraph scheduling process also uses a number of flowgraph transformations which are stored in the *FlowGraph Transformation* module. An X-window user interface allows the user to invoke all the tools from a single environment. This command window is shown in Figure 2. It shows the result of scheduling a *histogram* example on 6 processors connected by a single shared bus. The main window displays in text the processor assignments and utilization. In the remainder of the section, we discuss the Silage to CDFG translation process. Section 3 summarizes how we estimate computation times and interprocessor communication costs. Section 4 gives a brief description of our scheduling algorithm. Section 5 discusses our code generation strategy and Section 6 summarizes our results for various benchmarks.

2.1. Hierarchical Flowgraph Generation

The *SilageToFlowGraph* module was developed to translate a Silage program into a flowgraph with the same hierarchical structure. In order to build the flowgraph, all dependencies among the Silage variables must be determined. For scalars, this is straightforward. For arrays, however, the indices determine the dependency. Careful book-keeping of variable indices in arrays are necessary for accurate dependency analysis. This can be done as Silage forces them to be manifest expressions. After the Silage translation, a number of standard, architecture independent compiler transformations such as dead code elimination, manifest expressions, common sub-expressions and algebraic identities, are executed. Multirate applications are also reduced to a single sample rate via a graph clustering transformation. In this transformation, nodes which run at the same rate are grouped into a process. The rate of the slowest running process is taken as the basic rate, and the higher rate processes are invoked the required number of times using loops. Since these rates may not be integral multiples of the slowest rate, a greatest common divisor algorithm is used to obtain the minimum invocation necessary. This greatly reduces the size of data buffers which are needed between these communicating processes.

3. ESTIMATION

3.1. Computation and Memory Estimation

Real-time DSP implementations on multiprocessors require the schedule to be determined at compile-time to reduce run-time overheads. For good results, accurate estimates of the computation and communication costs are crucial. These estimates characterize the underlying architecture of the multiprocessor system for the scheduler. To calculate the computation costs on a particular processor, precise execution times or costs of primitive operations such as additions and multiplications were collected. Costs were also measured for the overheads of performing loop increments, loop tests, and function calls. With these values, the cost of every node in the hierarchy of any CDFG can be estimated by traversing the flowgraph bottom up, accumulating computation times of primitive nodes into subgraphs, and so on up to the root graph.

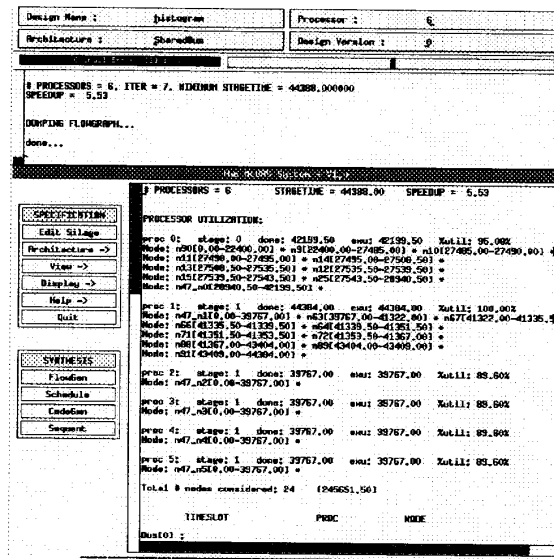


Figure 2: McDAS User Interface

Executing a node incurs memory storage for the input and output variables of the node. Similarly, an edge between two nodes assigned to different processors incurs buffer memory storage. The memory requirements to execute primitive nodes can be estimated, and the same bottom up strategy can estimate memory requirements of hierarchical nodes.

3.2. Communication Estimation

The communication costs depend on the amount of data being sent and the distance between the source and destination processors. The cost to send one unit of data between two processors without contention is available from the target architecture's manufacturer. To take into account bus contention in our scheduling, we explicitly schedule interprocessor data transfers onto the corresponding bus or busses. This gives the scheduler a very clear picture of the bus loading. When a data transfer needs a bus that has already been reserved for that time slot, it is scheduled on the next available time slot on the bus (assuming this is what the hardware will do). The abstraction of the communication delays presented to the scheduler is summed up in a parameter $\xi(n_i, p_k)$, called the *Earliest Starting Time* of node n_i on processor p_k . This parameter makes sense when all predecessor nodes to n_i have been scheduled, and is the earliest time all output data from these predecessor nodes can all be available at processor p_k for n_i to start. Note that bus congestions are taken into account when calculating the arrival times of the input data.

4. SCHEDULING

Given a CDFG, and a description of the target multiprocessor (number of processors, memory capacity, topology, estimation of computation costs of primitive operations, estimation of data transfer rate), the *FlowGraph Scheduler* module proceeds to maximize the computational throughput of the resultant implementation. Initially, the computation times and memory requirements of all nodes in the hierarchy are calculated using the methods outlined above. The scheduler starts searching at the top-level flowgraph hierarchy, and proceeds top-down. The goal of the scheduling is to exploit all available concurrency including pipelining, parallelism, and retiming to maximize throughput. We define the *stagetime* T to be the reciprocal of the throughput of

the system. T equals the time allocated to each pipeline stage, and thus to each processor in the stage. The search repetitively calls a scheduling routine which determines how many processors are needed to execute the algorithm, given a stagetime T . (This routine also effectively partitions the flowgraph into pipeline stages with one or more processors assigned to each stage). If the number of processors returned equals the amount given, we have a feasible solution. If it is greater, we have to increase T to give each processor more time. If it is less, we can decrease T so that more processors will be used. We converge to the minimum T solution which uses the available number of processors. As the algorithm attempts to minimize T , it may approach a point where there are nodes in the graph that are as large as the stagetime. If the search decides that the stagetime can be decreased further, node decomposition transformations are applied to break these nodes down to expose more concurrency. The pseudo code for the algorithm is shown in Figure 3.

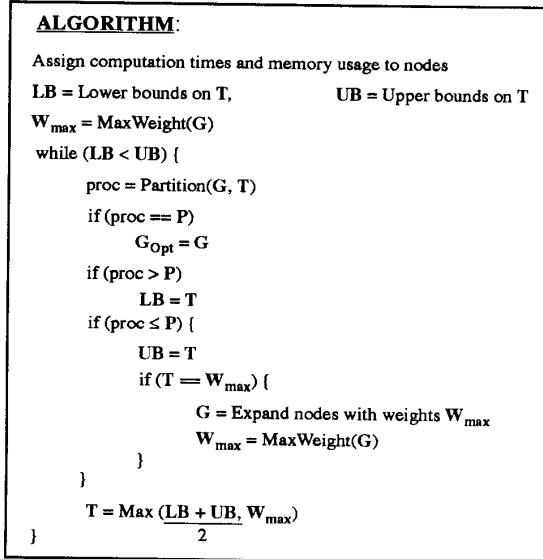


Figure 3: McDAS Scheduling Algorithm

The core of the search is the probing function *Partition()*, which does the actual flowgraph partitioning, scheduling, and processor assignment. Given the stagetime T , *Partition()* traverses the graph from input to output, and partitions the graph into stages of pipelines. Nodes are scheduled onto a processor until the total computation costs of the nodes plus the communication cost of output edges exceeds the stagetime T . Once a pipeline is filled, the scheduler proceeds to schedule the remaining nodes on the next pipeline stage. At any stage, the algorithm may decide to use multiple processors working in parallel. At the end, the graph is partitioned into a number of pipeline stages, where some stages may have more than one processors. The total number of processors needed is returned. An example of how *Partition()* works on a simple acyclic graph is shown in Figure 4. Values inside the nodes represent estimates of their computation costs, and values on the edges represent the additional delays for communication. When two nodes are assigned to the same processor, communications between them incur no cost. In scheduling node n_i , *Partition()* examines the *Earliest Starting Time* $\xi(n_i, p_k)$ for different feasible p_k 's to pick the best candidate.

The ability to simultaneously exploit many types of concurrency at different levels of granularity allows the scheduler to consider functional parallelism, data-partitioning, functional pipelining,

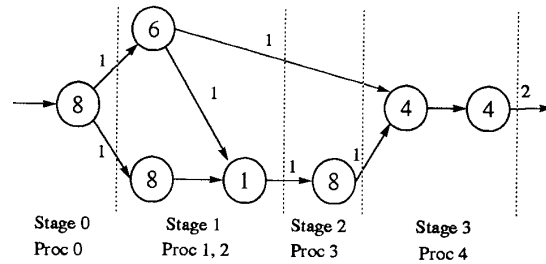


Figure 4: Partition() given stagetime $T = 10$

loop pipelining, or any combination of them in a unified manner. A detail presentation of the scheduling algorithm and the node decomposition transformations is given in [7].

5. CODE GENERATION

The *Code Generator* module takes the decorated CDFG, and using the same architectural description, generates a separate program for each processor. Each processor executes its program in an infinite loop, synchronizing once at the beginning of every sample. The code generation consists of two steps: Memory mapping, and code emission.

5.1. Memory Mapper

The memory mapper scans the decorated CDFG for edges whose input and output nodes are assigned to different processors. These represent interprocessor communications. They are implemented as circular buffers whose sizes are determined by the amount of data sent and the pipeline stage of the source and destination processors. Their physical residence depends on whether the system support message passing or shared memory communication and whether the memory is centralized or distributed. For message passing or distributed memory multiprocessors, these buffers would reside in the local memory of the destination processor. Thus, a write would go through the bus network, but the corresponding read would be from local memory. This can greatly reduce bus contentions. For centralized memory systems, the buffers would reside in the central memory, and reading and writing will access the bus. If these systems support caching, these buffers may be brought into the cache, effectively yielding a distributed memory system. Buffers for data broadcast across several pipeline stages only need to be allocated once in a centralized memory system, but have to be replicated in a distributed system. Figure 5 shows how these buffers are derived from the CDFG of Figure 4.

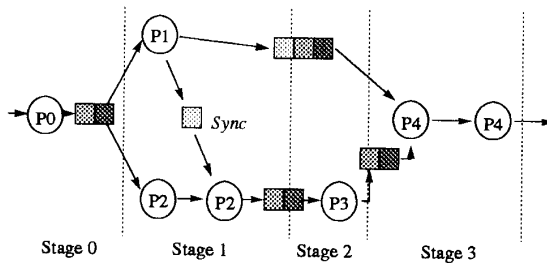


Figure 5: Buffer Allocation

The length of the buffer is determined as $\text{PipeStage}(P_{\text{dest}}) - \text{PipeStage}(P_{\text{src}}) + 1$. A pointer keeps track of the current free data block. The source processor always writes to the free block while

the destination processor always reads from the oldest block. If all processors globally synchronize at the end of each sample period, this scheme allows the two processors to communicate without ever corrupting data. When two processors in the same pipeline stage need to communicate, the length of the buffer is one and the processors need to locally synchronize to ensure no data corruption. Note that if it was possible to estimate the computation and communication times exactly, no local synchronizations would be necessary as the nodes are only scheduled after all input data have arrived.

5.2. Code Emission

At present, C code is generated. The code is parameterized to perform floating point or bit-true simulation depending on which library is included. Great care has been taken to ensure the code is as efficient as possible. Two multiprocessor systems have been targeted, the first one targeting the SMART[8] system with 10 DSP32 processors and the second one addressing the Sequent multiprocessor containing 14 Intel 386 processors. Real-time implementation on DSP's is currently supported with commercial C compilers. We are investigating the direct generation of DSP code from the flowgraph description[9].

6. RESULTS

A few designs are presented to demonstrate the basic capabilities of McDAS. The target architecture is the Sequent system, with 14 processors. The first is a histogram application, which involves classifying 128 samples into 32 subclasses. The schedule on 6 processors was shown in Figure 2, where a parallel loop was executed in pipeline and parallel. Figure 6(a) shows the loads are well balanced, both in the estimated computation time, and in the time actually measured of the Sequent. Figure 6(b) shows the

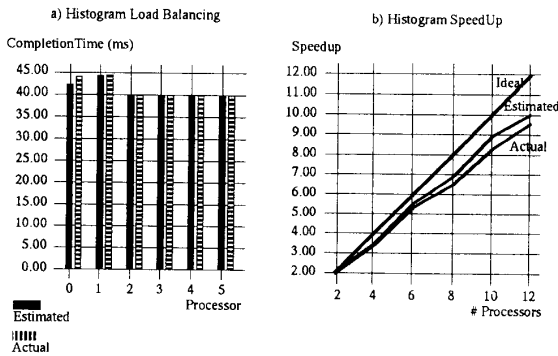


Figure 6: Partition() given stagetime T = 10

speedup across different processor implementations. As we can see, each additional processor improves the speedup.

Table 1 shows the performance of the scheduler on a variety of DSP examples with different types of concurrency and communication patterns. All assume that the number of available processors is 8, and a single shared bus interconnection. $H(G)$, $F(G)$, and $R(G)$ give the number of nodes in the hierarchical graph, in the flattened graph, and in the graph that the algorithm had to consider during its search. The percent communication tells how much of the processing time was spent on communication. The CPU measurements is on a Sun Sparc II, and include the Silage to Flowgraph compilation. The Discrete Fourier Transform (DFT) algorithm uses a recursive computation of the coefficients and is pipelined into 8 stages. The Dynamic Time Warp (DTW) algorithm for speech recognition compares 1000 templates to an unknown signal, and is data partitioned into 8 processors working

in parallel. The matrix multiplication example multiplies two 64x64 matrices, and is data partitioned into 8 processors working in parallel. All of these examples are computationally intensive, very regular, and require little inter-processor communications. Hence, near optimal speedup was achieved. The Cordic I algorithm converts cartesian to polar coordinates iteratively, and is pipelined into 7 stages, with the first stage having 2 processors in parallel. The same example Cordic II is partitioned on a ring architecture. Since the communication pattern is primarily between neighbor processors, the ring architecture reduces the bus congestion and improves the speedup by 18%. The next 3 examples: The Echo Canceller, the Adaptive Differential Pulse Code Modulator (Adpcm), and the Decision Feedback Equalizer (DFE) all contain single delay recursions in their computation, which prohibited pipelining. As a result, the speedup was not optimal, even though all the available concurrency was being exploited. Specifically, in the Echo Canceller and DFE examples, the partitioning algorithm was able to exploit the parallelism in the cycle to obtain a significant speedup in an otherwise serial computation. Faster speedup for these examples can only be obtained by transforming or reorganizing the algorithms themselves.

| Example | # Iter | F(G) | H(G) | R(G) | SpeedUp | Concurrency Exploited | % Comm | CPU (sec) |
|-----------------|--------|--------|------|------|---------|-----------------------|-------------|-----------|
| DFT | 11 | 9.5e5 | 67 | 10 | 7.99 | Pipelining | 1.67 | 4.1 |
| Pitch Extractor | 16 | 1.23e5 | 263 | 21 | 7.20 | Pipe / Par | 0.70 | 9.8 |
| Dyn Time Warp | 10 | 1.69e8 | 94 | 8 | 8.00 | Parallelism | < 0.01 | 0.8 |
| Matrix Mult | 6 | 2.10e6 | 21 | 8 | 8.00 | Parallelism | < 0.01 | 6.9 |
| Cordic I | 15 | 2319 | 64 | 23 | 5.63 | Pipelining | 73.35 | 5.4 |
| Cordic II | 15 | 2319 | 64 | 23 | 6.87 | Pipelining | 15.00 / Bus | 4.9 |
| Echo Canceller | 14 | 4683 | 83 | 23 | 3.82* | Retime / Par | 25.28 | 10.5 |
| Adpcm | 13 | 440 | 345 | 49 | 2.11* | Retime / Par | 40.34 | 34.7 |
| DFE | 9 | 508 | 114 | 43 | 5.07* | Retime / Par | 56.39 | 118.2 |

*: Flowgraph contains cycles

Table 1: Scheduling Results.

References:

- [1] E.A. Lee, et al, "Gabriel: A Design Environment for DSP," *IEEE Trans. on ASSP*, vol. 37, no. 11, Nov, 1987.
- [2] M.A. Zissman, et al, "A Block Diagram Compiler for Digital Signal Processing MIMD Computer," *Proc. ICASSP'87*, Texas.
- [3] H. Printz, "Automatic Mapping of Large Signal Processing Systems to a Parallel Machine," Ph.D. Thesis, Carnegie Mellon University, May 1991.
- [4] D.A. Schwartz, Barnwell, Hodges, "The optimal Synchronous Cyclo-static Array: A Multiprocessor Supercomputer for Digital Signal Processing," *Proc. ICASSP'86*, Tokyo, Japan, April, 86.
- [5] "MultiProx," Comdisco Systems Inc.
- [6] P. Hilfinger, "Silage, A High Level Language and Silicon Compiler for Digital Signal Processing," *Proceedings IEEE CICC Conference*, Portland, May 1985.
- [7] P. D. Hoang, J. Rabaey, "Program Partitioning for a Reconfigurable Multiprocessor System," *IEEE Workshop on VLSI Signal Processing IV*, San Diego, Nov. 1990.
- [8] W. Koh, and A. Yeung, P. Hoang, J. Rabaey, "A Configurable Multiprocessor System for DSP Behavioral Simulation," *ISCAS Symposium*, May 1989
- [9] D. R. Genin, Moortel, Desmet, Van de Vlede, "System Design, Optimization, and Intelligent Code Generation for Standard DSP," *ISCAS 89*, Portland.